LA-UR -92-562

LA-UR--92-562

DE92 008476

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE SUPERCOMPUTER DEBUGGING WORKSHOP '91 PROCEEDINGS

AUTHOR(S) J. Brown

SUBMITTED TO Supercomputer '91 Debugging Workshop,
Albuquerque, NM
November 14-16, 1991

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof

MASTER

Los Alamos
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Supercomputer Debugging Workshop '91
Proceedings

Albuquerque, New Mexico
November 14-16, 1991

FOREWORD

The Supercomputer Debugging Workshop '91 (SD '91) was held the week prior to Supercrmputing '91, and focused upon topics relating to debugger construction and usage in the Supercomputer programming environment. The workshop brought together debugger developers and users to discuss topics and experiences of mutual interest, and established a basis for future collaborations.

The objective of the workshop was to promote a free and open exchange of information between an interdisciplinary group of debugger developers and users from the commercial and academic communities, thereby advancing the state-of-the-art of debugger technology.

Program Chair:

Jeff Brown, *Los Alamos National Laboratory*

Local Arrangements:

Denise Dalmas, *Los Alamos National Laboratory*

Program Committee:

Bruce Kelly, *NERSC*
Alan Riddie, *NERSC*
Peter Rigsbee, *Cray Research*
Larry Streepy, *Convex*
Rich Title, *Thinking Machines*
Ben Young, *Cray Computer*

Keynote Speaker:

Ken Kennedy, *Rice University*

Table of Contents

Keynote Address:

Large Group Presentations:

Distributed Debugging:

Design of a Debugger for a Heterogeneous Distributed System
Arjun Khanna, ***Experimental Systems Lab, MCC***

Debugging in a Loosly Coupled Heterogeneous Computing Environment: A Case Study
Matt Kussow, ***Superconcurrency Research Team, Naval Ocean System Center***

Designing CDS: an On-Line Debugging System for the C_NET Programming Environment
Pierre Moukeli, ***Laboratoire de l'Informatique du Parallelisme***

User Interface to Debugging Tools and Standards:

DBL: An Interactive Debugging System
Mukkai S. Krishnamoorthy, ***Rensselaer Polytechnic Institute***

X Window System Interface for CDBX
Peter A. Rigsbee, ***Cray Research, Inc.***

DWARF: A Debugging Standard
Janis Livingston, ***Motorola, Inc.***

Watson: A Graphical User Interface Environment for Debugger Development
Randy Murrish, ***Cray Computer Corporation***

Debugging Optimized Code:

Debugging Optimized Code Without Surprises
Max Copperman, ***University of California at Santa Cruz***

The Symbolic Debugging of Code Transformed for Parallel Execution
Patricia Prather Pineo, ***Allegheny College***

Intermediate Languages for Debuggers
Benjamin B. Chase, ***Department of Computer Science, Rice University***

Debugging Parallel Codes:

MDB - A Parallel Debugger for Cedar
Bret Marsolf, ***Center for Supercomputing Research and Development, U of Ill.***

A Replay mechanism within an environment for distributed programming
S. Chaumette, ***Universite Bordeaux-I***

An Integrated Approach to Replay Analysis of Message-Passing Parallel Programs
Chad Hunter, ***The MITRE Corporation***

On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism
Robert Hood (for John Mellor-Crummey), ***Rice University***

Block-Structured Control of Parallel Tracing
Cherri M. Pancake, ***Auburn University***

Debugger Performance and Interface at Analysis Tools:

An Object-Oriented Design of a Debugger with *undo*
Robert Hood, ***Rice University***

Debugging with Lightweight Instrumentation
Benjamin Chase, ***Rice University***

Integration of Performance Analysis and Debugging
Marty Itzkowitz, ***Silicon Graphics Computer Systems***

Managing Debugger Process Execution: A Finite State Machine Approach
Paul A. Sanville, ***Silicon Graphics Computer Systems***

A Visual Debugger Constructed by Program Generating Technique
Ming Zhao, ***CS Division, Asian Institute of Technology***

Interactive Steering Using the Application Executive
Brian Bliss, ***Center for Supercomputing Research and Development, U of Ill.***

Summary:

"Dream Debugger"
Charlie McDonald, ***University of California at Santa Cruz***

Attendees at the Supercomputer '91 Debugging Workshop
November 14-16, 1991

Daniel Bates
US Government CIA
Rm. 2V29, Bldg. NHB
Washington, DC 20505
EMAIL ADDRESS:

Kent Beck (speaker)
MasPar Computer Corp.
749 N. Mary Ave.
Sunnyvale, CA 94086
EMAIL ADDRESS: kentb@maspar.com

John Blaylock
C-10. MS B296
LANL
Los Alamos, NM 87545
EMAIL ADDRESS: jwb@lanl.gov

Brian Bliss (speaker)
Center for Supercomputing Research
and Development
University of Illinois-Urbana
104 S. Wright St.
Urbana, IL 61801
EMAIL ADDRESS: bliss@csrd.uiuc.edu

Don Breazeal
Intel Supercomputer Systems
Division (SSD), MS CO1-01
15201 N.W. Green Brier Pkwy
Beaverton, OR 97006
EMAIL ADDRESS: donb@ssd.intel.com

Jeff Brown (Chairperson)
C-10, MS B296
LANL
Los Alamos, NM 87545
EMAIL ADDRESS:jxyb@lanl.gov

Karla A. Callaghan
Intel Supercomputer Systems Div.
15201 N.W. Greenbrier Pkwy, MS CO1/01
Beaverton, OR 97006
EMAIL ADDRESS: karla@ssd.intel.com

Ann Mei Chang (speaker)
Silicon Graphics
2011 N. Shoreline Blvd.
Mountain View, CA 94039-7311
EMAIL ADDRESS: ann@sgi.com

Ben Chase (speaker)
CITI, Rice University
P.O. Box 1892
Houston, TX 77251-1892
EMAIL ADDRESS: bbc@rice.edu

Serge Chaumette (speaker)
LaBRI, Universite Bordeaux I
351 Cours de la Liberation
33405 Talence, FRANCE
EMAIL ADDRESS: chaumette@gecocub.greco-prog.fr

Doreen Cheng
NASA Ames Research Ctr
MS/258-6
Moffett Field, CA 94035
EMAIL ADDRESS:dcheng@nas.nasa.gov

Jim Christensen
Office H1-C16
IBM T. I. Watson Research Ctr
P.O. Box 704
Yorktown Hts, NY 10598
EMAIL ADDRESS: jimc@watson.ibm.com

Johnny L. Collins
X-7, MS B257
LANL
Los Alamos, NM 87545
EMAIL ADDRESS: juan@lanl.gov

Max Copperman (speaker)
Computer and Information Services
University of California - Santa Cruz
G3 Koshland Way
Santa Cruz, CA 95064
EMAIL ADDRESS: max@cis.ucsc.edu

Alva Couch (speaker)
Tufts University
Dept. of Computer Science
Medford, MA 02155
EMAIL ADDRESS: couch@cs.tufts.edu

Jan Cuny
Dept. of Computer Science
Univ. of Washington
Seattle, WA 98195
EMAIL ADDRESS:

George Dalmas
US Government CIA
Rm. 2V29, Pldg. NHB
Washington, DC 20505
EMAIL ADDRESS:

John Delsignore
BBN Systems and Technologies
10 Moulton St.
Cambridge, MA 02138
EMAIL ADDRESS: jdelsign

Ray Glenn
Supercomputing Research Ctr.
17100 Science Dr.
Bowie, MD 20715-4300
EMAIL ADDRESS: glenn@super.org

Howard Gordon
Supercomputer Research Ctr
17100 Science Dr.
Bowie, MD 20714-4300
EMAIL ADDRESS: flash@super.org

Ron Guilmette
Ron Guilmette Computing
396 Ano Nuevo Ave., #216
Sunnyvale, CA 94086
EMAIL ADDRESS:rfg@icd.com:

Carolynn Hakansson
Verdix Corp.
1600 NW Compton Drive #357
Aloha, Oregon 97006-6905
EMAIL ADDRESS: carolyn@verdix.com

Ken Hansen
National Ctr. for Atomspheric
  Research
Boulder, CO
EMAIL ADDRESS:

Charles Haynes
Digital Equipment Corp.
305 Lytton Ave.
Palo Alto, CA 94070
EMAIL ADDRESS: haynes@wsl.pa.dec.com

Anthony Hefner
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
EMAIL ADDRESS: sasarh@dey.sas.com

Robert Henry
Tera Computer
400 N 34th #300
Seattle, WA 98103
EMAIL ADDRESS: rrh@tera.com

Mike Hester
Intel
5200 NW Elan Young Pkwy
CO4-02
Hillsboro, OR 97124
EMAIL ADDRESS: meh@iwarp.intel.com

Sue Utter Honig (speaker)
Theory Center
737 E and TC Bldg.
Cornell University
Ithaca, NY 14853-3801
EMAIL ADDRESS: psu@cornellf.tc.cornell.edu

Robert Hood (speaker)
CITI
Rice University
P.O. Box 1892
Houston, TX 77251-1892
EMAIL ADDRESS: hood@rice.edu

Bob Hotchkiss (speaker)
LANL
X-7, MS B257
Los Alamos, NM 87545
EMAIL ADDRESS: rsh@lanl.gov

IBM Palo Alto Scientific Ctr.
MS 35, 1530 Page Mill Rd.
Palo Alto, CA 94304
EMAIL ADDRESS: lohsieh@paloalto.vnet.ibm.com

Chad D. Hunter (speaker)
The MITRE Corporation
Burlington Road
Bedford, MA 01730
EMAIL ADDRESS: chad@linus.mitre.org

Marty Itzkowitz (speaker)
Silicon Graphics, Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94039-1980
EMAIL ADDRESS: martyi@wpd.sgi.com

Michael Karr
Software Options, Inc.
22 Hilliard St.
Cambridge, MA 02138
EMAIL ADDRESS: mike@soi.com

Bruce Kelly (Committee)
National Energy Research
  Supercomputer Ctr.
L-561, P.O. Box 5509
Livermore, CA 94551
EMAIL ADDRESS: kelly@nersc.gov

Ken Kennedy (Keynote Speaker)
Rice University
P.O. Box 1892
Houston, TX 77251-1892
EMAIL ADDRESS: ken@rice.edu

Zahira S. Khan
Dept. of Mathematics &
  Computer Science
Bloomsburg University
Bloomsburg, PA 17815
EMAIL ADDRESS:

Arjun Khanna (speaker)
Experimental Systems Lab.
MCC
3500 West Balcones Ctr. Drive
Austin, TX 78759-6509
EMAIL ADDRESS: arjun@mcc.com

Rance Kirtley
AT&T Bell Labs
Rm 14 B267
1 Whippany Rd.
Whippany, NJ 07981
EMAIL ADDRESS: attbl!homxc!kirtley

Richard Klamann
C-10, MS B296
LANL
Los Alamos, NM 87545
EMAIL ADDRESS: rmk@lanl.gov

Dept. of Computer Science
Rennsselaer Polytechnic Institute
Troy, NY 12180
EMAIL ADDRESS: moorthy@turing.cs.rpi.edu

Matt Kussow (speaker)
Naval Ocean Systems Center
Code 421, Bldg. 606
San Diego, CA 92152-5000
EMAIL ADDRESS: kussow@nosc.mil

Yeon-Jae Lee
Jumin Dev. Dept
NAIS Development Division
140-716 Dacom Bldg
65-228 3-GA HANGANGRO
Yongsan-Ku
SEOUL, KOREA
EMAIL ADDRESS:

Janis Livingston (speaker)
Motorola, Inc.
MMTG
6501 William Cannon Drive West
Austin, TX 78735-8598
EMAIL ADDRESS: janisl@oakhill.sps.mot.com

Louis Lopez
IBM Scientific Center
1530 Page Mill Rd.
Palo Alto, CA 95014
EMAIL ADDRESS: lopez@paloalto.linus1.ibm.com

Jack MacDonald
Microtec Research
2350 Mission College Blvd.
Santa Clara, CA 95054
EMAIL ADDRESS: lopez@paloalto.linus1.ibm.com

Ruth Ann Manning
Oak Ridge National Lab.
P.O. Box 2008
MS 6397, Bldg., 6026-B
Oak Ridge, TN 37831
EMAIL ADDRESS: rm6@ornl.gov

Bret Marsolf (speaker)
University of Illinois - Urbana
305 Talbot Laboratory
104 South Wright Street
Urbana, IL 61801
EMAIL ADDRESS: marsolf@csrd.uiuc.edu

Charles McDowell
Computer and Information Sciences
Univ. of California - Santa Cruz
Santa Cruz, CA 95064
EMAIL ADDRESS: charlie@cis.ucsc.edu

Dennis Moen
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
EMAIL ADDRESS: drm@cray.com

[illegible]
Department of Defense
9800 Savage Road
Fort Meade, MD 20755-6000
ATTN: T3
NO EMAIL ADDRESS

Pierre Moukeli (speaker)
Laboratorie de l' Informatique
  du Parallelisme
ENS-Lyon
46 Allee d' Italie
69364 Lyon Cedex 07, FRANCE
EMAIL ADDRESS: moukeli@lip.ens-lyon.fr

Khalid A. Mughal
Cornell University
Dept. of Computer Science
Upson Hall
Ithaca, NY 14853
EMAIL ADDRESS: khalid@cs.cornell.edu

Randy Murrish (speaker)
Cray Computer Corp.
1110 Bayfield Drive
Colorado Springs, CO 80906
EMAIL ADDRESS: mush@craycos.com

Tom Myers (speaker)
National Security Agency
9800 Savage Rd., P1
Ft. Meade, MD 20755
EMAIL ADDRESS: ctmyers@super.org

Kelly O'Hair
Supercomputer Systems Inc.
2021 Las Positas Court
Suite 101
Livermore, CA 94550
EMAIL ADDRESS: uunet!ssi!ohair

Wendy Palm
Department of Defense
9800 Savage Road
Fort Mead, MD 20755-6000
NO EMAIL ADDRESS

Cherri Pancake (speaker)
Dept. of Computer Engineering
Auburr University
Auburn, AL 36849
EMAIL ADDRESS: pancake@eng.auburn.edu

Patricia Pineo (speaker)
Dept. of Computer Science
Allegheny College
Meadville, PA 16335
EMAIL ADDRESS: ppol@music.alleg.edu

Arlan Pool
Mercury Computer Systems
600 Suffolk Street
Lowell, MA 01854
EMAIL ADDRESS: uunet!mercomp!alp

Robert Rapson
US Government CIA
Rm. 2V29 Bldg. NHB
Washington, DC 20505
EMAIL ADDRESS:

James D. Reed
CONVEX Computer Corp.
3025 South Parker Rd.
Suite 109
Aurora, CO 80014
EMAIL ADDRESS: jdreed@convex.comp

Alan Riddle (Committee)
NERSC, LLNL
P.O. Box 5509, L560
Livermore, CA 94550
EMAIL ADDRESS: riddle@winddune.nersc.govn

Peter Rigsbee (committee)
Cray Research, Inc.
655F Lone Oak Dr.
Eagan, MN 55121
EMAIL ADDRESS: par@cray.com

Paul A. Sanville (speaker)
Silicon Graphics, Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94039
EMAIL ADDRESS: sanville@wpd.sgi.com

Leslie Scarborough
IBM Palo Alto Science Ctr.
1530 Page Mill Road
Palo Alto, CA 94304
EMAIL ADDRESS: toomey@paloalto.ibm.com

Tony Sloane
Colorado University
1300 30th St., #B2-31
Boulder, CO 80303
EMAIL ADDRESS: tony@cs.colorado.edu

William Spangenberg
LANL
X-7, MS B257
Los Alamos, NM 87545
EMAIL ADDRESS: whs@lanl.gov

Jeff Spencer
US Government CIA
Rm. 2V29, Bldg. NHB
Washington, DC 20505
EMAIL ADDRESS:

Karen Spohrer (speaker)
Motorola, Inc.
MMTG
6501 William Cannon Dr. West
Austin, TX 78735-8598
EMAIL ADDRESS: karens@oakhill.sps.mot.com

DEC
K6 Main St (ML01-31B11)
Maynard, MA 01754
EMAIL ADDRESS:decwrl::"mugur_s@rdvax.dec.com":

Al Stipek
Cray Research
1440 Northland Drive
Mendota Heights, MN 55120
EMAIL ADDRESS:

Jeff Stoddard
C-10, MS B296
LANL
Los Alamos, NM 87544
EMAIL ADDRESS: jys@lanl.gov

Larry Streepy (Committee)
Convex Computer Corp.
3000 Waterview Pkwy
P.O. Box 833851
Richardson, TX 75083-3851
EMAIL ADDRESS: streepy@convex.com

Sandra Swanson
Cray Research, Inc.
655 Lone Oak Drive
Eagan, MN 55121
EMAIL ADDRESS: ss@cray.com

Jim Tabor
C-10, MS B296
LANL
Los Alamos, NM 87545
EMAIL ADDRESS: jet@lanl.gov

Richard Title (Committee)
Thinking Machines Corp.
245 First Street
Cambridge, MA 02142-1264
EMAIL ADDRESS: title@think.com

Harold Trease (speaker)
LANL
X-7, MS B257
Los Alamos, NM 87545
EMAIL ADDRESS: het@lanl.gov

Wheels VanderWeele
Verdix Corp.
1600 NW Compton Drive #357
Aloha, OR 97006
EMAIL ADDRESS: wheels@verdix.com

Cheryl Wampler (speaker)
C-10, MS B296
LANL
Los Alamos, NM 87545
EMAIL ADDRESS: clw@lanl.gov

Nancy Werner
LLNL
P.O. Box 808, MS L300
Livermore, CA 94550
EMAIL ADDRESS:

Elizabeth Williams
Supercomputer Research Ctr.
17100 Science Drive
Bowie, MD 20715-4300
EMAIL ADDRESS: ew@super.org

Joe Wolf
Cray Research
500 Montezuma #118
Santa Fe, NM 87501
EMAIL ADDRESS: jhw@zia.cray.com

Sasan Yaghmaee
Boeing Computer Services
P.O. Box 24346
Seattle, WA 98124-0346
EMAIL ADDRESS:

Ben Young (Committee)
Cray Computer Corp.
1110 Bayfield Dr.
Colorado Springs, CO 80906
EMAIL ADDRESS:bby@craycos.com

Bing Young
Cray Research
LLNL
P.O. Box 808
Livermore, CA 94550
EMAIL ADDRESS: bing@craywn.com

Ming Zhao
Asian Institute of Technology
Division of Computer Science
P.O. Box 2754
Bangkok 10501, THAILAND
EMAIL ADDRESS: zm%ait.ait.th%munnari.oz.au

DEBUGGING AND THE TERAFLOP COMPUTER

Ken Kennedy
Center for Research on Parallel Computation
Rice University
Houston, Texas

ABSTRACT

November 26, 1991

In a very real sense, the debugger is the inverse of a compiler, because its job is to interpret the execution of the compiled program in a language close to the programming language in which the original program is expressed. The state of the art is "source-level" debugging in which the debugger uses the compiler symbol table to interpret an execution in the source language.

The quest for a teraflop machine will introduce new machine designs and corresponding compiler complexities that will significantly complicate the job of the debugger. The teraflop machine is almost certain to be a highly parallel machine (thousands of processors) in which each processor is a sophisticated commodity microprocessor. With the advent of 64-bit addressing, in microprocessors, these machines are likely to have hardware shared memory, although they will be packaged like distributed-memory machines.

The compilers for parallel machines will introduce enormous complexities for the debugger because they will employ sophis ated transformations to enhance single-processor scheduling, parallelism and memory hierarchy age. Compilers are also likely to employ interprocedural optimizations.

As a result of these developments, future debuggers will be presented with three major challenges:

1. **reconstruction of program state in the presence of advanced optimizations,**
2. **location of schedule-dependencies (date races) in explicitly parallel programs and,**
3. **analysis and visualization of performance in highly optimized parallel programs employing thousands of processors.**

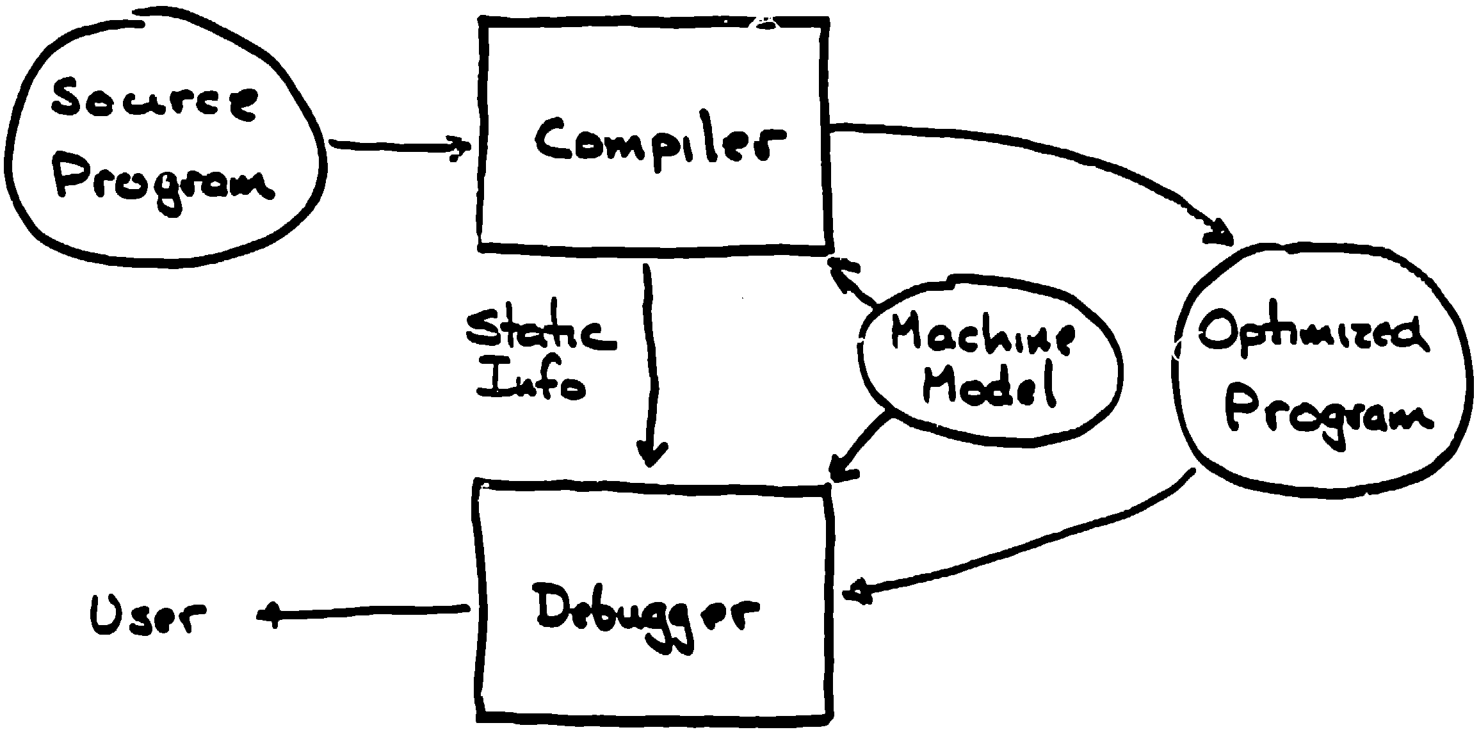
Debugging and the Teraflop Computer

Ken Kennedy

Center for Research on Parallel Computation

1. Debugging
2. Teraflop system
3. Languages
4. Debug challenges

Debugging



- Algorithm
  - does implementation meet specs
  - error
- Program Logic
  - mixed static and dynamic methods
- Parallelism
  - schedule-dependent errors
  - static and dynamic methods
- Performance
  - understanding program behavior

Teraflop Computing

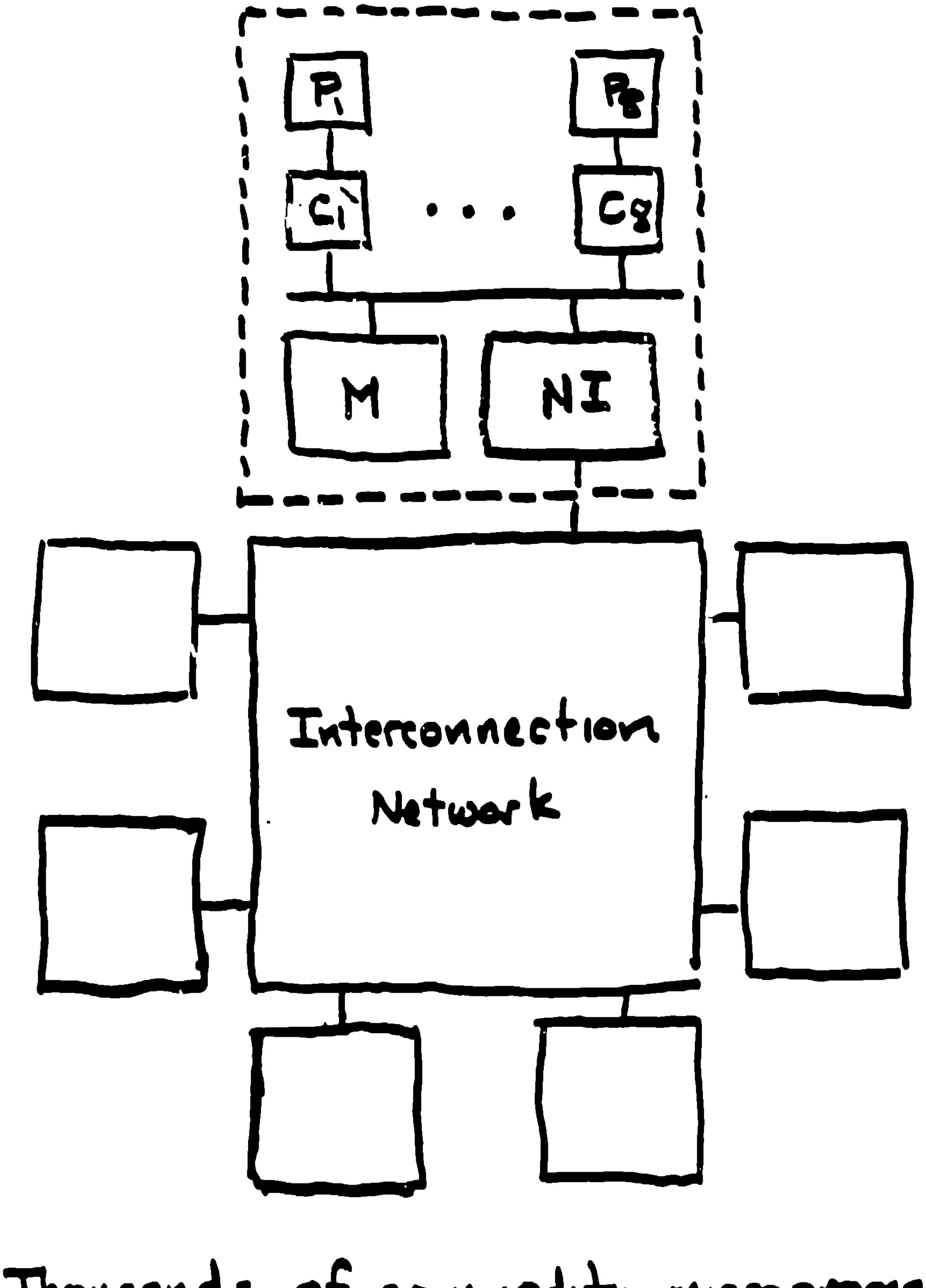
Goal: sustained teraflop by 2000

Strategy:

- Commodity microprocessors
    - 500 megaflops by 1995 (Intel)
- Interconnection network
    - high bandwidth
    - low latency

⇒ ⩾ 8000 processors

Future Parallel Machines



- Thousands of commodity microprocessors
- Distributed memory, shared address space

Advanced Compiler Techniques

- Pipeline scheduling
  - reordering instructions

- Compiler memory hierarchy management

```
DO I = 1, N                  DO I = 1, N, 2
  DO J = 1, N                  DO J = 1, N
    A(I) = A(I) + B(J)           A(I) = A(I) + B(J)
  ENDDO                          A(I+1) = A(I+1) + B(J)
ENDDO                          ENDDO
                             ENDDO
```

  - speed-up: x2-3

- Interprocedural Optimization
  - code transformations across procedure boundaries

Multiprocessor Cache Management

PROCESSOR 1
CACHE

```
PARALLEL DO I = 1,2
    A(I) = I
END PARALLEL DO
```

A(1) = 1

```
PARALLEL DO I=1,2
    A(3-I) = I
    UPDATE A(3-I)
END PARALLEL DO
```

A(1) = 1
A(2) = 1

```
PARALLEL DO I=1,2
   INVALIDATE A(I)
          = A(I)
END PARALLEL DO
```

A(1) = 1 on P1
WRONG

Fortran-Based Languages

- Fortran 77
- Message-passing Fortran
- Parallel-loop Fortran
  - PCF Fortran
- Fortran 90
- Fortran D

Fortran 77

Idea: rely on automatic parallelization for all parallel programming

Lesson of vectorization

- Advantages
  - ease of use
  - deterministic programs
  - portability
- Disadvantages
  - generality
  - moderately hard debugging
    - display of parallel execution in seq. language

Message-Passing Fortran

Idea: making processes independent (non-intersecting address spaces)
— communication by explicit message passing

```
SEND A(1:N) TO P(IAM+1)
RECEIVE T(1:N) FROM P(IAM-1)
```

Advantages:

- ease of implementation

Disadvantages

- hard to use
- mildly non-deterministic

PCF Fortran

Idea: specify loop and case parallelism in a shared-memory environment

```
PARALLEL DO I = 1, N
        ;
 END PARALLEL DO
```

Advantages

- relatively easy to use shared memory

Disadvantages

- wildly nondeterministic
- complexity of sharing

Non-Determinism

Problem: in PCF Fortran correct (deterministic) programs cannot be distinguished at compile time from an incorrect (nondeterministic) program.

```
PARALLEL DO I = 1, N
    X(N(I)) = X(N(I)) + 10
END PARALLEL DO
```

Corectness depends on values in N — may only be known at run time

Fortran 90 Array Language

Idea: multidimensional array assignments.

```
A(1:N, 1:M) = A(1:N, 1:M) + 100.0
```

Advantages

- simple explicit parallelism
- deterministic

Disadvantages

- generality
- debugging moderately difficult
- no data layout

Fortran D

Idea: add data layout stmts to F77 or F90.

```
ALIGN A(I,J) WITH D(I,J+1)
DISTRIBUTE D(BLOCK,*)
```

Advantages

- data layout
- deterministic (mostly)

Disadvantages

- complicated compilation
- debugging tricky
  performance
  optimization

Debugger Challenge 1:
State Reconstruction

Problem: to reconstruct user state in presence of complex optimizing transformations.

- Pipelining
- Memory hierarchy management
- Interprocedural optimization
- Parallelizing transformations
- Compiler coherence

Debugger Challenge 2:

Finding Schedule Dependences

Problem: location of sources of nondeterminism.

Obstacle: probe effect makes it difficult to recreate error.

Methods:

- Multiple-window
- Tracing and postmortem analysis
- On-the-fly methods

Parallel Debugging

Current Products:

- extension of Unix debugger to multiple processes
- ability to stop and single-step one or more processes
- one window per process

Not Good Enough!

Problem: location of schedule-dependent errors.

Post-Mortem Analysis

Goal: trace enough information to be able to reconstruct error schedule or find bug.

- trace all "significant" events
  - done all the time
- analyze trace post-mortem to reconstruct schedule or find bug

Disadvantage: overhead or precision

On-the-fly Methods

Idea: modify program to check all references to shared data to determine whether previous reference was to a concurrent thread.

Advantages

- catches all anomalies for the data set and schedule
- only shows real anomalies

Disadvantages

- space
- execution time
- data dependent
- schedule dependent

ParaScope Parallel Debugger

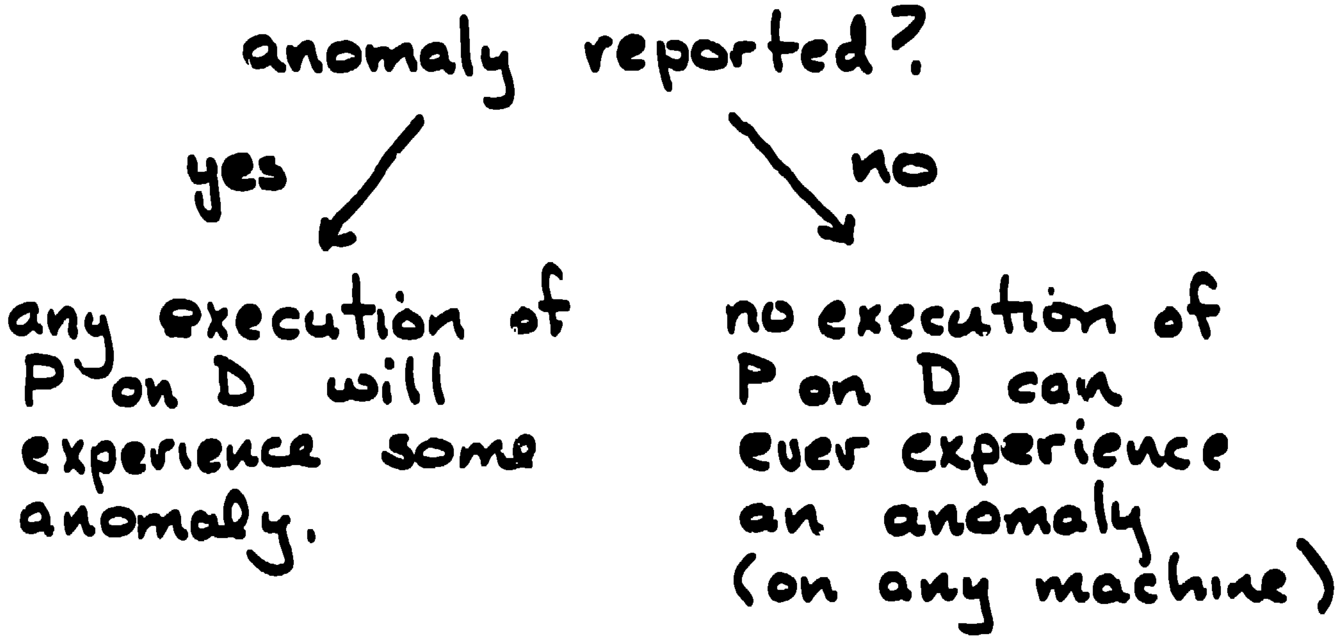
- On-the-Fly Race Detection
  - annotate dependence endpoints
  - report races as they occur
  - time and space overhead
    - shadow arrays for each potentially shared variable
- On-the-Fly, Post Mortem Debugging
  - schedule-invariant languages and programs
- Interprocedural Effects
  - annotations must be propagated interprocedurally
- Current Status

Guaranteed Replayability

- a property of parallel programs and parallel programming languages

let P be a program with guaranteed replayability

run P* (annotated version of program P) on input data set D



anomaly reported?

yes

any execution of P on D will experience some anomaly.

no

no execution of P on D can ever experience an anomaly (on any machine)

Debugger Challenge 3:

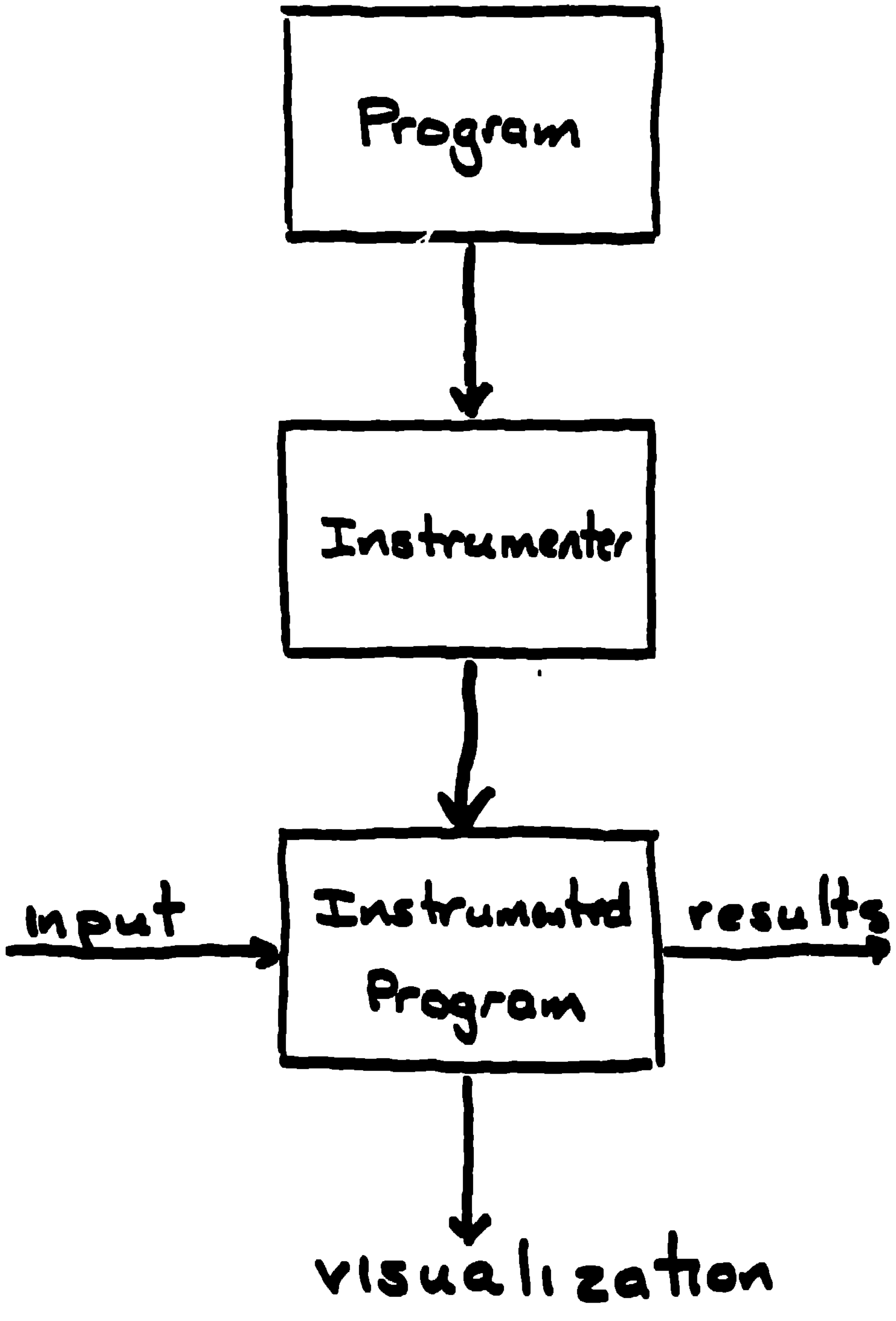
Performance Debugging

Problem: to devise ways to analyze and visualize the performance of thousands of processors.

Methods

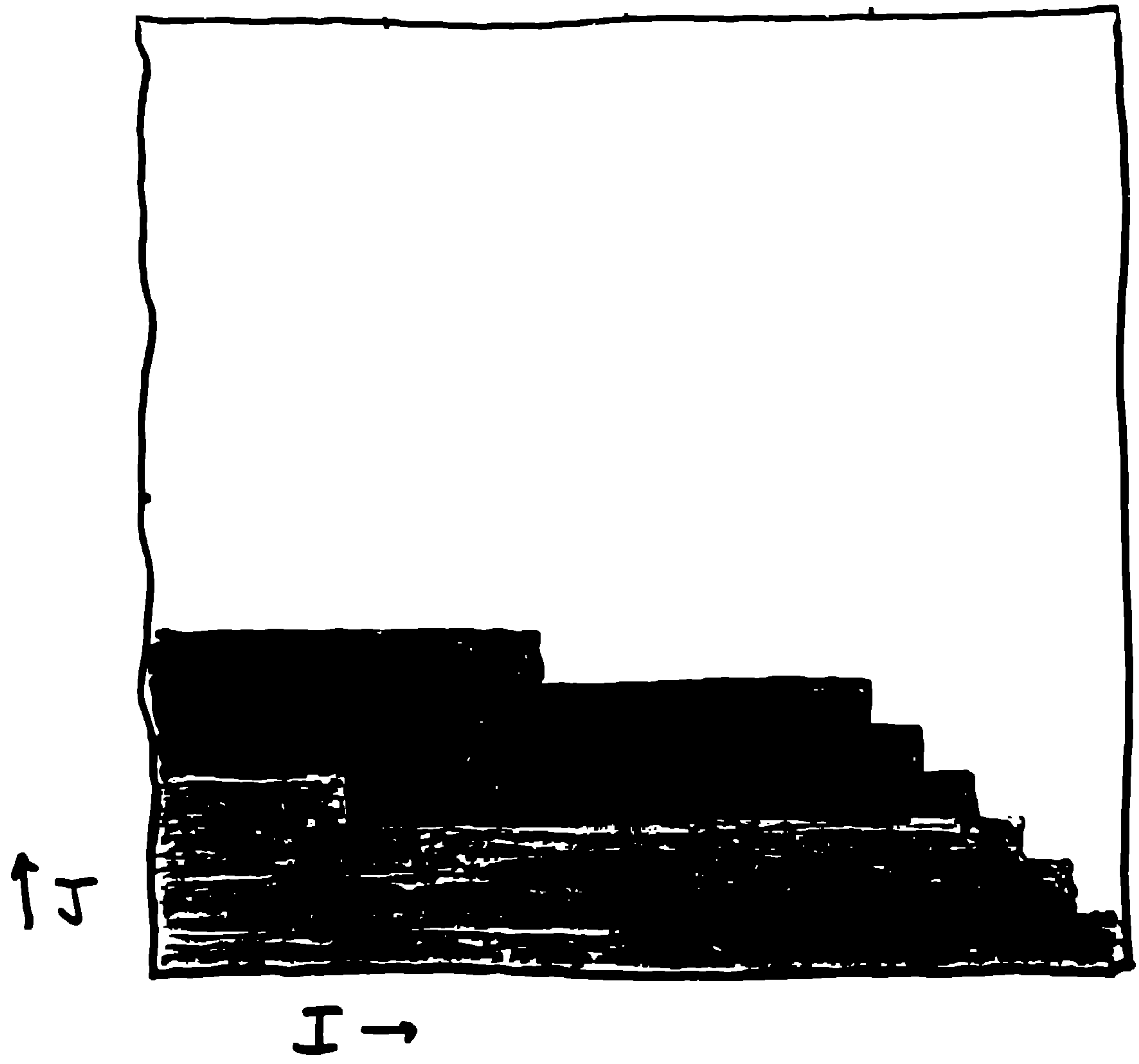
- trace analysis
- on-the-fly analysis
- hardware probes

Performance Visualization



- MAP - matrix access
- PFC-Sim - cache behavior

MAP (Dongarra, Sorensen)



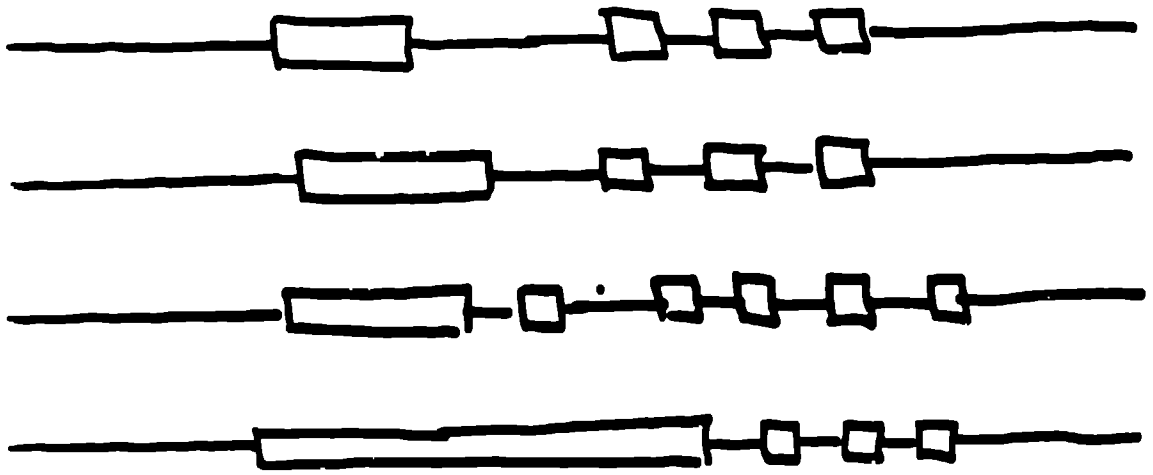
A(I,J)

Cache Browser

```
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
      ENDDO
   ENDDO
ENDDO
```

Visualization

Examples: Gist, Upshot



Problem: what happens with thousands of processors?

Summary

- Parallel systems are more complex
  - thousands of processors
  - memory hierarchies

- Compilers are becoming more complex
  - advanced transformations
  - parallelization
  - schedule-dependencies in languages

- Debugging will become more complex
  - state reconstruction
  - schedule dependence location
  - performance debugging

DEBUGGING PRODUCTION CODES

BY

HAROLD TREASE
COMPUTATIONAL PHYSICS GROUP (X–7)
LOS ALAMOS NATIONAL LABORATORY

USER

PERSPECTIVE

//1

MY DEBUGGING WORLD

THE WAY IT IS

THE WAY IT SHOULD BE

MY DEBUGGING WORLD

THE WAY IT IS:

- THINGS THAT MAKE LIFE BEARABLE
- THINGS THAT MAKE LIFE DIFFICULT
- THINGS THAT MAKE LIFE IMPOSSIBLE

THE WAY IT SHOULD BE:

- THINGS THAT (COULD) MAKE LIFE BETTER
- THINGS THAT WOULD MAKE LIFE GREAT
- THINGS THAT I DREAM OF

THE APPLICATION:

- UNSTRUCTURED MESHES
  - EVERYTHING IS DYNAMIC
  - NOTHING IS ASSUMED

THE DEBUGGING TOOLS:

- DDT, LDB AND CDBX (CRAYs)
- CMDBX AND PRISM (CMs)

THE WAY IT IS

MICROSCOPIC

RUN TIME PRINT STATEMENTS

NO KNOWLEDGE OF DATA STRUCTURES

EVERYTHING MUST BE VOLUNTEERED

THINGS THAT MAKE LIFE BEARABLE

USER CALLABLE ROUTINES
  USER PORTS

CHECK POINTING
  DO UNDO
  WHAT IFS

SOURCE LEVEL DEBUGGING

A HIGH LEVEL MACRO LANGUAGE

THINGS THAT MAKE LIFE DIFFICULT

PARALLEL PROCESSING

OPTIMIZED CODE

ASYNCHRONOUS I/O

THINGS THAT MAKE LIFE IMPOSSIBLE

NO AUTOMATIC CHECK POINTING FACILITY

A SIGNIFICANT PERFORMANCE HIT RELATED TO THE DEBUGGER

DEBUGGING: THE WAY IT SHOULD BE

NO DEBUGGING

A BETTER CLASS
OF CODE DEVELOPER

BETTER TOOLS

THINGS THAT (COULD) MAKE LIFE BETTER

INTEGRATED GRAPHICS AND DEBUGGER

INTEGRATED PERFORMANCE ANALYSIS
AND DEBUGGER

INTEGRATED DATA MANAGER
AND DEBUGGER

THINGS THAT WOULD MAKE LIFE GREAT

THINGS THAT I DREAM OF

HAVING DEBUGGER DEVELOPERS ACTUALLY USE THEIR CODES

HAVING DEBUGGER DEVELOPERS ACTUALLY ASK USERS WHAT THEY WOULD LIKE TO SEE

The Prism Programming Environment

Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, Steve Sistare, Rich Title

Authors' address:

Thinking Machines Corporation
245 First St.
Cambridge, Mass 02142

Email: allen@think.com, bowker@think.com, karen@think.com,
simons@think.com, sistare@think.com, title@think.com

ABSTRACT

The Prism programming environment is a graphical environment to support the development of Connection Machine programs. This paper discusses the design and implementation of Prism.

1. Goals of Prism

We set out a little over a year ago to build a Connection Machine programming environment meeting the following goals:

a. Make Connection Machine programmers more productive.

b. Support all Connection Machine programming models.

c. Provide a well-integrated set of tools.

d. Be easy to use.

e. Provide an attractive graphical interface that would demo well.

f. The environment should support multiple targets, e.g., both CM2 and CM5.

g. The work needed to be done on a fast schedule to have the environment ready in time for the CM5 announcement.

2. Overview of Prism

In the initial release of Prism we focused on the problems of debugging, performance analysis, and data visualization. We felt that by doing an excellent job in those areas we would provide the most benefit to users of the Connection Machine supercomputer. Prism deals with the other aspects of program development by providing interfaces to independent tools such as editors, the "make" utility, and online documentation.

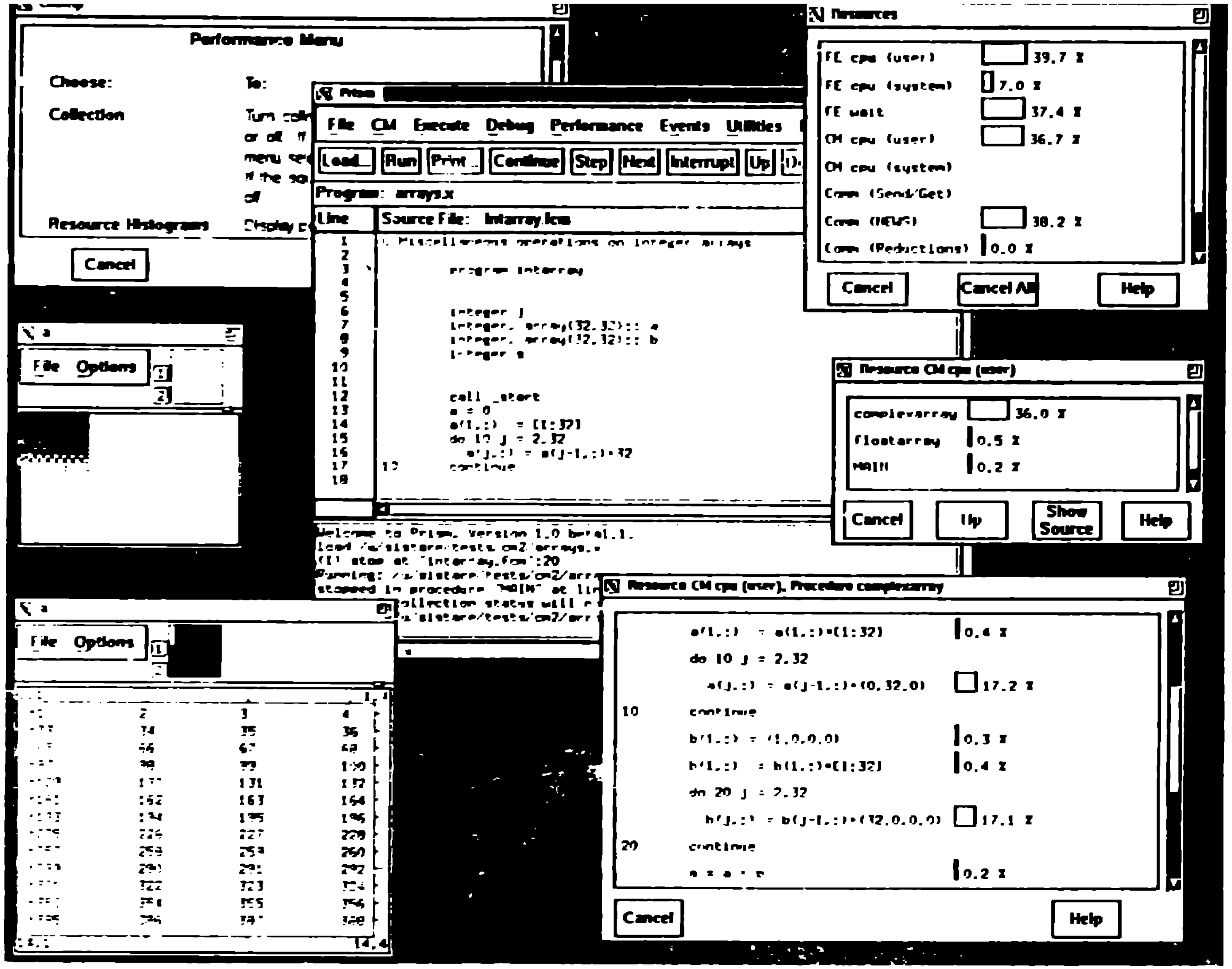
In the design of Prism, much emphasis was placed on providing an easy-to-use, intuitive, and attractive interface. Prism provides a point-and-click graphical interface, based on OSF/Motif. The design and implementation of the user interface is described in section 3.

Prism's debugging features include all the features of standard debuggers such as dbx, but in a graphical setting. In addition, Prism functionality goes beyond dbx in a number of ways. This will be described in detail in section 4.

Data-parallel programming usually involves the manipulation of large arrays, so the ability to visualize these arrays is important in a Connection Machine debugging environment. Section 6 describes Prism's data visualization capabilities.

Prism's performance-analysis features enable the user to find out where and how the program is spending its time. Prism goes beyond standard profilers such as 'prof' and 'gprof' in a number of ways: (1) Performance data is broken down according to what Connection Machine or front-end resource is being used, and (2) the resolution goes down to the source-line level. The performance analysis features are described in section 7.

At this point the reader may want to refer to figure 1, which shows a screen with some windows from the Prism environment. In the center is the main window, with the top-level pulldown menus, the source window, and the commands area. Surrounding that are various optional pop-up windows: A help window in the upper left, a couple of visualizers into the array "a" in the lower left, and some performance histograms on the right.



Program: arrays.x

Line Source File: Intarray.fcm

```
c Miscellaneous operations on integer arrays

        program intarray


        integer j
        integer, array(32,32):: a
        integer, array(32,32):: b
        integer s


        call _start
        a = 0
        a(1,:)  = [1:32]
        do 10 j = 2,32
          a(j,:) = a(j-1,:)+32
10      continue
```

```
Welcome to Prism, Version 1.0 beta1.1.
load /u/alstare/tests/cm2/arrays.x
(1) stop at "intarray.fcm":20
Running: /u/alstare/tests/cm2/arr
stopped in procedure "MAIN" at lin
collection status will n
/u/alstare/tests/cm2/arr
```

```
FE cpu (user)          39.7 %
FE cpu (system)        7.0 %
FE wait                37.4 %
CM cpu (user)          36.7 %
CM cpu (system)
Comm (Send/Get)
Comm (NEWS)            38.2 %
Comm (Reductions)      0.0 %
```

```
complexarray   36.0 %
floatarray     0.5 %
MAIN           0.2 %
```

```
        a(1,:)  = a(1,:)+[1:32]            0.4 %
        do 10 j = 2,32
          a(j,:) = a(j-1,:)+(0,32,0)       17.2 %
10      continue
        b(1,:) = (1,0,0,0)                 0.3 %
        b(1,:)  = b(1,:)+[1:32]            0.4 %
        do 20 j = 2,32
          b(j,:) = b(j-1,:)+(32,0,0,0)     17.1 %
20      continue
        s = a + b                          0.2 %
```

```
[illegible]    2      3      4
[illegible]    34     35     36
[illegible]    66     67     68
[illegible]    98     99     100
[illegible]    130    131    132
[illegible]    162    163    164
[illegible]    194    195    196
[illegible]    226    227    228
[illegible]    258    259    260
[illegible]    290    291    292
[illegible]    322    323    324
[illegible]    354    355    356
[illegible]    386    387    388
[illegible]                  14,4
```

Figure 1 : Screen dump with several Prism windows

3. User Interface

Prism's user interface runs under the X Window System and is based on the Motif widget set. The interface was designed to be easy to learn, and once learned, fast and efficient to use. These are the two key criteria that define the ease-of-use of any system.

Pull-down menus, buttons, and dialog boxes make most of the functionality of Prism easily and obviously available to the user. The appearance of dialog boxes is standardized as much as possible to increase user familiarity with the interface. For example, most dialogs have apply, close, and help buttons. Standard keyboard accelerators can activate the close or help buttons in any dialog at the touch of a key.

A comprehensive online-help system provides nicely-formatted documentation on all aspects of using Prism. In addition, context-sensitive help is available in all dialogs and pull-down menus, via a help button in the former and a help menu option in the latter. Finally, integration of Thinking Machines' WAIS text retrieval software into Prism provides users with a powerful capability for online searching of the entire Connection Machine documentation set, using relevance-feedback techniques.

Prism provides a number of shortcuts that increase the speed of use of the interface. Pull-down menu items may be copied to the tear-off region, where they become buttons that perform the same action. For some users, a command line interface may be used more rapidly than a graphical one, so commands can be typed in a text region that maintains a history of commands and Prism output. A number of actions that apply to program entities, such as printing a variable or listing a function, can be rapidly performed by interacting with the source region, which displays the source code for the current function or file. The user selects some text in the region and then chooses a popup-menu option to apply to the selected entity. These same options are accessible through the main menu with more guidance given to the user, but at a cost of additional mouse gestures and keystrokes.

Lastly, Prism allows interesting and useful graphical interactions with some of the underlying debugger functionality. The Where, File, and Function dialogs display lists of stack frames (i.e., function invocations), available source files, and available functions, respectively. If the user clicks on any list item, the source region displays the source code for that item, and if the item is in the Where or Function list, the default scope for variable lookups is set to the chosen function. Prism Events (see below) may be created and modified using the Event Table dialog, which allows complete generality in specifying event descriptions.

4. Debugging of data-parallel programs

The data-parallel languages for the Connection Machine include CM FORTRAN and C*. These are extensions of the FORTRAN and C languages which provide for the manipulation of arrays in parallel. Since these languages still have a single control flow, it is relatively straightforward to extend existing debugging paradigms (such as what dbx provides) to programs written in these languages. Therefore, we chose dbx as the base for Prism's debugging features. The rest of this section describes the extensions we made to turn dbx into a debugger for Connection Machine programs.

First of all, Prism needed to be taught to fetch parallel data from the Connection Machine. This mechanism is based on calling runtime routines via the "call" command mechanism. In this way, Prism can get at parallel CM FORTRAN arrays or C* parallel variables.

Secondly, dbx's expression parser was extended to handle CM FORTRAN (FORTRAN 90) expressions and C* expressions. The results of these expressions can be printed out in the command window or fed to one of Prism's visualizers (see section 6). This ability of the debugger to interpret expressions in our languages has proved valuable.

Thought had to be given to how best to provide graphical interfaces to standard debugger functionality. The source window (see figure 1) is the backbone of Prism: It provides visibility into the current source position, and provides the ability to set breakpoints and print variables. In additional, optional pop-up windows provide the functionality of commands such as "func", "file", and "where" (see figure 2). These windows interact in the natural ways. For example, clicking on a function in the "Func" window will re-position the source window to the start of that function.

Another area where dbx's capabilities were extended is in Prism's "event table". The idea is to generalize and enhance the dbx "stop...", "trace...", "when ...", and "display ..." capabilities. A Prism "event" consists of a trigger condition, and actions to be performed. Examples of trigger conditions: whenever the program reaches a certain point (standard breakpoints), whenever a condition becomes true ("stop if ..."), or whenever the value of a variable changes ("stop <var> ..."). Actions can be arbitrary Prism commands, such as "print". For example, by setting up an event that triggers on every line, whose action is to update a visualizer for the array "x", it is possible to watch the array "x" change as the program runs. Figure 3 shows the event table.

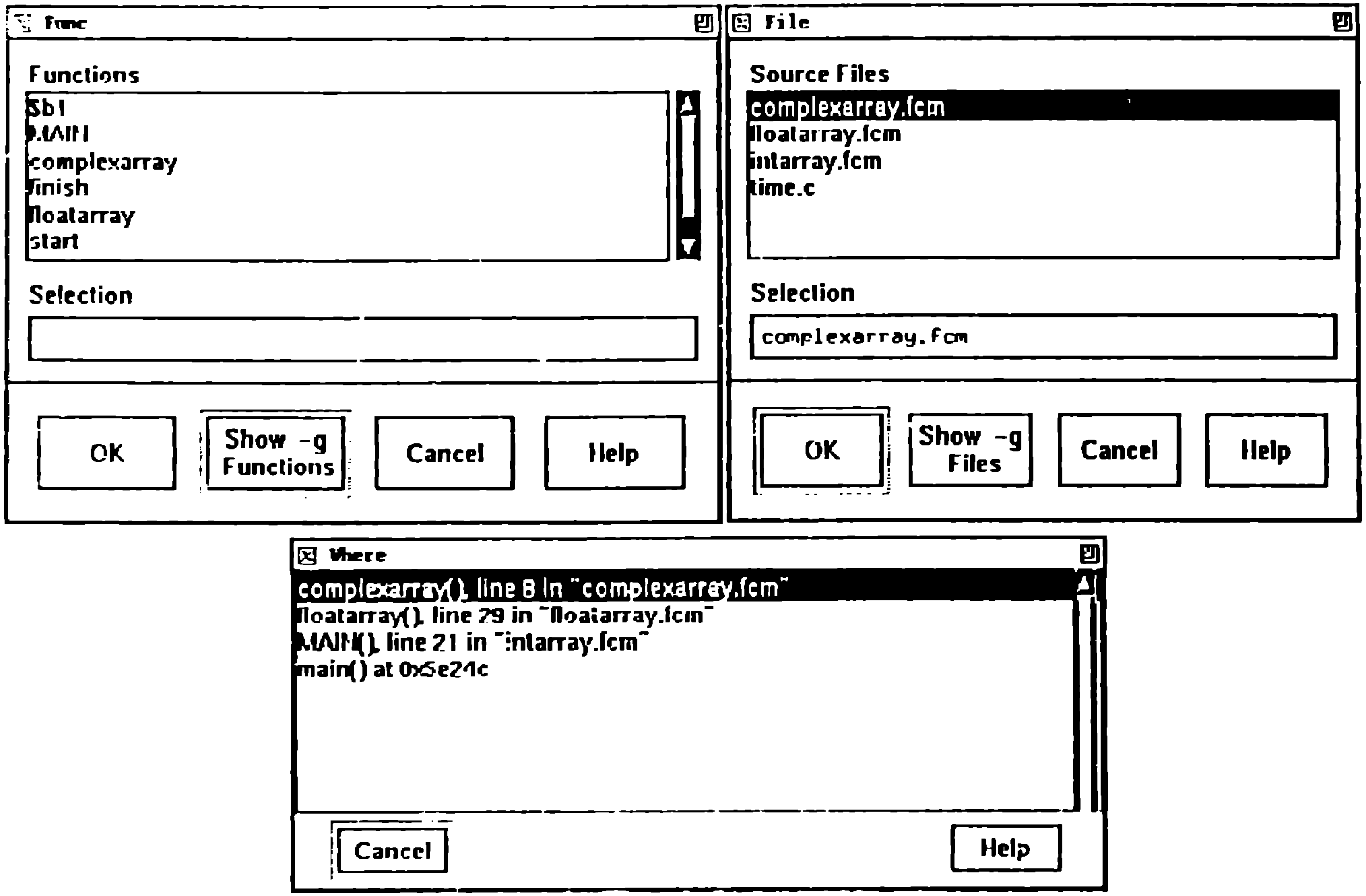
5. MIMD-Parallel debugging

With the advent of the CM5, we can now support another programming model on the Connection Machine: Multiple threads of control with explicity message passing between them. This programming model is supported by the CM5 operating system and its CMMD message-passing library.

Prism currently supports this programming model by offering an interface to the new Pndbx debugger. Prism can pop up a window running Pndbx. Prism remains active and can be used on the front-end portion of the user program, while the Pndbx debugger provides visibility and control of what is going on in the Processor Nodes.

A full description of Pndbx is outside the scope of this paper, but briefly, the idea is to provide dbx-like debugging into any of processor nodes. The user can switch his debugging context amongst the nodes. Capabilities are also provided for iterating any command across all nodes or sets of nodes (e.g., "where all"). Pndbx provides these capabilities with a dbx-like command interface.

For the future, we are looking into the following: (1) Providing a more graphical interface to the Pndbx functionality, (2) Closer integration of Pndbx and Prism, and (3) Investigating alternate paradigms for MIMD parallel debugging. This entire area is a fruitful one for further research.



Functions:

```
$bl
.MAIN
complexarray
finish
floatarray
start
```

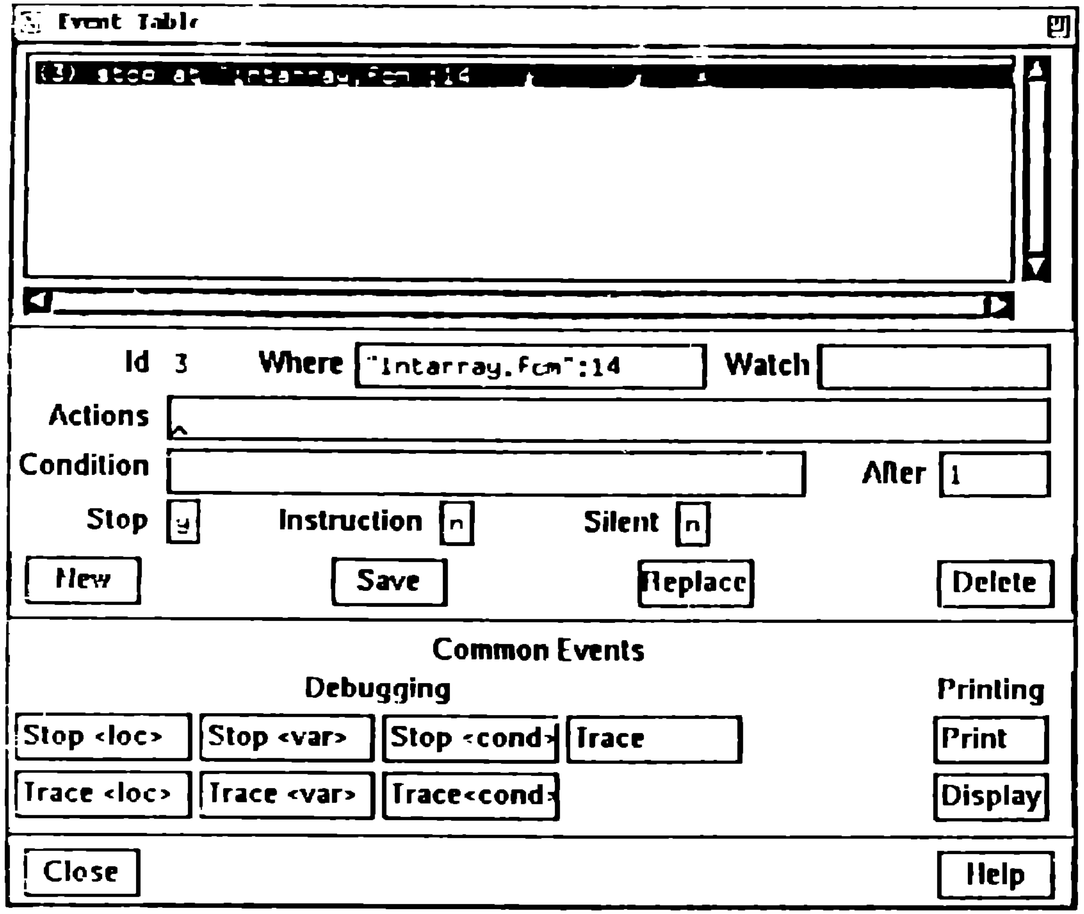
Source Files:

```
complexarray.fcm
floatarray.fcm
intarray.fcm
time.c
```

Selection: complexarray.fcm

```
complexarray(), line 8 in "complexarray.fcm"
floatarray(), line 29 in "floatarray.fcm"
MAIN(), line 21 in "intarray.fcm"
main() at 0x5e24c
```

Figure 3 : Graphical interfaces to "func", "file", "where"



(3) stop at "intarray.fcm":14

Id 3 Where "intarray.fcm":14 Watch

Actions

Condition After 1

Stop y Instruction n Silent n

Figure 3: The event table

6. Data Visualization

Prism currently provides a tightly-integrated capability for the visualization of parallel and serial data. A separate window, called a visualizer, may be created for each variable or expression to be visualized. Visualizers give the user an efficient means to navigate through and interpret the large amounts of data tnat are typically found in massively parallel programs.

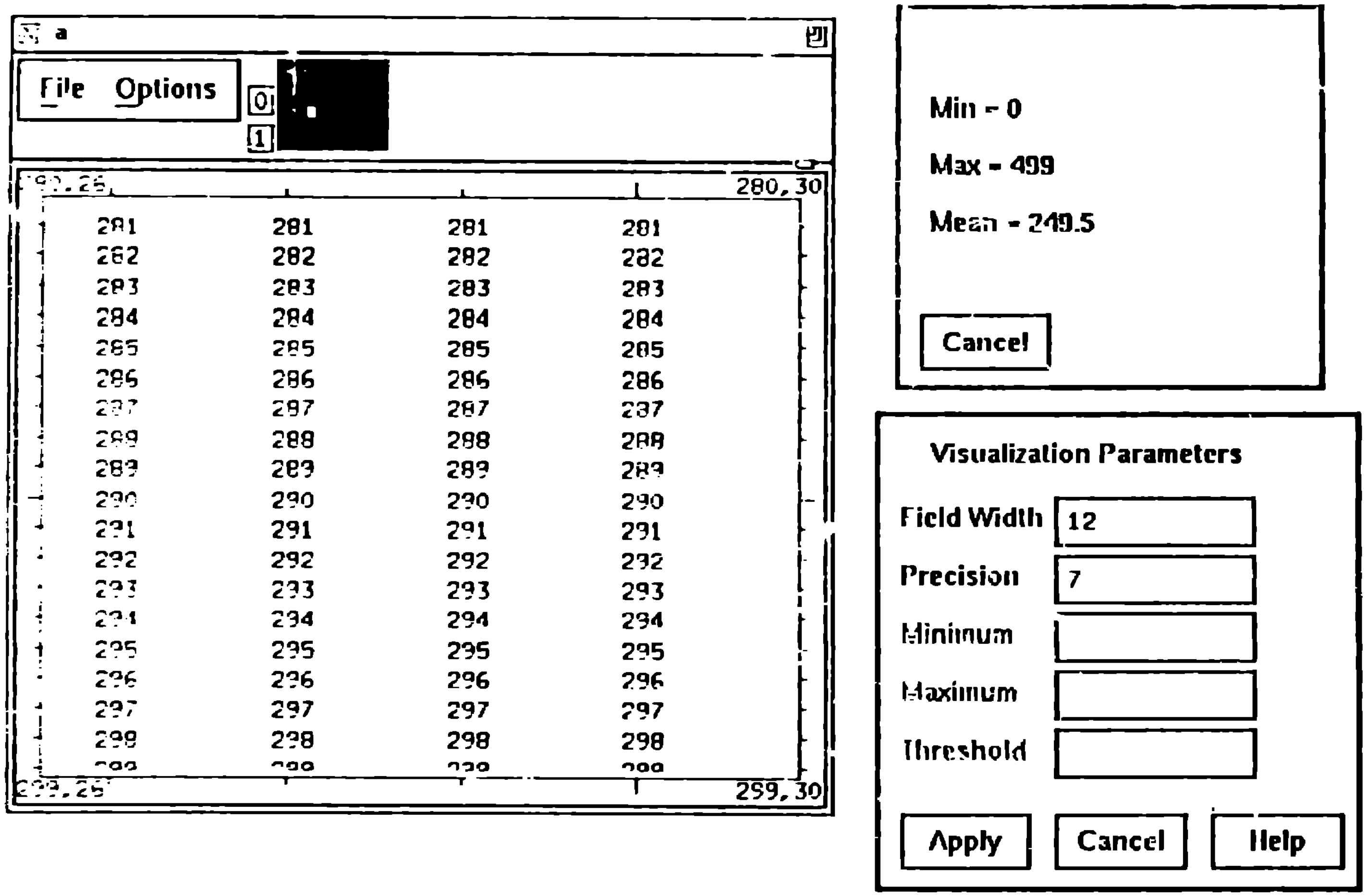
Visualizers are designed to view multi-dimensional arrays of data. Any variable or expression of type array may be viewed, whether the data resides on the CM, the front end, or some combination thereof. The visualizer displays one section of a two-dimensional slice of data, and the user may pan around in this slice using simple graphical gestures. A ruler may be enabled which shows the coordinates of the array elements at the four corners of the visualizer window. For higher-dimensional data, sliders are provided for varying the coordinates of axes that are orthogonal to the displayed slice. Assignments of array axes to the sliders and the two window dimensions are shown by small text fields which can be edited to change the displayed layout.

Data may be displayed in visualizers using one of several pre-defined graphical representations, which may be freely changed after a visualizer is created. Textual, colored pixel, and boolean pixel representations are currently available. For a textual visualizer, the ASCII representation of each array element is printed in the window. Though this may seem little different from the dbx print command, the data navigation capabilities offered by Prism make visualization using text vastly superior to dumping a potentially huge array to a terminal.

The pixel representations display one array element per pixel. A boolean pixel visualizer maps elements to black or white based on comparison to a threshold value, while a colored pixel visualizer takes the range of data and discretizes it to fit either a default spectral colormap or a colormap provided by the user. Because the pixel representations give an inexact indication of the value of each array element, Prism provides a point-and-click operation to query the value of any pixel.

Figure 4 shows a textual visualizer into a 2-dimensional array "a".

The builtin visualizers provided by Prism give the user fast and effective means to view data. In the future, Prism will be also able to export data to external visualization systems such as AVS.



```
280,26                                          280,30
   281        281        281        281
   282        282        282        282
   283        283        283        283
   284        284        284        284
   285        285        285        285
   286        286        286        286
   287        287        287        287
   288        288        288        288
   289        289        289        289
   290        290        290        290
   291        291        291        291
   292        292        292        292
   293        293        293        293
   294        294        294        294
   295        295        295        295
   296        296        296        296
   297        297        297        297
   298        298        298        298
299,26                                          299,30
```

Min = 0

Max = 499

Mean = 249.5

Visualization Parameters

Field Width: 12

Precision: 7

Minimum:

Maximum:

Threshold:

Figure 4 : A visualizer

7. Performance Analysis

While data-parallel programs share important characteristics with their sequential counterparts, i.e., sequential or pseudo-parallel control flow, their ambitious performance goals, combined with the added complexity of inter-processor communication and large data sets, make comprehensive performance tools an absolute necessity.

In Prism, we have begun to implement such a suite of performance analysis tools. Our initial work focuses on providing Connection Machine users with an understanding of where and how their programs are spending their time. Per-procedure and per-source-line graphical displays are available for the utilization of the major resources (e.g., processor nodes, inter-processor communications network, mass-storage devices) of Connection Machine systems. Furthermore, these displays are available for each node in a program's dynamic call-graph. The displays are designed so that the user can easily identify the most heavily utilized resource, and, by navigating about the call-graph by simply pointing and clicking, home in on the places in the source code that are the major contributors to that utilization.

Figure 5 shows some of Prism's performance histograms.

We feel that providing integrated profiling of a program's use of all the resources available to it represents an important advance over profiling tools that provide information about processor utilization only. It is important that performance analysis tools permit the user to easily understand where the leverage is; this is not possible without an understanding of a program's usage of all parts of the computing system that can contribute to delay.

8. Summary

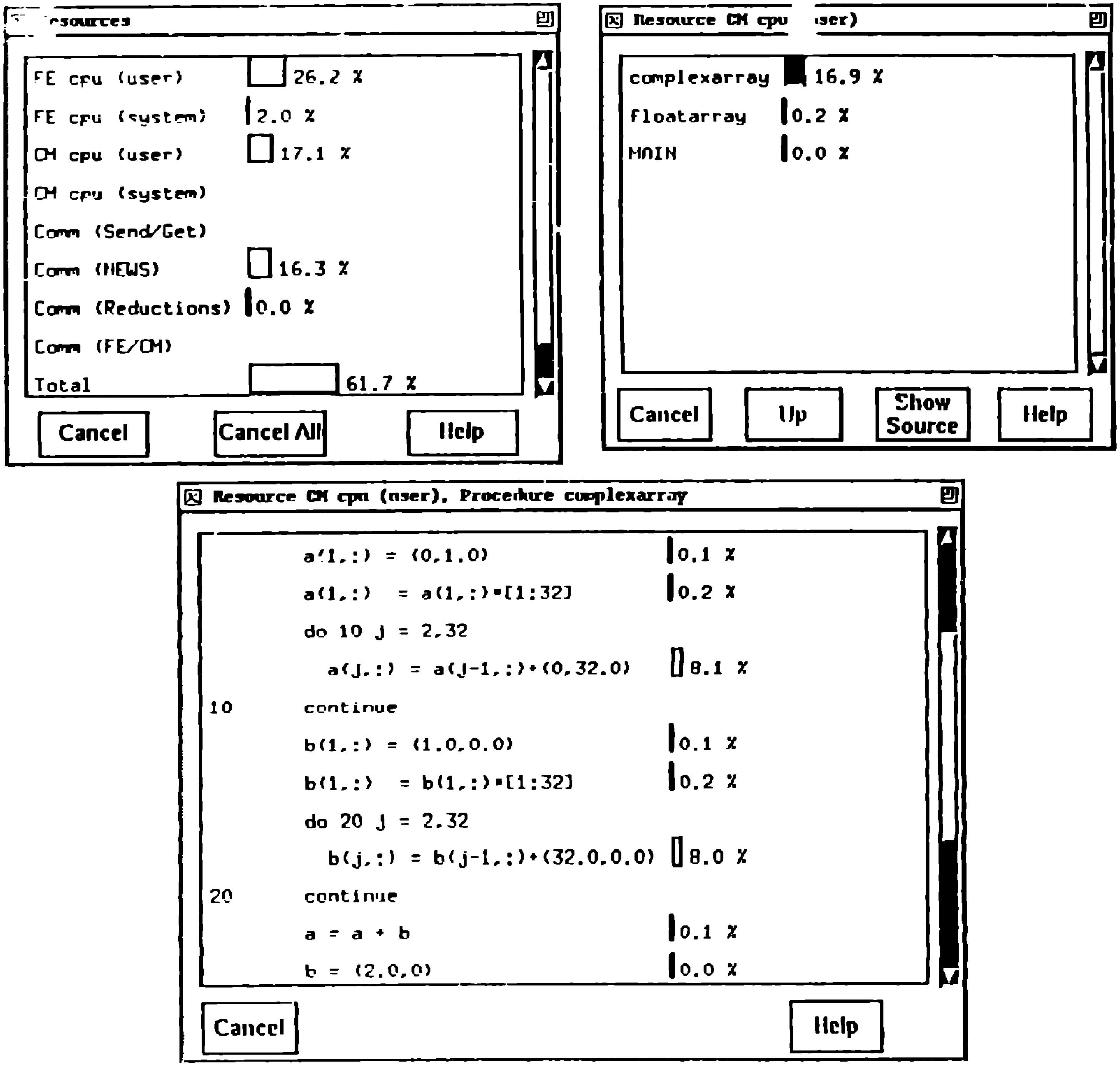
This section summarizes how we achieved the various goals of the project:

a. The OSF/Motif and X-windows based interface provides ease of use and an attractive appearance.

b. By basing Prism on existing software such as dbx we were able to get it up and running quickly.

c. By isolating target-dependent things in runtime routines (e.g., fetching actual data from the Connection Machine), we were able to easily retarget Prism. In fact, the same Prism executable works for both CM2's and CM5's.

d. By tackling problems that are difficult on supercomputers (debugging, performance analysis) we were able to provide a truly useful tool.



esources

```
FE cpu (user)        26.2 %
FE cpu (system)      2.0 %
CM cpu (user)        17.1 %
CM cpu (system)
Comm (Send/Get)
Comm (NEWS)          16.3 %
Comm (Reductions)    0.0 %
Comm (FE/CM)
Total                61.7 %
```

Resource CM cpu user)

```
complexarray   16.9 %
floatarray     0.2 %
MAIN           0.0 %
```

Resource CM cpu (user), Procedure complexarray

```
        a(1,:) = (0,1,0)                       0.1 %
        a(1,:)  = a(1,:)*[1:32]                0.2 %
        do 10 j = 2,32
          a(j,:) = a(j-1,:)+(0,32,0)            8.1 %
10      continue
        b(1,:) = (1.0,0.0)                     0.1 %
        b(1,:)  = b(1,:)*[1:32]                0.2 %
        do 20 j = 2,32
          b(j,:) = b(j-1,:)+(32.0,0.0)          8.0 %
20      continue
        a = a + b                              0.1 %
        b = (2.0,0)                            0.0 %
```

Figure 5 : Some performance histograms

CXdb
A New View On Optimization

Larry V. Streepy, Jr.
Co-developers:
Gary Brooks, Russell Buyse, Mark Chiarelli,
Mike Garzione, Gil Hansen, Dave Lingle,
Steve Simmons, Jeff Woods

Convex Computer Corporation
3000 Waterview Parkway
Richardson, TX 75083
streepy@convex.com

November 12, 1991

Abstract

The state-of-the-art in optimizing compiler technology has increased rapidly over the years — a pace that debugger technology has not been able to match. Convex's newest offering in debugger technology, CXdb, synchronizes the two technologies. CXdb is a full-featured debugger that provides the developer capabilities to debug optimized code. It provides a sophisticated user interface for effective communication of debugging information and a rich command set enabling the user to easily work with the application being debugged. CXdb's understanding of compiler optimizations are based on an innovative set of information emitted by the compiler, the Compiler-Debugger Interface (CDI). CXdb also derives compiler-synthesized variable values at run-time.

1.0 Introduction

This paper describes:

- The motivation behind CXdb's development.
- The data contained within the CDI and how it is used to understand the program being debugged.
- How the Graphical User Interface enhances the users understanding of the program state.
- Features of the command language specifically aimed at handling optimized code.

This paper closes with comments on possible future directions for CXdb functionality.

1.1 Motivation

The state-of-the-art in optimizing compilers has been steadily advancing over the past several years. Current compilation technology can provide automatic scalar, vector, and parallel optimizations on an application [Conv90] [Conv91a] [Lu91] and [Sark90]. However, the corresponding art of debugging technology has not been keeping pace. Typical current-day debuggers require that the application be compiled with optimizations disabled.

Being able to debug with optimizations enabled has several distinct advantages. The following list presents some of these advantages.

1. Applications typically run several times faster when optimized. The longer it takes the program to exhibit a bug, the longer the developer will have to wait until the actual task of debugging can begin. This lengthens the overall time-to-solution. When debugging optimized code, the length of each edit-

compile-debug cycle is reduced in direct relationship with the speedups provided by those optimizations.

2. Certain classes of bugs may exhibit themselves only when optimizations are applied. Therefore, having to debug only the non-optimized code may prevent you from ever finding the bug. For example, arithmetic optimizations may alter the convergence behavior of an algorithm.
3. Compiler developers are also debugger users. For the compiler developer to be able to debug compiler components that affect optimizations, the debugger has to be able to handle the object code in its optimized form. Without this support the developer is forced into the tediousness of working at the instruction level.
4. Performance debugging. Bad performance can be considered a "bug" just like a logic error. It is essential to debug the code *as optimized* to determine what additional optimizations to make.
5. Support for production codes. Developers will be able to work with customers in the field using the optimized application. Core files submitted by customers can be operated on directly.

The result of these restrictions and problems is that the developer is placed under a considerable handicap when trying to develop, debug, and tune optimized applications. CXdb was developed with the solution of these problems as one of its major design goals. The remaining sections of this paper provide a tour of the various features of CXdb and describe how they help developers work with optimized code.

1.2 Existing Research

Most of the current research in debugger support of optimized code has been focused on hiding the effects of the optimizations from the developer, [Henn82], [WaSr85] [Zell83], [Zell84], [CoMc88], and [Zura90]. The objective of their research is to present the user with *expected behavior*. Although this approach works with specific classes of optimizations, it becomes intractable when you consider that optimizations are cascaded, merged, and applied multiple times in various orders.

It isn't clear that all optimizations can be made transparent. To maintain the transparency of the optimizations each optimization must be reevaluated each time it is revised or a new optimization is added. This leads to excessive maintenance in the debugger to keep it in synch with the compiler. Such maintenance costs are unacceptable in today's extremely competitive market.

CXdb was developed to address compiler optimizations by depicting what is *actually* happening. CXdb uses visual feedback to present the effects of the optimizations on a program's behavior.

1.3 Types of Optimization

Many optimizations can be applied during the compilation process. These optimizations can be divided into three major categories: scalar, vector, and parallel [Conv90] [Conv91a]. The technology of automatic scalar and vector optimization is well advanced. The science of automatic parallel optimization is comparatively new. Table 1 presents a sample of the optimizations that fall within these categories.

Table 1. Optimizations by Category

Category	Optimization
scalar	instruction scheduling span-dependent instructions global register allocation tree-height reduction redundant-assignment elimination assignment substitution common-subexpression elimination redundant-use elimination constant propagation and folding algebraic and trigonometric simplification dead-code elimination hoisting and sinking scalar and array references copy propagation code motion strength reduction
vector	strip mining loop distribution loop interchange paired hoist and sink conditional induction variables
parallel	loop distribution Parallel strip-mining variable vector strip-mining scalar spreading and reduction (plus combinations with vector optimizations)

2.0 The Compiler-Debugger Interface

One of the limiting factors in current UNIX debuggers, such as gdb, sdb, and dbx, is the mechanism that transmits compile-time information to the debugger. All of these debuggers use a STAB[1] based approach to retrieve compiler-produced debugging information (caveat: The sdb debugger is based on a set of debugging data, known as DWARF, that is currently undergoing a standardization effort). The limitations of STAB- or DWARF-based implementations in supporting optimized code debugging are summarized as follows:

- Syntactic granularity. The view of the source code provided is restricted to basic blocks and statements. This granularity is too coarse to handle optimizations which often operate at the expression level (which encompasses most scalar optimizations). Additionally, the view is purely textual; not syntactic.
- Variable value location. Under optimization the value of a variable may migrate between several machine locations (memory, register, nowhere). There is no method of encoding this information within the STAB mechanisms.
- Mapping source code and object locations. The STAB mechanism only supports a 1:1 mapping from source lines to object code. Optimizations can replicate or fuse object code segments in ways that require a many-to-many (M:N) mapping of source code to object code.

With these limitations and the requirements of handling optimized code in mind, an entirely new mechanism was developed. This new compiler-debugger interface (CDI) is represented by a set of compiler-created data files.

The Convex compilers are composed of language-specific front-ends and a common back-end. The front-ends perform the lexical and semantic analysis of the compilation process. The back-end implements optimization and code generation. Each module (front-end or back-end) is responsible for generating a portion of the CDI. The front-end generated data files correspond to each primary source file. The back-end generated data files correspond to each object file produced. The components of the CDI are presented in the following sections. Table 2 presents a brief overview of the data files and their contents.

Table 2. CDI Data File Overview

Component	Data File	Compiler Generation
Source File Map	executable	back-end
Section Table	executable	back-end
Name Space	.ns	front-end
Type/Scope Information	.tsi	front-end
Source Unit Table	.sut	front-end
Source Range Table	.srt	back-end
Variable Table	.vt	back-end
Location Range Table	.lrt	back-end
Expression Table	.xpt	back-end

2.1 Front-end Components

2.1.1 Namespace

Each namespace data file (NS) contains the top-level symbols defined within the source file. Usually these symbols are visible across the entire program. In C, top-level symbols consist of external identifiers. In Fortran, they consist of subroutine, function, and common-block names.

The NS contains the partial language namespace contributed by the source file. The union of all the partial namespaces comprises the language namespace for the application being debugged. In programs written in multiple languages, the debugger maintains one language namespace for each source language. The namespaces provide a mapping from symbol names to source files. This mapping provides support for dynamic loading of debugger data files.

2.1.2 Source Unit Table

The source unit table (SUT) contains the encoding for the source unit trees resulting from the input source file. A *source unit* is an abstracted, language independent piece of the program. A file's source units form a tree that reflects the syntax of the program. Source units are classified into groups depending on the type of linguistic construct from which they are derived. The currently supported source

[1] STAB (Symbol TABle) information includes name, type, and location for variables and address ranges for source statements

unit types, also called granularities, are listed in Table 3.

Table 3. Source Unit Granularities

Granularity	Description
Expression	any valid combination of constants, operators, and operands in the current source language
Statement	any valid statement in the current source language
Block	statements that constitute the body of a routine, loop, or conditional construct
Loop	an iterative construct (examples: for, while, DO)
Routine	a main routine, subroutine, or function

Each SUT entry contains the following information:

index	Unique integer index within .sut file
kind	Granularity of source unit, see Table 3
position	Start and end source positions
scope node	Reference to scope node, see description of scope nodes below

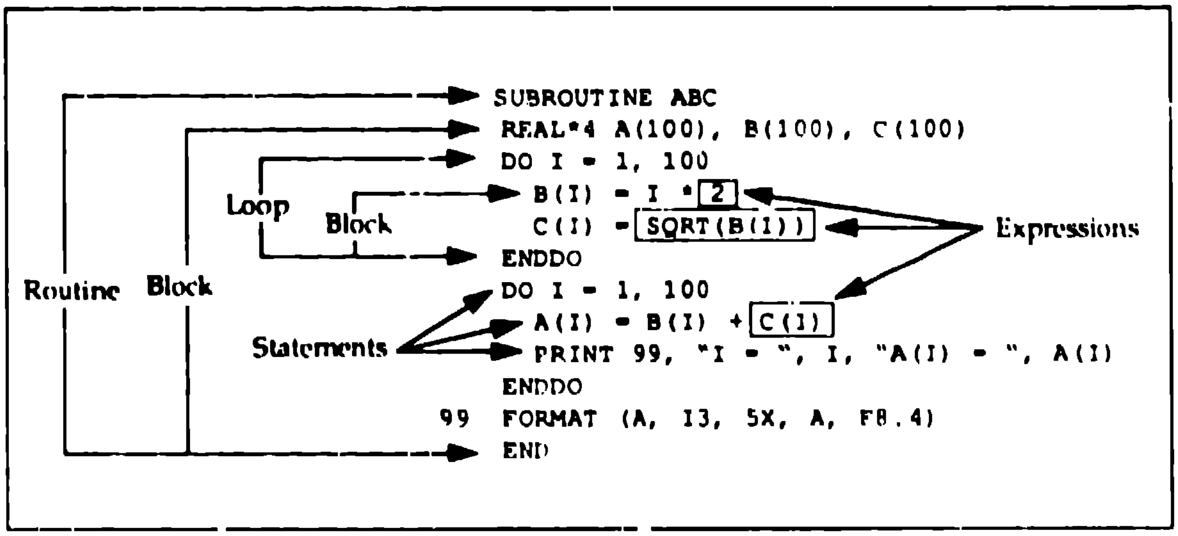
The source units are formed into trees based on the lexical nesting of their source positions. The source unit tree is used to control the highlighting of program text by the user interface. Figure 1 shows examples of the different types of source units in a Fortran routine.

3 Type and Scope Information

The type/scope information (TSI) data file contains the encoding for the type descriptors, scope nodes, scope entries, and scope blocks for the compilation unit. Each front end produces the following information:

- A *scope entry* for each user-defined symbol. Scope entries are used to model the different symbol types that can appear in the program's lexical environment. Currently, there are five scope entry types: *variable entries* (including routines), *type entries* (C typedef's), *type tag entries* (for structs and unions), *common block entries* (Fortran common blocks), and *enumeration entries* (enumeration literals).
- A *scope node* tree that reflects the scope of the symbols in the source code. The scope nodes form the backbone of the lexical scoping environment. Scope nodes provide different levels of lexical visibility. Scope entries on the same scope node have the same visibility.
- A *scope block* tree that reflects the block structure of the source code. Scope blocks are used to specify arbitrary program symbols from any point within a program. For example, to reference statics or common block variables. To reference a program symbol in another branch of the scope environment, the debugger

Figure 1. Example of source units in Fortran

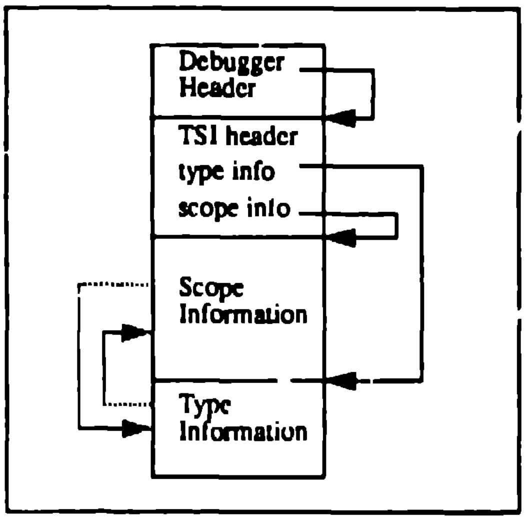


```
     SUBROUTINE ABC
      REAL*4 A(100), B(100), C(100)
      DO I = 1, 100
        B(I) = I * 2
        C(I) = SQRT(B(I))
      ENDDO
      DO I = 1, 100
        A(I) = B(I) + C(I)
        PRINT 99, "I = ", I, "A(I) = ", A(I)
      ENDDO
  99  FORMAT (A, I3, 5X, A, F8.4)
      END
```

uses scope blocks to navigate through the scope environment. Scope blocks provide a means for referring to scope levels (nodes) by name. Scope blocks are linked downward to all immediately nested scope blocks. A path through the scope environment is specified as a sequence of scope block names from an outermost scope block down to the desired scope block.

The layout of the TSI file is more complicated than other data files and deserves a little more attention. Figure 2 presents the general structure of the TSI file.

Figure 2. TSI file layout



The *debugger header* is a standard header, which all debugger data files contain. It contains size and version information verifies the data file was created by a version of the compiler compatible with the executing version of CXdb. The *TSI header* contains information describing the number of entries in the sections that follow, as well as pointers (offsets) to the start of the other sections in the file.

The *scope information* consists of a series of scope entries, scope nodes, and scope blocks as described above. The *type information* consists of a series of type descriptors that are indexed by scope entries. All user-defined data types are defined in this data file.

2.2 Back-end Components

The compiler back-end creates data files to represent the layout of object code corresponding to input source code. These data files include information on:

- Variable attributes (user and compiler-synthesized)
- Object-code to source-code mappings
- Memory layout of sections within the object file
- Ephemeral variable locations
- Synthesized variable expressions

Each back-end generated data file is described below.

2.2.1 Expression Table

The expression table (XPT) encodes information for handling compiler-synthesized variables. The compiler generates synthesized variables as either a replacement for a user-defined variable (called a *derived* synthesized variable) or a mechanism for runtime support (called a *runtime* synthesized variable). The variable is synthesized so that either optimizations or code generation may proceed with less difficulty.

For a derived synthesized variable, the *uses* of the old variable are replaced with more efficient uses of a derived variable. Sometimes, all uses of the old variable are replaced and the old variable does not physically exist. The new variable value is expressed as a linear function of the variables from which it was derived. For example, the user may have the following code:

```
while( X[i++] != 0 ) ...
```

Rather than incrementing i, multiplying it times the size of an element of x, and then adding it to the base address of x, the compiler optimizes this code by replacing i with a variable that is compile-time computable and increments the variable (call it ?ifoo) by sizeof(x[0]) on each iteration. The resultant variable is derived from the following linear equation based on the original variable i.

```
?ifoo = &X + (i * sizeof(x[0]))
```

By maintaining the linear equations that define a synthesized variable, the debugger can solve the equation for i to derive the current value of the variable that has been replaced. In the example above, the value of `i` can be expressed as:

```
i = (?ifoo - &x) / sizeof(x[0])
```

Runtime synthesized variables are created to hold information needed at runtime. They may be created for many reasons. Some examples are: tracking vector lengths, creating vector masks, tracking vector spills, retaining argument pointers, etc. This type of synthesized variable has a direct value and, thus, no corresponding linear equation set.

The XPT contains the following information for each synthesized variable.

Purpose An indication of the variable's use (for example, vector spill area, vector length, etc.)

Identifier Variable name

Type Enumeration value indicating the specific synthesized variable type

VT Index Reference to the variable table entry for the synthesized variable

Expression tree Linear equation encoding for variable derivation (not included for runtime support variables)

2.2.2 Location Range Table

The location range table (LRT) encodes information to track the home location of a variable and the run-time variable-to-location bindings. Possible machine locations are:

- Registers
- Stack frame relative (for local variables)
- Register relative (for arguments)
- Segment relative (for statics and globals)

Variables can have both *home location* entries (the location where the variable resides at throughout program execution) and *ephemeral location* entries (describing the various run-time locations).

Each entry in the LRT contains the following information:

VT Index Reference to the variable table entry

Addr range Start and end instruction addresses in the executable image over which this entry holds true

Location Machine location encoding

Due to optimizations a variable may have multiple overlapping entries. Given an execution address (PC) and a variable, the LRT indicates which location(s), if any, are associated with that variable, as follows:

$$(\mathtt{PC},\ \mathtt{Var}) \Rightarrow \{\mathtt{Loc}_1, \ldots, \mathtt{Loc}_n\}$$

Given a PC and a machine location, the table can also be used to determine which variable currently resides at that location. The following equation is used when examining the registers or stack frames to determine associated variables:

$$(\mathtt{PC},\ \mathtt{Loc}) \Rightarrow \{\mathtt{Var}\}$$

If no location is bound to a variable, the variable is said to be ***unavailable***. Unavailable variables can occur, for example, due to constant propagation and redundant assignment elimination.

2.2.3 Section Table

The section table is not encoded within a special data file, but is encoded within the symbol table of the created executable. Special STAB entries encode the following information:

Type The section type (TEXT, DATA, etc.)

Address The section's base address

Object file The object file contributing to this section

Each object file may contribute a portion of the text, data, bss, tdata, and tbss sections of the final executable. The table maps virtual addresses in an executable to relative addresses within a section of an object module. This map, along with the NS, TSI, and SFM, control the on-demand loading of debugger data files.

2.2.4 Source File Map

Like the section table, the source file map (SFM) is encoded within the executable. However, it is encoded directly within the text section of the object code produced. It is simply a list of the source file names that produced the object file.

2.2.5 Source Range Table

The source range table (SRT) encodes information describing the object code ranges for each source unit. A source unit, as described in Section 2.1.2, is a lexical component of the input source code. Without optimization, a single source unit will map to a single range of object code (that is, a single statement may generate a sequence of five machine instructions). Optimizations destroy this 1:N mapping. Optimizations merge, split, remove, and replicate the object code associated with the input source units.

The SRT is a two-way, N:M map from source units to object code (PC) ranges. Each entry in the SRT contains the following information:

SU index	Source unit index
Range	Range of object addresses occupied by this source unit. The addresses are relative to the beginning of the object file
SFM index	Source file map index

Optimizations (for example, instruction scheduling) may create multiple entries for a source unit. Multiple source units may contain the same address because source units are nested constructs (that is, a routine may contain loops, which contain blocks, which contain statements, which contain expressions).

The PC-to-SourceUnit mapping, following, determines the active source units at a given program location. It also determines the current scope because source units contain an index to a scope node. The user interface uses the active source unit tree to determine which source code segments to highlight.

$$(PC) \Longrightarrow (SU_1, \ldots, SU_n)$$

The SourceUnit-to-range mapping, following, determines the starting addresses of a specific source unit. Without optimizations, this mapping is typically 1:1 (that is, each source unit maps to a single range of addresses). However, when optimizations are applied, this mapping becomes 1:N. The debugger uses the starting addresses, or *entry points*, for a source unit to determine where to place breakpoint instructions for managing stepping[2], breakpoints, tracepoints, and eventpoints.

$$(SU) \Longrightarrow (Range_1, \ldots, Range_n)$$

2.2.6 Variable Table

The variable table (VT) encodes information about all user-defined and compiler-synthesized variables in an object module. Variables are divided into two classes that indicate the extent of the variable. Those variables whose lifetimes extend across the entire program have *indefinite extent* and those whose life is restricted to a particular scope have *definite extent*.

Examples of infinite extent variables are static and external variables in C, package-local variables in Ada, and common block variables in Fortran. Variables of this type are allocated a *home location* in global memory. During program execution these variables may be allocated to other locations (for example, registers). The location range table determines the run-time locations for a given variable. See the discussion of the LRT, Section 2.2.2, for more details.

Finite extent variables, which includes all *automatic* variables in C, have no home location but may still migrate between machine locations within the extent of their lifetime as described within the LRT.

Each VT entry contains the following information:

Type	Synthesized or user-defined
Storage class	Storage class of variable. This is a much broader definition than is used in conjunction with C. Some examples are auto, static, dynamic, register, argument, section based, etc.
Reference	Reference to related variable (for common blocks, equivalences, and based variables)
Flags	Flags that define additional attributes of the variable such as row-wise array, Cray pointer, dummy argument, compiler temporary, Fortran format specifier, etc.
Scope entry	Index to the scope entry defining this variable. The scope entry contains the type descriptor for the variable

[2] CXdb uses special breakpoints, called *transient* breakpoints, to control source level stepping. The source unit address ranges are used to determine where to place these transients. See Section 4.2 for more details

3.0 User Interface Components

CXdb uses CX/Motif (Convex's version of the OSF/Motif toolkit) [OSF91] for its graphical user interface (GUI), and the Maryland Windows Library [Tor83] for its full-screen CRT user interface [BuCh91].

Although the CRT interface does not support a direct pointer device (mouse), pull-down menus, or pop-up windows, it does provide a powerful multi-window environment that maximizes the use of a CRT's limited screen real estate. The CRT interface supports the highlighting necessary to provide program animation as described for the GUI source window in Section 3.2. With the limitations noted, the CRT interface is very similar to the GUI presented below. It will not be discussed further.

The GUI is composed of several specialized windows used to interact with the debugger and the target[3] process. Each window is introduced briefly in the list below and described more fully in subsequent sections.

Command Window All command entry (via the keyboard) is through this window. It contains pull-down menus for access to the full CXdb command set (*command composition*), and *expert buttons* for access to commonly used commands (via the mouse).

Source Window All program source code is viewed from this window. Source units are highlighted to indicate program execution. Source based eventpoints[4] are indicated by special icons.

Disassembly Window Presents an annotated listing of the disassembled object code of the executable. Addresses are annotated with the associated variable name (if known). Pop-up windows provide access to the machine register state. Eventpoints are also indicated with icons in this window.

Examine Window Presents the contents of process memory in a wide range of user-selectable formats ranging from binary to Fortran complex.

Stack Window Presents a backtrace of the current stack contents. Point-and-click operations present detailed information on selected stack frames.

Process IO Window All I/O operations on the process being debugged are isolated within this window. It provides full CRT emulation.[5]

Help Window Provides access to the complete online Reference Guide [Conv91b]. Sophisticated search capabilities provide easy navigation between reference pages.

Each of the source, disassembly, examine, and stack windows are created on a per-thread basis. This provides the user with direct access to individual thread state in a parallel application. Multi-threaded application support in an integral component of CXdb's design. See Section 4.3 and 4.4 for additional information on multiple thread support.

The combination of these specialized windows present a synergistic interface providing rapid access to program information and effective control of the target process. Each of the major GUI components are described below.

3.1 Command Window

The command window is the major focal point for controlling CXdb's operation. It provides a series of pull-down menus and buttons providing mouse-based access to all CXdb commands. The command window enables the user to:

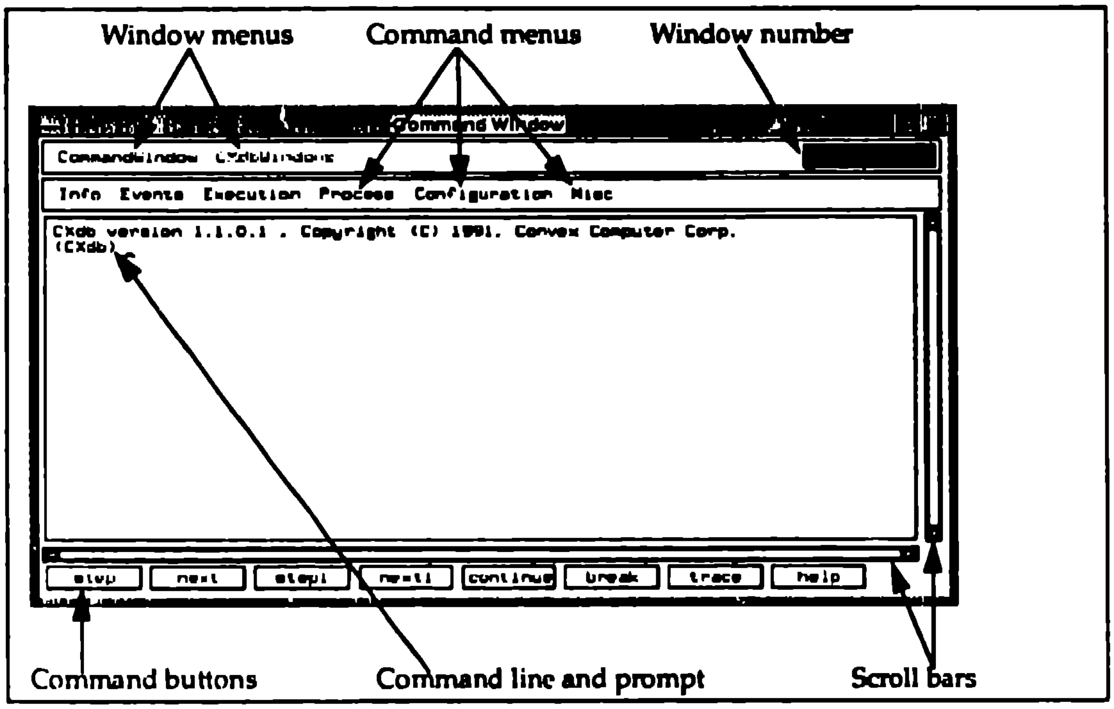
- Enter CXdb commands
- Receive output from CXdb commands
- Receive error messages or status information about CXdb commands
- Review previous commands and retrieve them from the command history
- Control the creation and display of the other windows within the GUI

[3] The term *target process* will be used interchangeably with *process being debugged*.

[4] Eventpoint is a generic term for CXdb's execution control mechanisms including breakpoints, tracepoints, watchpoints, etc.

[5] The CRT emulation is accomplished by using xterm, an application supplied with the X Window System

Figure 3. Command window components



```
CXdb version 1.1.0.1 . Copyright (C) 1991. Convex Computer Corp.
(CXdb)
```

Figure 3 shows the major components of the command window.

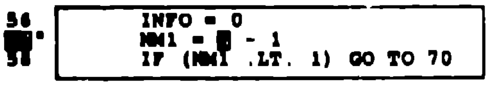
3.2 Source Window

The source window displays the source code for the application being debugged. Scroll-bars and pull-down menus provide quick access to the source code within the application.

The current point of execution is indicated by highlighting the innermost active source units in reverse video (active source units are determined by the PC and the SRT, see Section 2.2.5). Figure 4 shows the source window containing a Fortran code fragment where the assignment on line 40 is highlighted, indicating that the assignment is being performed.

The source window's capabilities are best described by example. The following example shows how highlighting can convey the execution behavior of optimized code. In particular, it demonstrates that a precise understanding of how the code was optimized can be obtained by repetitive stepping at the machine instruction level. The pictures used in this example were taken from an actual CXdb session. Within these pictures, the icon next to line 56, appearing as a ●, is actually an eventpoint icon indicating a breakpoint on line 56 (the icon became unreadable and shifted toward line 57 when the images were scaled to fit within this document).

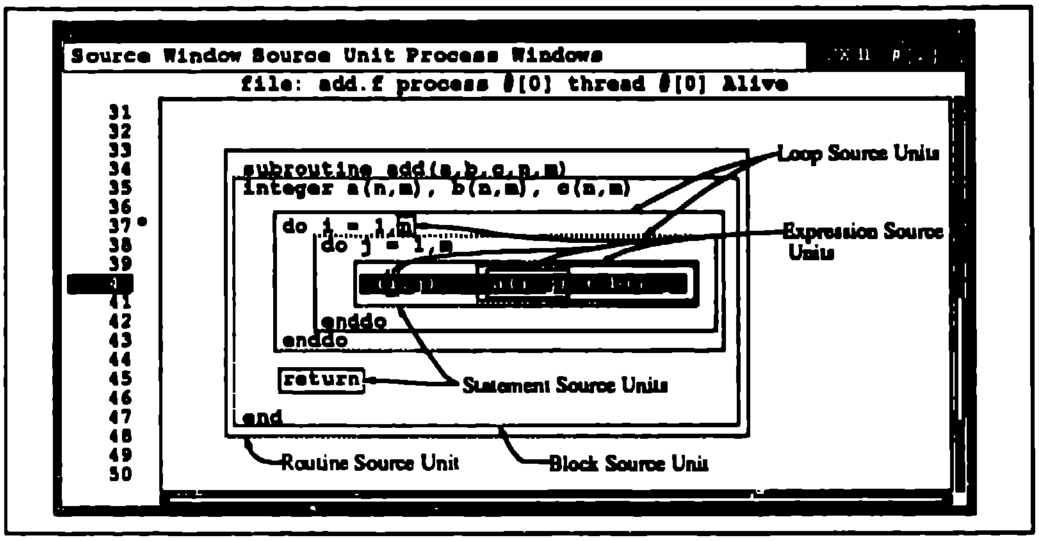
In the presence of optimization, no assumptions can be made about what has happened before the current PC or what will happen after the current PC. The effects of optimization are communicated through patterns of highlighted source units in a sequence, called *program animation*. Consider the following sequence of source window fragments. A breakpoint has been set on the routine whose first statement is on line 56.



```
56       INFO = 0
   ●     NM1 = N - 1
58       IF (NM1 .LT. 1) GO TO 70
```

Notice that when execution stops on entry to the routine, it does not stop on line 56 but on line 57 indicating that instruction scheduling has reordered the computation. The variable N is highlighted indicating that the first instruc

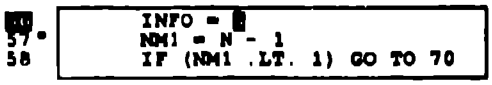
Figure 4. Source window with source unit annotations



```
Source Window Source Unit Process Windows
file: add.f process #[0] thread #[0] Alive
      subroutine add(a,b,c,n,m)
      integer a(n,m), b(n,m), c(n,m)
      do i = 1,m
         do j = 1,n
            c(i,j) = a(i,j) + b(i,j)
         enddo
      enddo
      return
      end
```

tion of the routine is loading N into a register (the disassembly window is extremely useful when debugging at this level, see Section 3.3).

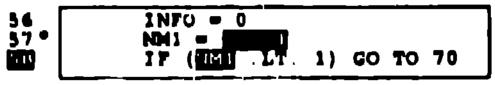
Stepping one machine instruction results in highlighting the constant 0 on the previous line, line 56.



```
56    INFO = 0
57•   NM1 = N - 1
58    IF (NM1 .LT. 1) GO TO 70
```

Like variables, a highlighted constant indicates that the constant is being loaded into a register. Jumping from an expression in one statement to an expression in a subsequent, or even previous, statement is typical of instruction reordering. The above sequence shows that the effects of reordering cannot be conveyed statically, but only dynamically through animation of the program's execution.

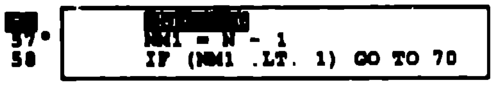
Stepping again results in multiple source units being highlighted.



```
56    INFO = 0
57•   NM1 = N - 1
58    IF (NM1 .LT. 1) GO TO 70
```

This indicates that the highlighted expressions are equivalent. The use of the variable NM1 in the predicate of the conditional is equated by the compiler with the definition of NM1 on line 57 (assignment substitution).

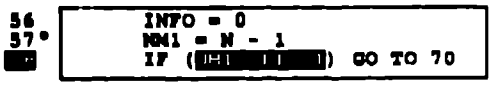
Another step results in the highlighting of the assignment to INFO.



```
56    INFO = 0
57•   NM1 = N - 1
58    IF (NM1 .LT. 1) GO TO 70
```

This indicates that the value of INFO is being updated. In general, assignments will be highlighted during the computation of the *L-value* of the assignment[6] (left-hand side) as well as the store, or register transfer, of the assignment.

Stepping again causes the conditional expression on line 58 to be highlighted.

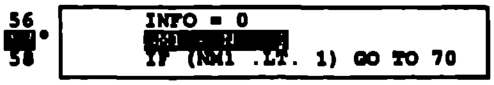


```
56    INFO = 0
57•   NM1 = N - 1
58    IF (NM1 .LT. 1) GO TO 70
```

This indicates that the predicate is about to be evaluated. Recall that the value of NM1 was highlighted (evaluated) in a previous step. The second operand, the constant 1, has not been highlighted indicating that it is used as an immediate operand (the disassembly window can confirm this).

[6] This due to compiler idiosyncracies; not a specification of the highlighting model.

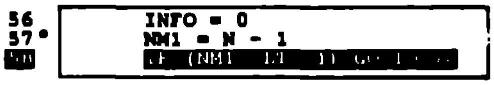
After the predicate is evaluated, the assignment to NM1 occurs.



```
56          INFO = 0
57•         NM1 = N - 1
58          IF (NM1 .LT. 1) GO TO 70
```

Had NM1 not been needed in later computations, the assignment would have never been highlighted, indicating that it had been eliminated (dead-code elimination). Eliminated code, such as dead code or redundant assignments, are conveyed implicitly by never being highlighted (see Section 5.1 for additional discussion of this topic).

Finally, the IF statement on line 58 is about to be executed.



```
56          INFO = 0
57•         NM1 = N - 1
58          IF (NM1 .LT. 1) GO TO 70
```

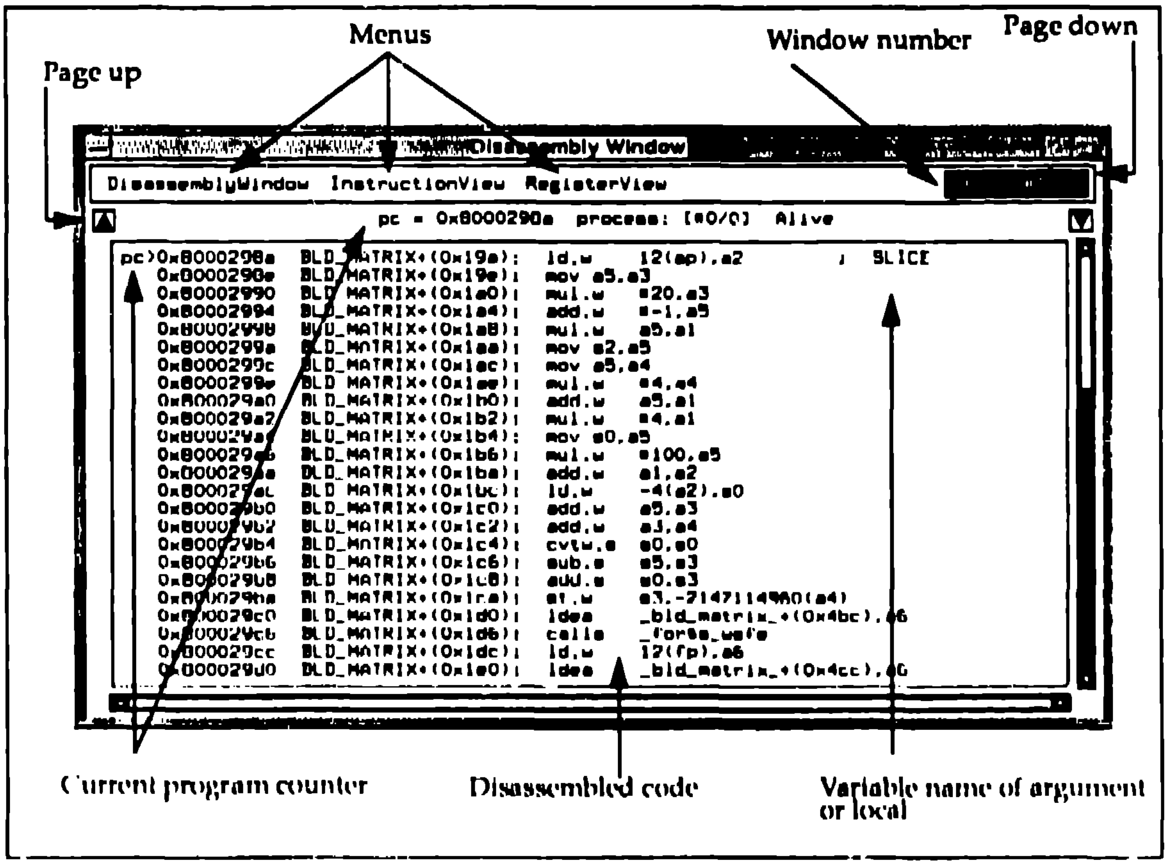
The IF is highlighted at the conditional branch that tests the result of the predicate. In this case the consequent is a GOTO, which has been folded into the sense of the conditional branch. When the consequent is taken, the GOTO will not be highlighted, because control is short circuited to label 70.

This example illustrates that highlighting in reverse video can convey the effects of numerous optimizations. However, it is not sufficient for more involved optimizations such as hoisting or sinking. See Section 5.1 for potential solutions to some existing highlighting problems.

3.3 Disassembly Window

The disassembly window is the main access point to the machine level debugging features of CXdb. The disassembly window provides access to the disassembled object code of the target process. Figure 5 shows an example of the disassembly window. Each disassembled instruction is

Figure 5. Disassembly Window



```
pc = 0x8000298a  process: [#0/0]  Alive

pc>0x8000298a  BLD_MATRIX+(0x19a):  ld.w    12(ap),a2          ;  SLICE
   0x8000298e  BLD_MATRIX+(0x19e):  mov a5,a3
   0x80002990  BLD_MATRIX+(0x1a0):  mul.w   #20,a3
   0x80002994  BLD_MATRIX+(0x1a4):  add.w   #-1,a5
   0x80002998  BLD_MATRIX+(0x1a8):  mul.w   a5,a1
   0x8000299a  BLD_MATRIX+(0x1aa):  mov a2,a5
   0x8000299c  BLD_MATRIX+(0x1ac):  mov a5,a4
   0x8000299e  BLD_MATRIX+(0x1ae):  mul.w   a4,a4
   0x800029a0  BLD_MATRIX+(0x1b0):  add.w   a5,a1
   0x800029a2  BLD_MATRIX+(0x1b2):  mul.w   a4,a1
   0x800029a4  BLD_MATRIX+(0x1b4):  mov s0,a5
   0x800029a6  BLD_MATRIX+(0x1b6):  mul.w   #100,a5
   0x800029aa  BLD_MATRIX+(0x1ba):  add.w   a1,a2
   0x800029ac  BLD_MATRIX+(0x1bc):  ld.w    -4(a2),s0
   0x800029b0  BLD_MATRIX+(0x1c0):  add.w   a5,a3
   0x800029b2  BLD_MATRIX+(0x1c2):  add.w   a3,a4
   0x800029b4  BLD_MATRIX+(0x1c4):  cvtw.s  s0,s0
   0x800029b6  BLD_MATRIX+(0x1c6):  sub.s   s5,s3
   0x800029b8  BLD_MATRIX+(0x1c8):  add.s   s0,s3
   0x800029ba  BLD_MATRIX+(0x1ca):  st.w    s3,-2147114960(a4)
   0x800029c0  BLD_MATRIX+(0x1d0):  ldea    _bld_matrix_+(0x4bc),a6
   0x800029c6  BLD_MATRIX+(0x1d6):  calls   _forts_safe
   0x800029cc  BLD_MATRIX+(0x1dc):  ld.w    12(fp),a6
   0x800029d0  BLD_MATRIX+(0x1e0):  ldea    _bld_matrix_+(0x4cc),a0
```

annotated with the name of the referenced variable. The LRT, see Section 2.2.2, maps machine locations to variable names[7].

The disassembly window has an operating mode called *auto-update* mode. In auto-update mode, the disassembly window is updated every time the process stops. The region of object code surrounding the PC is disassembled and displayed. This mode is most useful when visualizing program execution as described in Section 3.2.

The target process' complete register state can be displayed through pop-up windows created from the disassembly window's pull-down menus. These register pop-ups are described below.

3.3.1 Register Pop-ups

The disassembly window's pull-down menus provide access to pop-up windows displaying the complete register state of the process being debugged. The following register sets can be viewed:

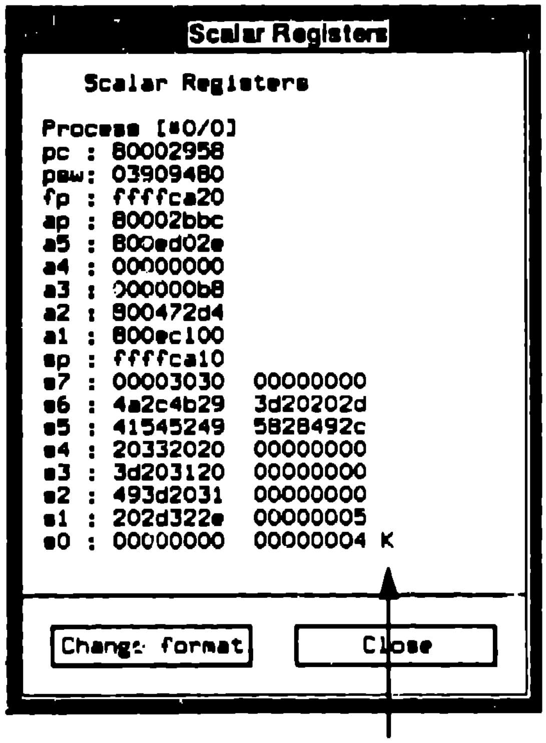
- Scalar registers
- Vector registers
- Communication registers
- PSW detail

Whenever the pop-up windows are displayed their contents are kept up to date. When the process stops, the register values are calculated and re-displayed. Figure 6 shows an example of the scalar register pop-up window. Similar to the annotations on the disassembly window, the register pop-ups are labeled with variables currently located within each register.

3.4 Examine Window

The examine window is another tool for working below the source level It provides a formatted view onto a memory region. The user controls the display format with a pop-up dialog window. The formats available include a

Figure 6. Scalar register pop-up window



```
Scalar Registers

Process [#0/0]
pc : 80002958
psw: 03909480
fp : ffffca20
ap : 80002bbc
a5 : 800ed02e
a4 : 00000000
a3 : 000000b8
a2 : 800472d4
a1 : 800ec100
sp : ffffca10
s7 : 00003030  00000000
s6 : 4a2c4b29  3d20202d
s5 : 41545249  5828492c
s4 : 20332020  00000000
s3 : 3d203120  00000000
s2 : 493d2031  00000000
s1 : 202d322e  00000005
s0 : 00000000  00000004 K
```

Variable K currently in register S0

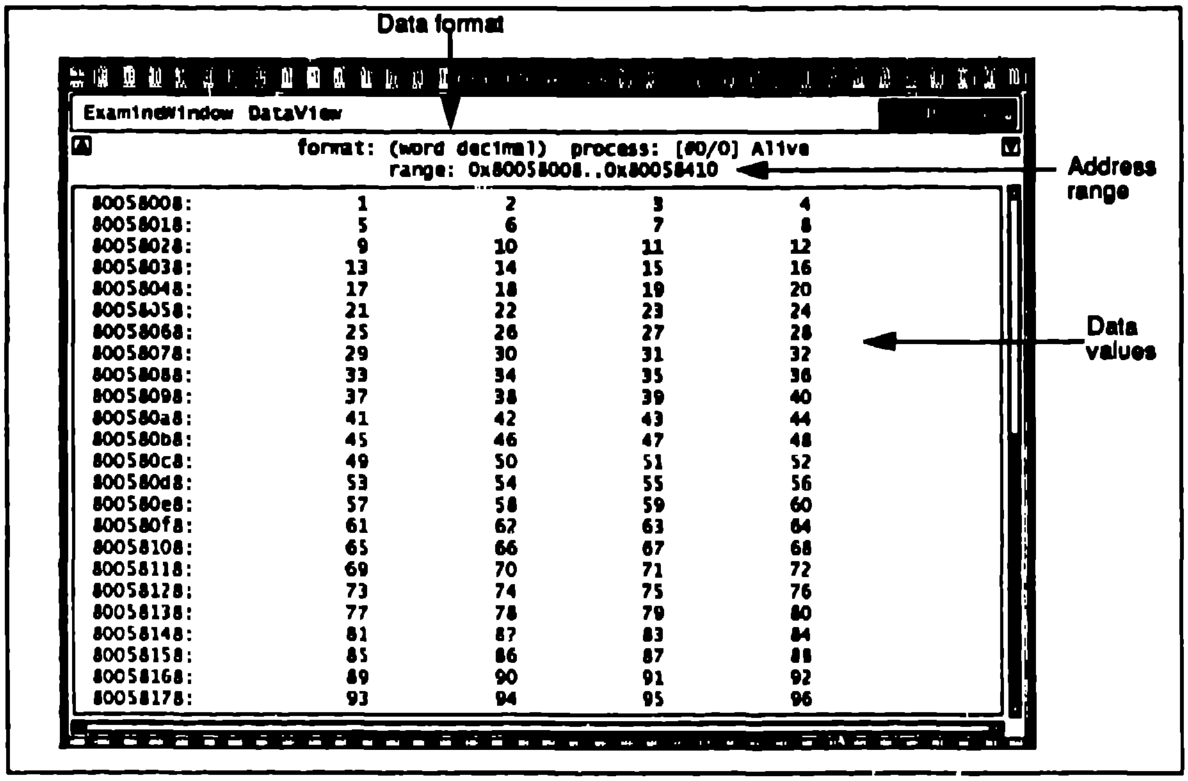
choice of word size (from one to 16 bytes) and type. The possible types are:

- decimal
- unsigned decimal
- hexadecimal
- octal
- binary
- character
- float (fixed and scientific notation)
- Fortran Logical
- Fortran Complex

The examine window, like the disassembly window, has an auto-update mode where its contents are updated each

[7] CXdb is capable of working with only partial debugging information (that is, when some of the source files have not been compiled for debugging) In such cases, CXdb may not be able to map the address reference to a variable name.

Figure 7. Examine window



```
ExamineWindow DataView
                format: (word decimal)  process: [#0/0] Alive
                        range: 0x80058008..0x80058410
80058008:          1          2          3          4
80058018:          5          6          7          8
80058028:          9         10         11         12
80058038:         13         14         15         16
80058048:         17         18         19         20
80058058:         21         22         23         24
80058068:         25         26         27         28
80058078:         29         30         31         32
80058088:         33         34         35         36
80058098:         37         38         39         40
800580a8:         41         42         43         44
800580b8:         45         46         47         48
800580c8:         49         50         51         52
800580d8:         53         54         55         56
800580e8:         57         58         59         60
800580f8:         61         62         63         64
80058108:         65         66         67         68
80058118:         69         70         71         72
80058128:         73         74         75         76
80058138:         77         78         79         80
80058148:         81         82         83         84
80058158:         85         86         87         88
80058168:         89         90         91         92
80058178:         93         94         95         96
```

time the process stops. With the examine window in auto-update mode you can watch a segment of process memory, such as part of an array, change as you step process execution. Figure 7 shows an example of the examine window.

3.5 Stack Window

The stack window presents a symbolic backtrace of the current program stack contents. The information displayed is equivalent to that of the `backtrace` command. It contains an entry for each call frame on the stack. By default, the stack window is in auto-update mode. Each entry contains the following information:

- Frame number
- Execution address at time of call
- Name of function
- Argument names and values
- Symbolic program location
- Current frame indicator

Figure 8 shows an example of the stack window.

The stack window provides point-and-click access to a pop-up window that displays detailed information about selected stack frames. The pop-up displays the following information:

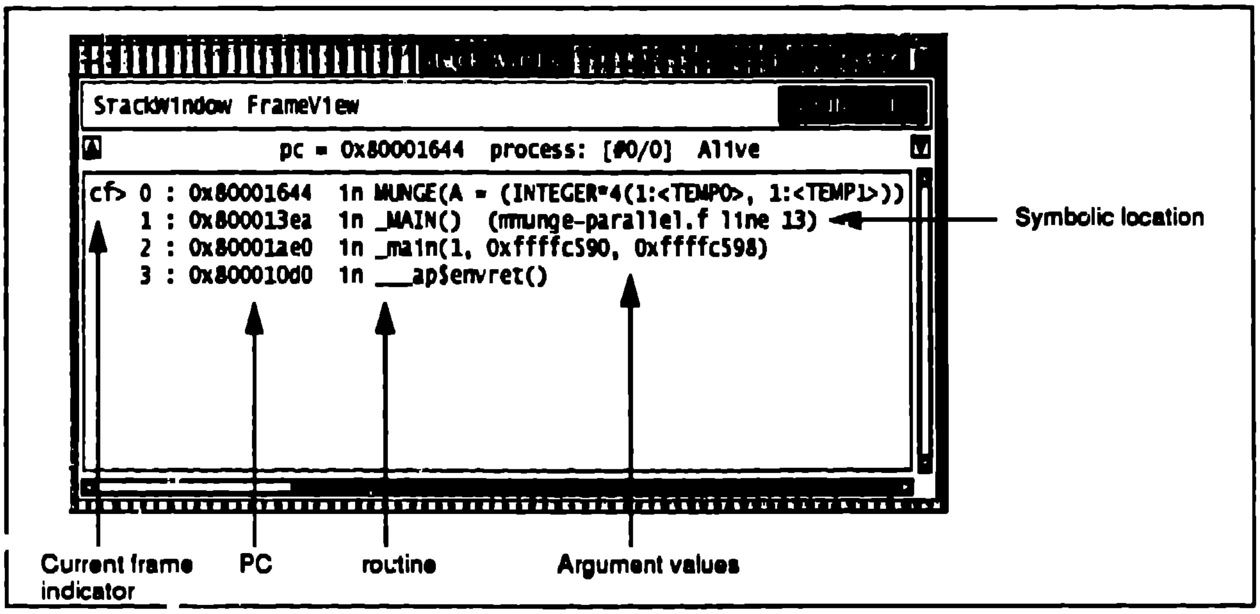
- Frame address
- PC within frame and symbolic location
- Source language of frame
- Argument names and values
- Local variable names and values

Figure 9 shows an example of the frame pop-up window.

3.6 Process I/O Window

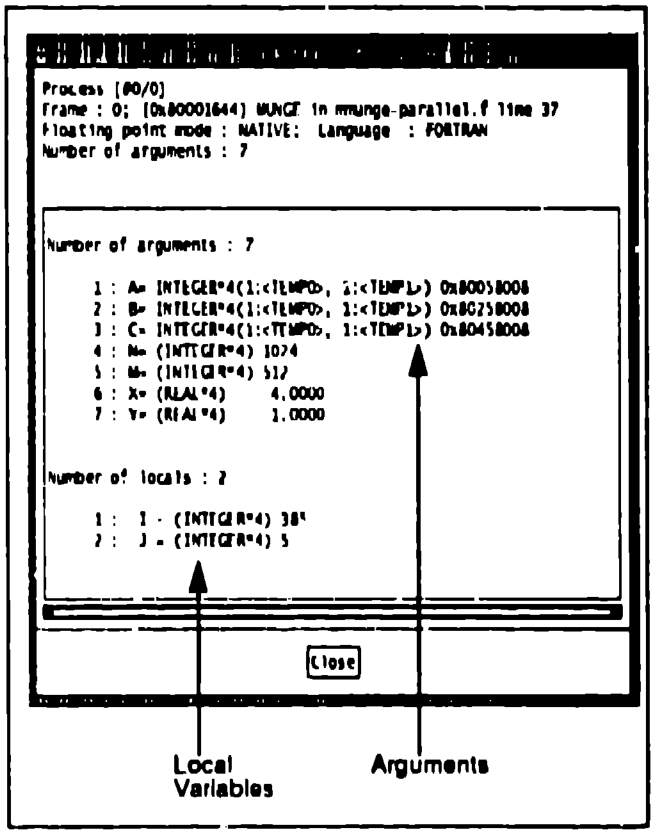
The process I/O window isolates all the target process' I/O activities. This isolation provides two distinct interaction

Figure 8. Stack window



```
StackWindow FrameView
              pc = 0x80001644  process: [#0/0]  Alive
cf> 0 : 0x80001644  in MUNGE(A = (INTEGER*4(1:<TEMP0>, 1:<TEMP1>))
    1 : 0x800013ea  in _MAIN()  (mmunge-parallel.f line 13)
    2 : 0x80001ae0  in _main(1, 0xffffc590, 0xffffc598)
    3 : 0x800010d0  in __ap$envret()
```

Figure 9. Frame pop-up window



```
Process [#0/0]
Frame : 0; [0x80001644] MUNGE in mmunge-parallel.f line 37
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 7

Number of arguments : 7

    1 : A= INTEGER*4(1:<TEMP0>, 1:<TEMP1>) 0x80058008
    2 : B= INTEGER*4(1:<TEMP0>, 1:<TEMP1>) 0x80258008
    3 : C= INTEGER*4(1:<TEMP0>, 1:<TEMP1>) 0x80458008
    4 : N= (INTEGER*4) 1024
    5 : M= (INTEGER*4) 512
    6 : X= (REAL*4)     4.0000
    7 : Y= (REAL*4)     1.0000

Number of locals : 2

    1 :  I - (INTEGER*4) 385
    2 :  J = (INTEGER*4) 5
```

contexts: one for CXdb and another for the target process. The X window application, xterm, manages this window. Xterm provides complete CRT emulation allowing the user to debug applications that do full screen formatting. The separation of interaction also guarantees that the target process' control of the output screen will not be disturbed by interacting with CXdb. Because this window is just an xterm, no example of it will be shown.

3.7 Help Window

The help window displays online help text relating to various CXdb topics. The help window appears when you use the help command or press the *help* expert button on the command window.

The online help system contains all topics in the *CONVEX CXdb Reference* manual [Conv91b]. These topics cover four categories of information:

- **Concepts** — Explanations of the major topics involved with using CXdb.
- **Commands** — Descriptions of all the CXdb commands.
- **Parameters** — Descriptions of some of the major command parameters.
- **CXdb messages** — Explanations of informational messages and error messages generated by CXdb.

You can either request help on a listed topic or you can search the help files to locate a particular word or phrase.

Figure 10 shows an example of the help window.

4.0 Command Language Components

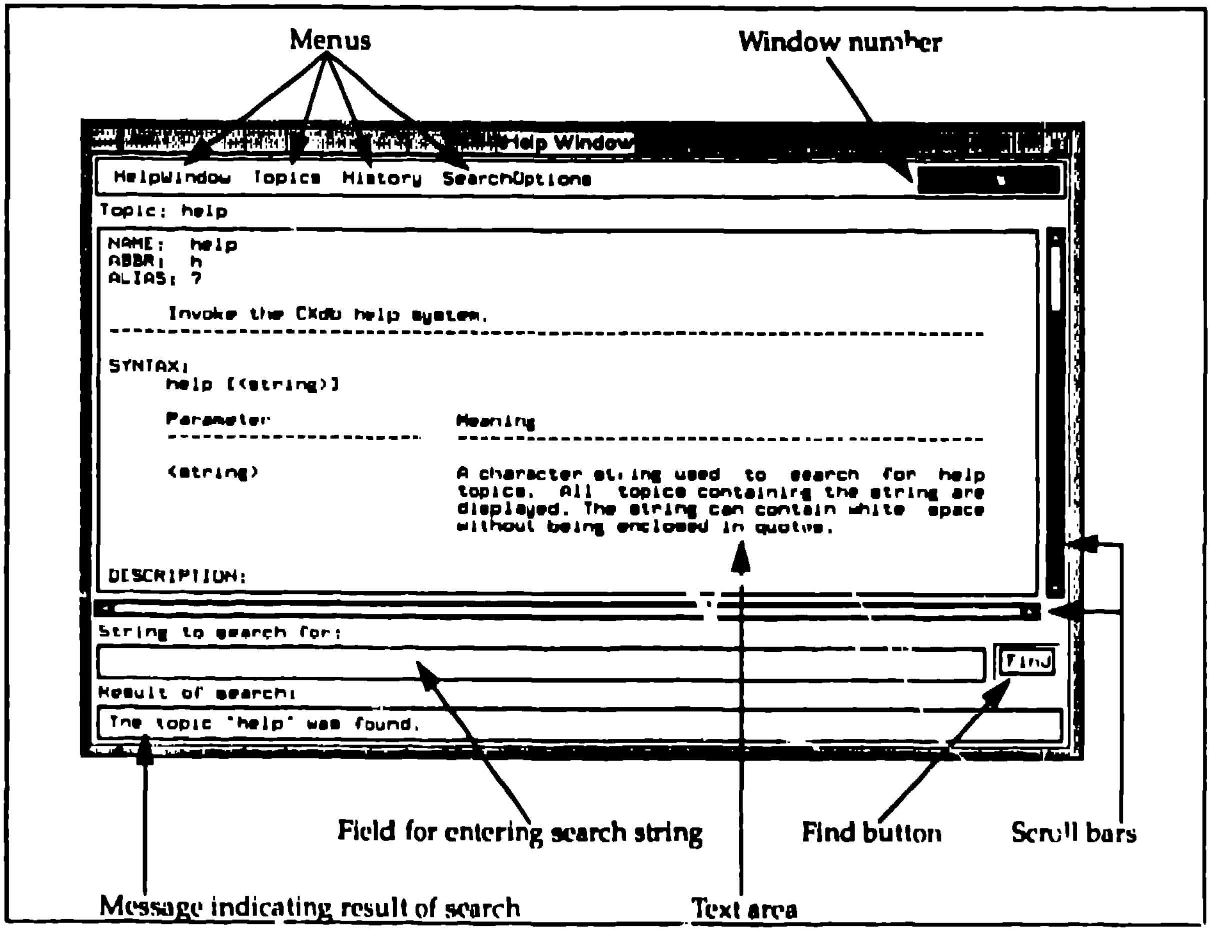
CXdb's command language was designed to provide a feature-rich and powerful debugging environment. CXdb's command environment provides the following features:

- A log of all commands entered (including menu selections) can be maintained.
- The output of a command can be redirected to one or more files (when combined with command logging an entire record of a CXdb session can be created).
- Eventpoint handlers can be created that consist of any sequence of CXdb commands[8]. When handlers are combined with output redirection, a user can create powerful new features like tracing a variable's value or recording parameters on each call to a function.
- Customize the command language with command aliases and macros.

Many of CXdb's commands focus specifically on handling optimized code. These specialized commands are described in the following sections.

[8] Except commands that cause process execution, like *step*. The *resume* command can be used to continue process execution.

Figure 10. Help window



Topic: help

```
NAME:  help
ABBR:  h
ALIAS: ?

     Invoke the CXdb help system.
-------------------------------------------------------------------------

SYNTAX:
     help [<string>]

     Parameter                 Meaning
     ---------------------     ----------------------------------------

     <string>                  A character string used  to  search  for  help
                               topics.  All  topics containing the string are
                               displayed. The string can contain white  space
                               without being enclosed in quotes.

DESCRIPTION:
```

String to search for:

Result of search:

The topic 'help' was found.

4.1 Informational Commands

CXdb provides an extensive set of commands that provide information on the state of CXdb and the target process. Information may be obtained on the following broad categories:

- CXdb's configuration
- Defined eventpoints
- Command aliases and macros
- Defined key-bindings
- Default signal handling settings
- Process state
- Process registers
- Process stack
- Executable organization

There are three commands that are specifically aimed at optimized code handling: `info expression`; `info line`; and `info sourceunit`. Each of these commands is described below.

4.1.1 Info Expression

The `info expression` command is an extremely versatile command that provides detailed information about a language expression[9]. The `info expression` command displays the following information about the specified language expression, when applicable:

Object Type	Type of object represented by the expression; one of *identifier*, *expression result*, or *debugger variable*
Location	Current machine location(s) of the variable
Size	Total object size
Type	Expression data type
Value	Expression's current value
Liveness	PC ranges and associated machine locations where the variable's value is available. Outside these liveness ranges, the value of the variable is not available.
Synthesized Variables	Variables generated by the compiler as part of the optimization process that are derived from the variable
Orientation	Array orientation: row- or column-major
Bounds	Array bounds
Entry	Entry point for a function
Return type	Data type of the value returned by a function
Prototype	The complete ANSI style prototype of a function
Var Type	Type of object represented by a debugger variable

The `info expression` command is the only way to determine all of the locations that a variable may occupy (that is, access to the LRT) or what synthesized variables have been derived from it (that is, access to the XPT). Figure 11 shows an example of an `info expression` command's output that includes variable liveness ranges. Figure 12 shows output that includes derived synthesized variables.

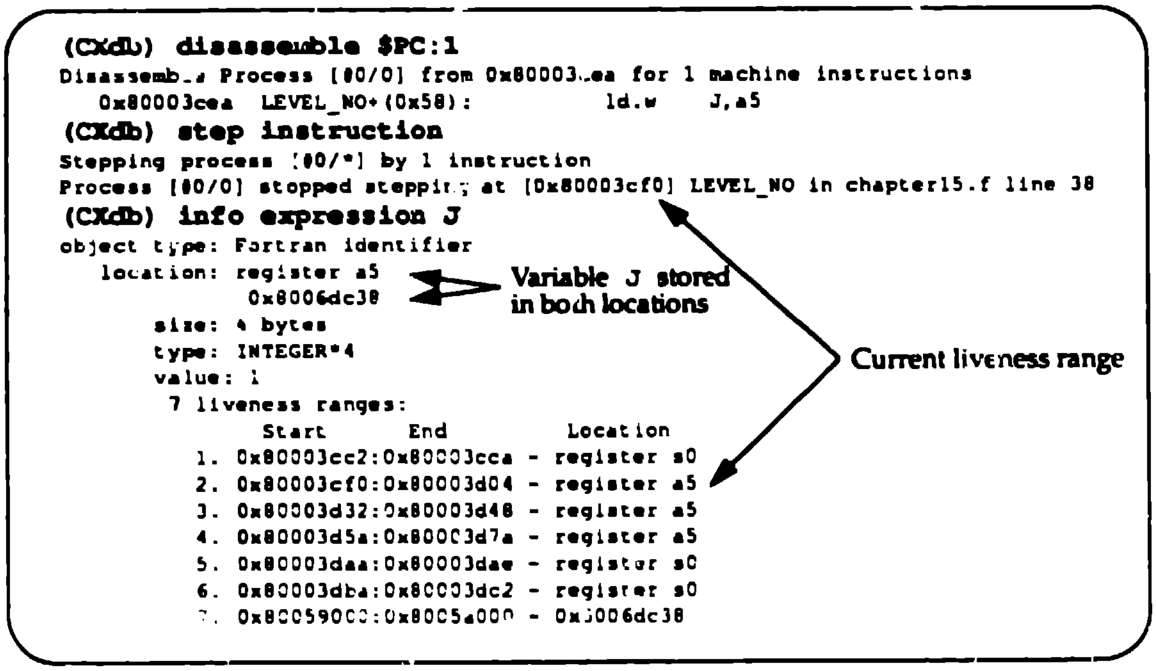
4.1.2 Info Line and Info Sourceunit

The `info line` and `info sourceunit` commands provide access to the SRT information. The `info line` command displays information on all source units that start on the specified line, while the `info sourceunit` commands displays the same data on a specific source unit. The following information is displayed:

- Source unit index
- Regions of object code generated by this source unit. A zero region indicates that no object code was generated for the source unit.
- Source text row and column positions for the start and end of the source unit
- Source unit kind, see Table 3, Source Unit Granularities
- Source unit text extracted from the source file

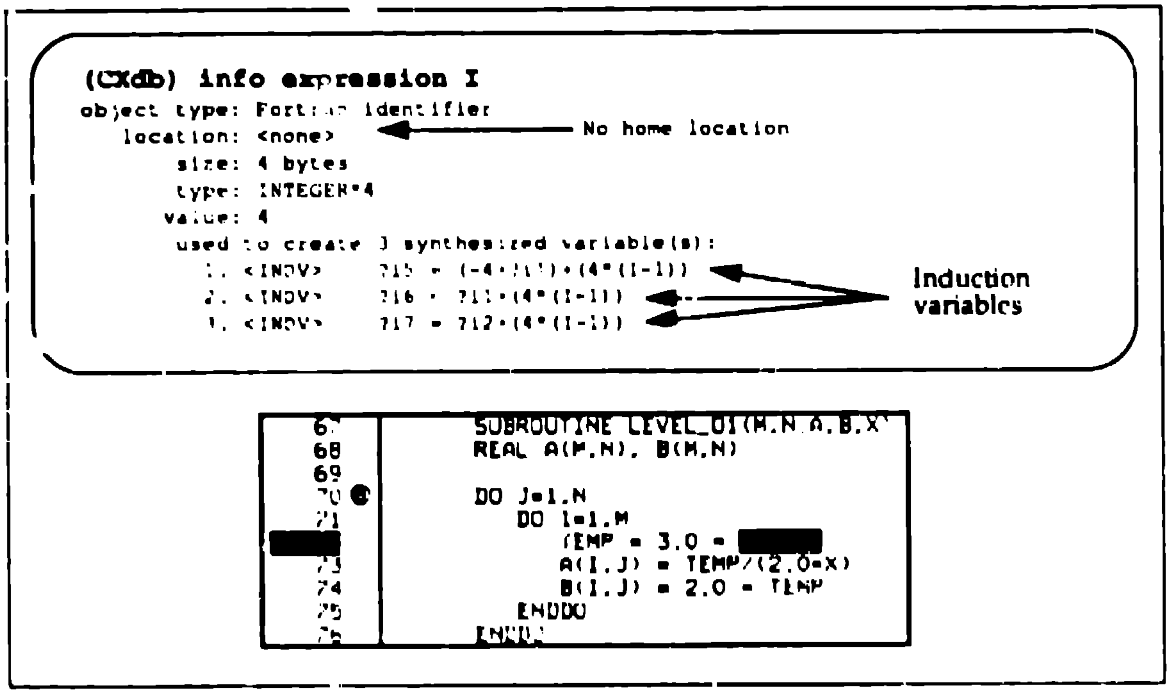
[9] A language expression is any expression that is valid in the current source language context with extensions for using debugger variables and specifying address ranges and offsets.

Figure 11. Info Expression showing liveness ranges



```
(CXdb) disassemble $PC:1
Disassemb.. Process [#0/0] from 0x80003.ea for 1 machine instructions
   0x80003cea  LEVEL_NO+(0x58):           ld.w    J,a5
(CXdb) step instruction
Stepping process [#0/*] by 1 instruction
Process [#0/0] stopped steppi.. at [0x80003cf0] LEVEL_NO in chapter15.f line 38
(CXdb) info expression J
object type: Fortran identifier
   location: register a5
             0x8006dc38
       size: 4 bytes
       type: INTEGER*4
      value: 1
     7 liveness ranges:
            Start        End           Location
        1. 0x80003cc2:0x80003cca - register s0
        2. 0x80003cf0:0x80003d04 - register a5
        3. 0x80003d32:0x80003d48 - register a5
        4. 0x80003d5a:0x80003d7a - register a5
        5. 0x80003daa:0x80003dae - register s0
        6. 0x80003dba:0x80003dc2 - register s0
        7. 0x80059000:0x8005a000 - 0x8006dc38
```

Figure 12. Info expression showing synthesized variables



```
(CXdb) info expression I
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 4
      used to create 3 synthesized variable(s):
        1. <INDV>    T15 = (-4+T11)+(4*(I-1))
        2. <INDV>    T16 = T11+(4*(I-1))
        3. <INDV>    T17 = T12+(4*(I-1))
```

```
67          SUBROUTINE LEVEL_01(M,N,A,B,X)
68          REAL A(M,N), B(M,N)
69
70 ●        DO J=1,N
71             DO I=1,M
[redacted]        TEMP = 3.0 * [redacted]
73                A(I,J) = TEMP/(2.0*X)
74                B(I,J) = 2.0 * TEMP
75             ENDDO
76          ENDDO
```

The output of this command can be used to determine dead code (that is, source units that produced no object code), and code motion. Figure 13 shows an example of the `info line` command.

4.2 Stepping and Granularities

Source units provide a much more detailed understanding of the syntactic breakdown of an application's source code than is possible with the current STAB mechanism. This enhanced understanding enables much finer grain control over incremental program execution.

CXdb extends the standard `step` and `next` commands to operate on the granularities listed in Table 3. Stepping by each of the granularities provides its own unique advantage. These advantages are listed below.

Routine Stepping by *routine* proceeds from routine entry to routine entry. This is very useful when debugging unfamiliar code where the run-time call sequence is unknown.

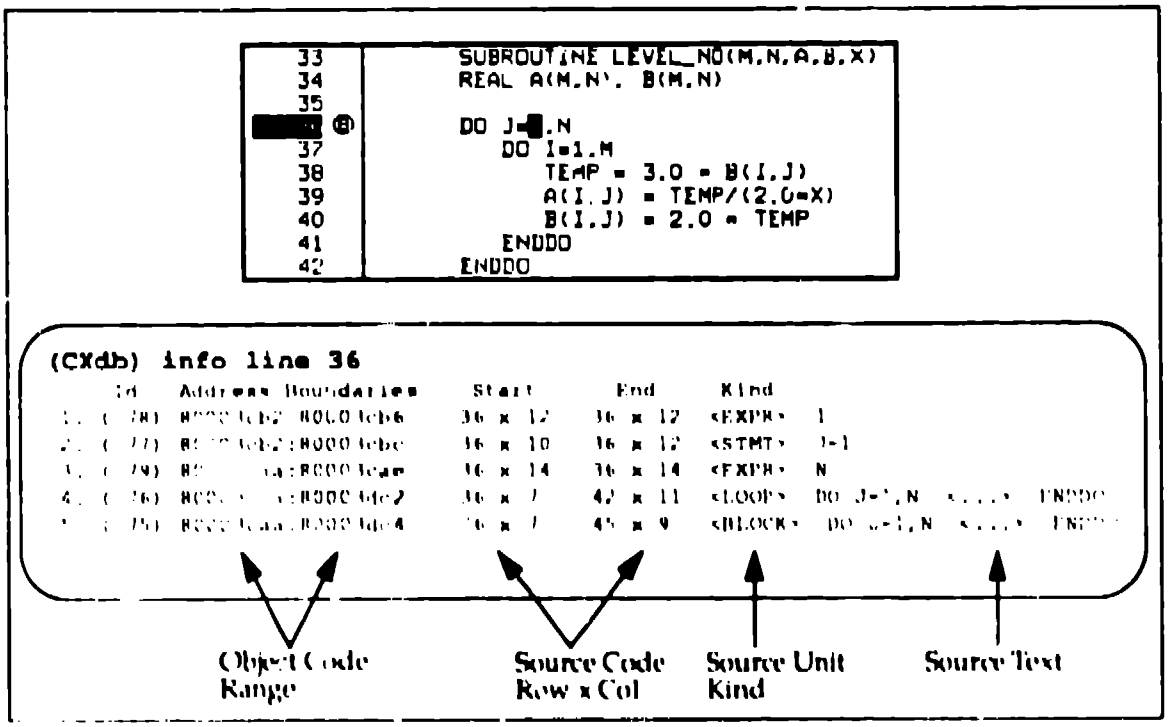
Loop In many programs the loops are the most interesting components (especially when focusing on the optimizations that apply to loops). Stepping by *loop* is an efficient mechanism for moving from loop to loop.

Block Once inside a loop, stepping by *block* continues execution to the next loop iteration. Outside of a loop, it is a quick way of stepping to the next lexical scope change.

Statement Stepping by *statement* provides the standard stepping mechanism that most debugger users are familiar with. However, its usefulness is extremely limited when working on optimized code.

Expression Stepping by *expression* is extremely beneficial when working on optimized code. Many optimizations occur at the expression level.

Figure 13. Info line example



```
33      SUBROUTINE LEVEL_NO(M,N,A,B,X)
34      REAL A(M,N), B(M,N)
35
36 Ⓑ    DO J=1.N
37         DO I=1.M
38            TEMP = 3.0 * B(I,J)
39            A(I.J) = TEMP/(2.0*X)
40            B(I.J) = 2.0 * TEMP
41         ENDDO
42      ENDDO
```

```
(CXdb) info line 36
      Id    Address Boundaries      Start      End       Kind
 1. ( 78) 8000[illegible]:8000lcb6   36 x 12    36 x 12   <EXPR>   1
 2. ( 77) 8[illegible]:8000lcbe      36 x 10    36 x 12   <STMT>   J=1
 3. ( 79) 8[illegible]:8000lcae      36 x 14    36 x 14   <EXPR>   N
 4. ( 76) 8000[illegible]:8000ldc2   36 x 7     42 x 11   <LOOP>   DO J=1,N  <....>  ENDDO
 5. ( 75) 8000[illegible]:8000ldc4   36 x 7     45 x 9    <BLOCK>  DO J=1,N  <....>  ENDDO
```

Typical debuggers, like dbx or gdb, implement stepping by repeatedly stepping the target process by *machine instruction* until it reaches the next statement. This mechanism works fairly well for statement level stepping, but CXdb provides multiple stepping granularities. Using an instruction stepping technique on a command like `finish loop` would be infeasible because the number of machine instructions executed within a loop can be enormous.

To solve this problem, CXdb uses a mechanism called *transient breakpoints*. A transient breakpoint is a breakpoint placed by CXdb to implement incremental execution commands like `step` or `next`. The SRT is used to determine the object code ranges occupied by a given source unit (for example, a loop or statement). Transient breakpoints are then placed at the first instruction of each object code range.

Once the transient breakpoints have been placed, the process is executed at normal execution speed. When the target stops with a breakpoint trap, the step operation is complete[10]. Stepping with transient breakpoints works very well for routine, loop, block, and statement granularities. However, stepping by expression is implemented using the standard machine level stepping approach. Because of the large number of expression within a routine, it is impractical to implement with breakpoints.

Because a machine instruction is not considered a granularity (because it isn't a source unit), the `step instruction` command provides stepping control at this level. When working on highly optimized code `step instruction` and `step expression` become two of the most frequently used commands.

In addition to the standard `step` and `next` commands, CXdb offers another command, `finish`, that allows you to *step out of* a specified source unit. For example, assume execution has stopped somewhere within a loop and you want to run until the loop exits. With most debuggers, you would have to locate the end of the loop manually and place a breakpoint there before continuing execution. With CXdb, you can simply type `finish loop` and CXdb will do all the work. Similarly, `finish routine` resumes execution until the current routine is about to exit. Note that execution is halted *before* control returns to the calling routine. This provides the user the opportunity to interrogate the program state prior to leaving the routine. The `finish` command can be used on any granularity, but loop and routine are the most useful.

4.3 Specialized Eventpoints

CXdb's eventpoint mechanism supports the following capabilities:

Breakpoint Halt program execution when a specified location is reached.

Tracepoint Print a message when execution reaches a specified location.

Watchpoint Halt program execution when a specified memory region is modified.

Relation Halt program execution when a specified language expression evaluates to TRUE.

Exec Halt program execution when it performs an exec(2) system call.

In addition to these features, CXdb provides two eventpoints specifically for handling parallel optimizations. They are:

Spawn Halt execution when any new threads of execution are created. This detects regions that have been parallelized.

Join Halt execution when any thread joins (that is, exits). This detects when a thread has completed its portion of a parallel region.

4.4 Multiple Thread Support

CXdb was designed specifically to handle parallel applications. Most of the GUI windows are instantiated on a per-thread basis (see Section 3.0). There are special eventpoints for detecting thread creation and death (see Section 4.3). Finally, the command language was built to support multiple threads in a consistent manner.

Each command may be prefixed with a *command focus*. A command focus specifies the specific process[11] and threads within that process to be affected by the command For example, the command

```
(CXdb) :t1,2 step
```

[10] This is a greatly simplified explanation of the inner workings of this process, but it is representative of the mechanism

will step only threads 1 and 2, leaving the other threads at their current location. The command

```
(CXdb)  :t0,3 backtrace
```

produces a stack backtrace for both thread 0 and thread 3. For commands that create eventpoints, specifying a command focus determines which threads will be affected by the eventpoint. For example, the command

```
(CXdb)  :t0 break routine foo
```

places a breakpoint at the entry point to the routine foo that will only affect thread 0. Any other thread that encounters the breakpoint will not be stopped by it. The default focus is all threads, which is equivalent to :t* in a focus specification.

5.0 Future Directions

Although CXdb is very full-featured, it is nowhere near "done". This section presents a number of areas that may be pursued as future enhancements of CXdb.

5.1 Enhanced Highlighting Techniques

As shown in previous examples, simple reverse video highlighting is sufficient for many of the scalar optimizations. However, it can not convey some of the more involved, loop-oriented optimizations such as hoisting or sinking. Also, code removal optimizations (for example, dead code removal or redundant assignment elimination) are conveyed implicitly by never highlighting the associated source units.

Some possible solutions to these problems are discussed in the following sections.

5.1.1 Hierarchical Highlighting

Consider the hoisting of invariant code out of a loop. Using reverse video, loop invariant code will be highlighted when the loop becomes active. The highlighting does not convey whether the code is inside or outside the loop. To convey this, *hierarchical highlighting* is needed.

In one form of hierarchical highlighting all active source units, not just the innermost source units, would be highlighted. This must be done in a way so that nested source units can be distinguished. Inactive source units are not highlighted. With hierarchical highlighting, motion out of loops can be conveyed by highlighting the hoisted expression, but not the loop body in which it is textually nested.

5.1.2 Conveying Dead Code

Currently, dead code is indicated implicitly, that is, if it never gets highlighted, then it is dead code. It would be beneficial to make the indication of dead code explicit. For example, one potential method is to use a different font when displaying dead code in the source window. This would provide a simple but effective mechanism for explicitly indicating dead code.

5.2 Multi-Process Debugging

CXdb's command language was designed with the final goal of supporting multi-process debugging from a single session. However, this has not yet been implemented.

Being able to debug multiple processes is beneficial when trying to debug processes which fork and exec child processes. CXdb will be able to "inherit" the child processes created by the initial target process and then present them to the user for debugging control.

5.3 Multi-Architecture Support

In the evolution of heterogeneous computing it will become necessary to debug processes running on dissimilar architectures. To perform this, the debugger will have to be able to handle the interaction of processes which have different register sets, memory layouts, etc. Also, this feature ties in very closely with remote debugging, discussed in the next section.

5.4 Remote Debugging

Current day development environments typically consist of networks of workstations with a central compute server for handling large scale problems. Environments are becoming more and more distributed. To support this environment well, CXdb will need to be able to debug a process running on a remote platform. This will also be a requirement for kernel debugging.

[11] CXdb's command language was designed with multiple process support in mind. However, multi process debugging is not supported in the current release.

6.0 Summary

CXdb is an extremely effective tool for solving difficult development problems, specifically debugging optimized code. By using a new model to communicate the results of compiler optimizations to the debugger, CXdb overcomes the limitations of STAB- and DWARF-based debugging.

CXdb's Compiler-Debugger Interface (CDI) increases disk space requirements and compilation time, but much more effectively provides complex information describing optimized code. CXdb's Graphical User Interface (GUI) presents complex program information in easily accessible components, allowing rapid access to information with advanced program animation techniques that are needed to solve complex debugging tasks.

The CDI and GUI, along with the ability to comprehend granularity at a level needed for optimized code, make CXdb an effective tool for understanding compiler optimizations and solving difficult development problems.

7.0 Acknowledgments

I would like to acknowledge my colleagues who participated in the definition and development of CXdb, in particular, Gary Brooks, Russ Buyse, Mark Chiarelli, Gil Hansen, David Lingle, Steve Simmons, Jeff Woods, Mike Garzione, Ken Harward, Ray Cetrone, Keith Knox, Lloyd Tharel, Tim Powell, Rich Bleikamp, and Bill Torkelson. Finally, I would like to thank Kathy Harris for her skill as an editor and efforts at making thi. paper publishable.

8.0 Trademarks and Copyrights

9.0 References

[BrHa91] Brooks, G., Hansen, G., and Simmons, S., "A Compiler-Debugger Interface for the Visual Debugging of Optimized Code", Convex Computer Corporation, (unpublished) (September 1991).

[BuCh91] Buyse, R. and Chiarelli, M., "A User Interface Strategy for CXdb", presented at Xhibition 91.

[CoMe88] Coutant, D. S., Meloy, S., and Ruscetta, M., "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", *ACM Proceedings on Programming Language Design and Implementation*, SIGPLAN Notices 23, 7 (July 1988) 125-134.

[Conv90] CONVEX FORTRAN Optimization Guide, 2nd Edition, Convex Computer Corporation (1990).

[Conv91a] CONVEX C Optimization Guide 2nd Edition, Convex Computer Corporation (1991).

[Conv91b] CONVEX CXdb Reference, 1st Edition, Convex Computer Corporation (1991).

[Conv91c] CONVEX CXdb User's Guide, 1st Edition, Convex Computer Corporation (1991).

[Henn82] Hennessy, J., "Symbolic Debugging of Optimized Code", *ACM Transactions on Programming Languages and Systems*, 3, 3 (July 1982) 323-344.

[Lu91] Lu, L. C., "A Unified Framework for Systematic Loop Transformations", *ACM Proceedings of the Symposium on Principles & Practice of Parallel Programming*, **26**, 7 (July 1991) 28-38.

[OSF91] Open Software Foundation, "OSF/Motif Programmer's Guide", Revision 1.1, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Sark90] Sarkar, V., "Instruction Reordering for Fork-Join Parallelism", *ACM Proceedings on Programming Language Design and Implementation*, SIGPLAN Notices 25, 6 (June 1990) 322-336.

[WaSr85] Wall, D., Srivastava, A. and Templin, F., "A Note on Hennessy's "Symbolic Debugging of Optimized Code"", *ACM Transactions on Programming Languages and Systems*, 7, 1 (January 1985) 176-181.

[Zell83] Zellweger, P. T., "An Interactive High-Level Debugger for Control-Flow Optimized Programs" in *ACM Proceedings of the Software Engineering Symposium on High-Level Debugging*, SIGPLAN Notices, 18, 8 (August 1983) 159-171.

[Zell84] Zellweger, P. T., "Interactive Source-Level Debugging of Optimized Programs", Ph.D. Thesis, University of California, Berkeley, CA, (1983); also Report CSL-84-5, Xerox PARC, Palo Alto, CA (May 1984).

[Zura90] Zurawski, L. W., "Debugging Optimized Code with Expected Behavior", University of Illinois at Urbana-Champaign (unpublished) (August 1990)

Debugging at the National Security Agency: "The State of the Use Message"

Tom Myers

ctmyers@super.org

National Security Agency

9800 Savage Road
Fort George G. Meade, MD
20755-6000
301-688-7164

Outline

- Preliminaries
- History
- Current Situation
- Trends
- Conclusions

Preliminaries

We are not

that different!

Preliminaries

- Diverse User Populaticn
    - Scientist/programmer (80%)
        - Interactive experimentation
        - Algorithm development
        - Correctness primary concern
    - Programmer/scientist (15%)
        - Production code development
        - Tool development
        - Performance a major concern
- Diverse Workloads
    - Interactive - consistent, fast computer response time
        - Compile/link (seconds)
        - Run/debug (sub-second)
        - Edit/think (sub-second)
    - Batch/production - run time or throughput
    - Background

History - Where we've been

- Since 1966 users have been accustomed to fast, responsive, simple window based editing and debugging
- High speed (~250kb/s) terminals displayed 20 by 80 text windows one at a time
- The debugger was a tool for browsing files and it knew about the format of special files such as the memory image of a running process, core dumps, and checkpoints (these all included kernel information)
- The debugger processed symbol information from compilers to annotate its displays and for symbolic browsing (compiler always produced symbols)
- The debugging window displayed 20 consecutive words in a variety of selectable formats
- The fast interactive environment supported debugging one bug at a time

History - Where we're going

- In 1988 we began to change to Unicos, to buy commercial off the shelf systems rather than continue the development of our own system
- The pluses - We got better access to Cray compilers, to standard network products, and use of graphical workstations
- The minuses - Users got an unfamiliar, more complicated, less responsive environment and the debugging tools were a giant step backwards
- Too many tools have a legacy of teletype interfaces
- Users demand services and capabilities at least as good as they used to have

Currently

They use print statements!

Why?

Current Debugging Situation

- Users are conservative, they won't try dbx/cdbx if:
    - Special compiles are needed
    - It doesn't work on optimized code
    - It's takes too much time to learn
    - It breaks on some kinds of codes
- Instead they use print statements which:
    - Are easier to use and more flexible
    - Work for all languages and levels of optimization
- Lack of debugging tools has limited the use of multitasking to large production codes
- Only experimental use of parallel processors, but again there is a lack of debugging tools

may also be used in the context of a cast or in conjunction with the **typedef** storage class specifier to declare a new typedef name.

1.4.5 Intrinsic Functions

Intrinsic functions have access to the task record and consequently, access to the interpreter's semantic stack, the number of arguments, and their data types. The task record itself is of data type **union ae_TASK_REC**, which has the equivalent typedef name **ae_task_rec**. Note that arguments to intrinsic functions do not undergo promotions of any sort before the call and may be polymorphic in type, as may the return value of the intrinsic.

getrec() The intrinsic function **getrec()** returns a pointer to the task record of the interpreter executing the call:

ae_task_rec *getrec ()

The symbol table must have been scanned and a definition for **ae_task_rec** entered into ae's internal symbol tables in order to assign a type to the result of a call to **getrec()**.

symbolof() The **symbolof()** intrinsic function returns the internal symbol associated with the argument. The argument may be any data object, that is, any identifier except for a typedef name, struct/union/enum tag, or a **goto** label. The symbols used by ae have data type **struct ae_SYMBOL**, which has the equivalent typedef name **ae_symbol**.

ae_symbol *symbolof (<expression>)

The symbol table must have been scanned and a definition for **ae_symbol** entered into ae's internal symbol tables in order to assign a type to the result of a call to **symbolof()**.

cmai() The **cmai()** intrinsic function (*column major array index*) indexes its array argument in column major order, as opposed to C's row major order. In addition, nonzero lower dimension bounds are taken into account when performing the index calculations, whereas **e1[e2]** is evaluated by the

C interpreter in exactly the same manner as *(e1+e2).

<type> cmai (<array>, <expression>,...)

The arguments following the array argument must be integral in type, and their number must not exceed the number of dimensions in the array. A pointer is considered syntactically equivalent to an array with a lower dimension bound of 0 by **cmai()**. This routine was added mainly for extra compatibility with compiled Fortran code; it is not possible to declare arrays with a nonzero lower dimension bound in the interpreter without using the **typedec** declarator (See Subsection 1.4.4).

print() The **print()** intrinsic function pretty-prints each of its arguments by traversing the object and its corresponding type descriptor:

void print ([<file>,]<expression>,...)

If the first argument has type **FILE** *, the remaining arguments are printed to it. Otherwise, all of the arguments are printed to **stdout**. Strings appearing in the argument list must have data type **char** [] and not **char** * in order for a string to printed out instead of the pointer value.

1.4.6 Access to Local Compiler-Defined Data Objects

As mentioned in Subsection 1.1, one may reference a global variable or routine declared with the **static** storage class by the identifier

<filename>'<ident>

where **<filename>** is the name of the compilation unit[18] that the variable or routine **<ident>** is declared in. All characters in **<filename>** that cannot be used in a legal C identifier are replaced with underscores. Should the name begin with a digit, an underscore is prepended to the entire name.

If the user wishes to access such identifiers without the prefix **<filename>'**, he may insert the symbols for all of the routines and data objects declared local to the compilation unit **<filename>** into the current scope by

[18] The name of the source file passed as an argument to the compiler, not an included file name nor the name of the resulting object file.

calling the function **ae_load_static_scope()**:[19]

```
void ae_load_static_scope (rec, comp_unit)
    ae_task_rec *rec;
    ae_symbol *comp_unit;
```

where

rec is the task record for the current invocation of the ae interpreter.

comp_unit is the symbol for the compilation unit **<filename>**.

For example, if the interpreter encounters the following code fragment when invoked from the program described in Subsection 1.1,

```
ae_load_static_scope (ae_getrec (),
    ae_lookup_symbol ("test.c", ae_static_file_hash));
```

then the user may reference the variable **static_global** sans the prefix **test_c'**. **ae_lookup_symbol()** is described in Subsection 1.4.7.

The user may insert symbols for variables local to a routine (for a given stack frame) by calling the library routine **ae_load_dynamic_scope()**:

```
void ae_load_dyn_scope (rec, routine, block, fp, ap)
    ae_task_rec *rec;
    ae_symbol *routine;
    int block;
    char *fp;
    char *ap;
```

where

rec is the task record for the current invocation of the ae interpreter. Task records are described in Subsection 1.4.5.

routine is the symbol for the routine whose local identifiers are being loaded. A Fortran main program usually has a corresponding identifier called **MAIN** or **MAIN_**.

[19] When using versions of the application executive that have not been compiled entirely with symbolic debugging information (see Section 2), the user may manually insert the symbols for a library routine, or alternatively call it using the ld symbol table entry (see Subsection 1.2.

block is the block number. To load the outermost block in **routine**, **block** should be zero.

fp is the frame pointer for the stack frame being loaded.

ap is the argument pointer for the stack frame being loaded. On machines where the addresses of arguments are given as an offset from the frame pointer instead of the argument pointer (i.e., all machines that ae has been ported to so far), one should pass the frame pointer as the last two arguments to **ae_load_dyn_scope()**.

Obviously, user must have some way of extracting the frame pointer and argument pointer (if the argument pointer is needed) for the stack frame being loaded. This can be accomplished through a routine in an auxiliary debugging library, source code instrumentation, or use of a debugging utility such as dbx or gdb.

1.4.7 Other Commonly Called Library Routines

In addition to calling library routines to control ae's scoping mechanism and calling the various intrinsic functions, the user may find the following library routines useful. A symbol table entry must exist for the routine to be called; this may be accomplished in any of three ways:

1. A symbol for the routine is manually entered into ae's symbol tables, as described in Subsection 1.3.
2. A version of ae that has itself been compiled with symbolic debugging information is used (see Section 2), and the stab scanner is invoked to create the symbol table entry for the routine.
3. The routine is called using the ld symbol table entries created by the stab scanner. See Subsection 1.2.

A brief description of the more commonly called library routines follows.

ae_init() When the interpreter or stab scanner is invoked for the first time, ae's internal statically allocated symbol tables and type descriptors are initialized. Should the user wish this initialization to occur without invoking either the stab scanner or the interpreter, he may call **ae_init()**:

```
void ae_init ()
```

ae_lookup_symbol() Most symbol table entries may be accessed through the interpreter by calling the intrinsic function **ae_symbolof()** as described in Subsection 1.4.5. This mechanism works only for identifiers representing data objects and program units, not for type names. Should the user wish to access the symbolic representation of a struct/union/enum tag or a typedef name, **ae_lookup_symbol()** may be called to search the appropriate table:

```
ae_symbol *ae_lookup_symbol (name, table)
    char *name;
    ae_symbol_table *table;
```

where **name** is the name of the symbol, and **table** is the table to search. A pointer to the symbol is returned, or null if there is no symbol in **table** by the appropriate name. The following symbol tables are statically allocated by ae:

ae_static_tag_hash contains struct/union/enum tags.

ae_static_file_hash contains the symbols for source files and header files.

ae_static_location_hash contains the linker symbol table entries.

ae_static_ident_hash contains other identifier symbols, including typedef names.

The task record for a given invocation of the interpreter contains the symbol tables for dynamically-allocated identifiers; the user should examine the source code documentation for information regarding their access.

ae_remove_symbol() If the user wishes to remove an identifier from ae's internal symbol tables, he may do so by calling **ae_remove_symbol()**:

```
ae_symbol *ae_remove_symbol (symbol)
    ae_symbol *symbol;
```

Each symbol contains a pointer to the table in which it is inserted, so it is not necessary to specify this information. A pointer to the symbol in question is usually extracted using the **symbolof()** intrinsic function described in Subsection 1.4.5. Tag names may be removed by using the **typeof** construct to extract the type descriptor, which contains a pointer to the symbol table entry for the tag. To remove a typedef name, the user must manually look up the symbol in ae's internal tables (see below). A pointer to the symbol is returned.

ae_fprintf() **ae_fprintf()** is identical to the C library function **fprintf()**:

```
void *ae_fprintf (file, format, ...)
    FILE *file;
    char *format;
```

except that ae's internal data objects may be pretty printed by specifying different descriptors in the format string. In addition to the usual **%d**, **%f**, **%x**, etc., **%S** may be specified to pretty print a symbol (the argument should be of type **ae_symbol ***) and **%T** may be specified to pretty print a type descriptor (the argument should be of type **ae_type ***). Other internal data objects may be printed with different descriptors; the interested user should consult the source code documentation.

1.4.8 Error Recovery

Errors encountered by the ae interpreter and stab scanner fall into four classes:

errors The interpreter encountered erroneous C source code which caused it to issue a message. The interpreter continues uninterrupted. Such errors often occur in declarations, where the identifier in question is discarded.

stmt errors The interpreter was unable to parse a statement correctly. It resets itself, and discards remaining input tokens until a semicolon[20] is encountered. Syntax errors are stmt errors. A diagnostic message is then printed concerning the state of the parser. Diagnostic messages may be turned off by setting the flag **ae_silent** to a nonzero value. See Subsection 1.5.

block errors The interpreter encountered an error which made it unable to parse the remainder of the current block correctly. It resets itself and discards input tokens until an unmatched "}" is encountered. A block error may occur when a formal parameter to the interpreted routine is not declared. A diagnostic message is then printed concerning the state of the parser.

[20] Should a left bracket appear in the input before the semicolon, all text, including semicolons, is discarded until the matching right bracket is encountered.

fatal errors An internal check for a condition necessary for the interpreter to function properly has failed. Versions of **libae.a**, or a variant thereof, that have been compiled with the **-DAE_DEBUG** flag to cc contain extensive checking for fatal errors. The default action when a fatal error is encountered is the terminate program execution. If the flag **ae_no_error_exit** is set (See Subsection 1.5), then the interpreter returns to its caller instead of calling **exit()**. The error message issued often displays faulty data structures in detail, and therefore can be quite lengthy.

The error handling mechanism used to reset the interpreter deserves special mention. The state of the parser is stored before each statement and block. This state includes the indices into the various stacks used by the LALR(1) parser. The grammar was carefully written to insure that should such an error subsequently occur, the stacks would never have shrunk past the point where its index was saved. Therefore, error recovery is a matter of jumping back to the appropriate routine (using the UNIX setjmp/longjmp mechanism), restoring the stacks to their previous state by resetting their indices, and discarding input tokens until a recognizable construct is found (a ; or an unmatched "}").

If the flag **ae_save_stmt_err_jmp** is set, the interpreter will save its current state in the global variable **ae_stmt_err_jmp**[21] before each statement is parsed. The user may use this to return control to the interpreter should an error occur. If the interpreter is invoked in parallel and **ae_save_stmt_err_jmp** is set, then there exists a race condition for writing **ae_stmt_err_jmp**.

1.4.9 Parallelism

The interpreter can be invoked by concurrently executing threads of control and no corruption of the symbol tables will occur, so long as certain restrictions are observed. The first declaration for any statically allocated data object should not be encountered simultaneously by two or more threads. Once the initial declaration for an object has been parsed, the data object may be referenced by concurrent invocations of the interpreter. Secondly, if an incomplete **struct** or **union** type is declared, it should not later achieve completeness through a type definition parsed simultaneously by two different threads.

[21] declared as **ae_error_jmp *ae_stmt_err_jmp**

In other words, one must use the appropriate synchronization in order to eliminate all possible race conditions for the creation symbol itself, although nondeterminacy may still exist when modifying to the actual data object the symbol represents. Once a data object has been declared, subsequent declarations for it may occur in parallel (so long as they do not complete a previously incomplete **struct** or **union** type).

The machine's native synchronization mechanism may be used so long as it accessed through library routines. On Alliant machines, synchronization consists of calling the routines **lock()** and **unlock()**.[22] If the synchronization mechanism is accessed through compiler intrinsics, the user may be able to write routines on top of them that perform the same function.

Simultaneous invocations of the interpreter may take the input text stream from a string using **sae()**. If the input stream comes from outside the process, special mechanisms[23] may be needed to keep the entire process from blocking when only one invocation of the interpreter is waiting for input.[24] This is discussed in Section 3. Once this is done, however, it is possible to synchronize the different invocations of the interpreter through synchronization of their input streams.

1.5 Flags

Several flags may be changed by the user in order to customize the behavior of the interpreter. Storage for them is allocated in the ae library; the user should declare them as **extern** in his own code.

ae_silent (default 0) Diagnostic messages are suppressed when **ae_silent** is nonzero.

ae_ntrp_const_addresses (default 0) This flag is usually set (to a nonzero value) when calling Fortran routines through the interpreter. This allows the application of the address operator (**&**) to a constant, so that it may be passed by reference without creating a temporary variable to hold the constant's value. For example, if **ffunc** is the name of

[22] Since they exist in a library not compiled with symbolic debugging information, the routines must be called with their corresponding ld symbol table entries _lock and _unlock, or declarations for them manually inserted in ae's internal symbol tables as described in Subsection 1.3.

[23] which may require modification of the way that the ae interpreter reads its input stream

[24] Additionally, most parallel UNIX operating systems require that all i/o take place within a critical section of code.

a Fortran subroutine compiled with symbolic debugging information that takes one argument of type **integer**, **ae_stab()** has successfully scanned the symbol table, and **ae_ntrp_const_addresses** is set, then then the code fragment

```
{ ffunc (&5); }
```

passes a constant with the value 5 as a reference parameter to **ffunc()**.

ae_ntrp_nonstd_addresses (default 1) When this flag is set (nonzero), then the address operator (**&**) is not ignored if it appears before an array or function. This is useful when displaying type descriptors:

```
int a[10];
{ ae_fprintf (ae_ef, "%T", typeof (&a)); }
```

If **ae_ntrp_nonstd_addresses** is zero, then the **&** operator is ignored, and a type descriptor for an array of ten integers is displayed. If the flag is set, the **&** operator is applied to the array, and a type descriptor for a pointer to an array of 10 integers is displayed.

ae_ntrp_no_error_exit (default 0) If this flag is set, then the interpreter or stab scanner will return to the caller instead of exiting when a fatal error is encountered. See Subsection 1.4.8.

ae_print_brief_types (default 1) When this flag is set, **ae_fprintf()** (See Subsection 1.4.7) prints type descriptors in less detail. When the flag is zero. the often quite lengthy and unenlightening lists of struct/union members and enum constants are displayed in full detail.

ae_save_stmt_err_jmp (default 0) If **ae_save_stmt_err_jmp** is set, then the interpreter will save a pointer to its current state in the global variable **ae_stmt_err_jmp** before each statement is parsed. See Subsection 1.4.8.

1.6 Bugs

The argument to the **sizeof** operator is fully evaluated. Not really a bug since it is a non-standard construct, the argument to the **typeof** operator is also fully evaluated. This will change with the correction of the bug concerning the **sizeof** operator.

The size and exact configuration of a struct defined by the interpreter may erroneously differ from the one defined by the host machine's C compiler. One may avoid conflicts by declaring new objects using the type name defined when scanning the executable's symbol table.

New data objects defined by the interpreter may not contain an initializer in the declaration statement.

Although function prototypes (new to the ANSI standard) are accepted by the interpreter, the arguments are not type checked or properly promoted when a function declared in such a manner is called. Instead, the default promotions take place. See Section A7.3.2 in Kernighan & Ritchie [KR88]. On many machines, no distinction is made in the executable's symbol table between functions declared with new style and old style parameter lists.

The stab scanner does not create location symbols corresponding to linker symbols which are private to a compilation unit (See Subsection 1.2). Symbolic information for a routine or statically allocated variable private to a compilation unit that has not been compiled with symbolic debugging information must be manually entered using **sae_declare()** or **fre_declare()**.

2 Installation

There are four basic versions of the application executive library that can be built. Each of these may be compiled entirely with symbolic debugging information (i.e., using the **-g** option), or with only enough files compiled with **-g** to insure that the type information necessary for the interpreter to operate correctly[25] exists in the symbol table of the executable. For the latter versions (that include the stab scanner), the stab scanner will operate more quickly and use less space, as there is less information to translate to ae's symbol tables. These versions are appended with the suffix "**_g.a**", as opposed to just "**.a**".

libae.a or **libae_g.a** These libraries contain the application executive in its entirety: the C interpreter, stab scanner, both of which rely on ae's internal symbol tables.

libae_ntrp.a or **libae_ntrp_g.a** Only the interpreter and internal symbol tables are included in these versions. The stab scanner is omitted.

libae_stab.a or **libae_stab_g.a** Only the stab scanner and internal symbol tables are included in these versions. The interpreter is omitted

libae_sym.a or **libae_sym.g.a** Only ae's symbol table management is included in these versions - no interpreter or stab scanner.

It is perfectly reasonable to ask the question, "If the linker only includes the needed object files from a library, why does one need four different versions?" The motivation behind this was portability. Since the Vista project [TCJ+91, JT91, TJC91a, TJC91b] relied on ae's internal symbol tables, and although the stab scanner and interpreter can be invoked, they were not imperative to the operation of Vista, the decision was made to create a version of ae (**libae_sym.a**) that could be ported with the least amount of time and effort. This brings up the question, "Why not split ae into three different libraries, the interpreter, the stab scanner, and a library of routines to access ae's internal symbol tables?" The task record passed throughout the calling sequence[26] differs for the interpreter and the stab

[25] The interpreter must find the type descriptors for **ae_type**, **ae_symbol**, and **ae_task_rec** in order to assign a type to the result of a **typeof** expression, to check the type of an argument to the typedec declarator, or to assign a type to the result of a call to **ae_symbolof()** or **ae_getrec()**.

[26] Actually, a pointer to the task record is passed.

scanner, with each version being declared as the variant of a union. The symbol table management routines access fields common to both variants. When compiling the library to include only the interpreter or the stab scanner, we would not want to include the variant used by the other. Hence the declaration for **union ae_TASK_REC** differs between **libae_ntrp.a** and **libae_stab.a** (and **libae.a** and **libae_sym.a**). Linking with routines from more than one version of the library results in multiple declarations of **union ae_TASK_REC**. The stab scanner will then encounter inconsistent type information. See Subsection 1.2 for information on how the stab scanner resolves inconsistent type information.

To compile a given version of the application executive library for a given architecture, the installer should **cd** to the top level ae source directory and enter the command:

make -f Makefile.<arch> <lib>

where

arch is one of

SPARC A Sparc workstation, made by Sun Microsystems. Use this version with the supplied C compiler.

SPARC_GCC A Sparc workstation. Use this version with the GNU C compiler *gcc*.

ALLIANT_FX An Alliant FX/1, Alliant FX/8, or Alliant FX/80. If porting to a new architecture, one may wish to start by using this version, as is the most compatible with the portable c compiler *pcc*.

ALLIANT_FX2800 An Alliant FX/2800 series computer.

lib is one of the aforementioned library versions.

Building the interpreter or stab scanner requires the use of the parser generator *bison* [DS88], and the lexical analyzer for the interpreter is created using the *flex* utility [Law90], version 1.3 or later. Earlier versions of flex are incompatible, as the application executive contains its own version of the *skeleton* lexical analyzer "flex.skel". See the instructions for using the -S option to flex and the ae source code for more information.

3 Debugger Configurations

The application executive itself cannot be considered a debugger per se; more functionality is needed, namely the ability to show a stack trace, set breakpoints, and trap exceptions. These features are quite machine-specific, whereas ae is more portable, at least among UNIX/C platforms.

3.1 Sequential Debugger

Since ae exists as a library, in the simplest configurations the debugger exists entirely within the same process and address space as the application.

Writing a signal handler is a relatively straightforward process. An initialization procedure tells the operating system which routines, or signal handlers, to call when certain exceptions occur. The application executes, and for the sake of argument, assume that an exception occurs. Control is transferred back to the operating system, which directly invokes the signal handler (either using the current run time stack or a special *signal stack*.) The signal handler needs to do little more than print an error message concerning the nature of the error ("bus error" or "arithmetic exception", for instance), and invoke the ae interpreter. The user may then interactively examine and modify the state of the program in the usual sense of a breakpoint debugger.

Doing this often involves calling several compiled debugging routines, the first of which is usually a stack trace utility. A stack trace must be able to follow the chain of frame pointers, and print the frame pointer (fp), argument pointer (ap), and the return address / program counter (pc) for each frame. ae's internal symbol tables must then be searched for the routine that contains the executable code at a given address. This information should be displayed in a readable format that includes the block number in question for each routine.

Once the user has determined where in the course of execution the exception occurred, he will usually wish to examine various data objects to gain more information regarding the nature of the error. If these data objects exist locally to a routine, he may use the stack trace information to load their symbolic representation into ae's internal symbol tables by calling **ae_load_dyn_scope()** (see Subsection 1.4.7). Alternatively, this process can be automated by integrating it with the stack trace utility.

The initialization routine, signal handler(s), and debugging routines may exist in a library compiled with the user's code along with the ae library.

The only modification necessary to the user's code is to call an initialization routine which sets the signal handlers and scans the symbol table of the executable using **ae_stab()**.

Figure 1 shows the stack configuration after an exception has occurred and the ae interpreter invoked. Before each statement is parsed, the interpreter saves its current state in the global variable **ae_stmt_err_jmp**.[27] Should an exception occur in the interpreter while evaluating an expression or in debugging routines called by the interpreter, the signal handler can determine that the interpreter was already invoked and transfer control back to it.

Should the user wish to set breakpoints, he may either do so by calling the signal handler directly in his code before compilation, or dynamic breakpoints may be placed within the instruction stream at run time. The latter requires that the executable code be placed in a writable address space. This can be accomplished with command line options to the UNIX linker, *ld*. The instructions at the appropriate address are saved, and then overwritten[28] with an opcode which raises a signal when executed. A breakpoint is then handled in the same manner as any other exception. When the user wishes to return from the breakpoint, the opcodes at the correct address are restored, and we exit the signal handler. When the signal handler returns, it restores the registers necessary to continue execution from that point.

This procedure, of course, clears the breakpoint. In order to leave a breakpoint in effect, another breakpoint may be set at the next instruction. This breakpoint automatically resets the first breakpoint and clears itself in the process of returning.

Interesting variants of this implementation include overwriting the executable code with jump statements to the signal handler, as opposed to an opcode that raises an exception, thereby avoiding operating system intervention. To return from a breakpoint without clearing it, one may append the stored opcode with a jump statement back to the instruction following the breakpoint, jump to the stored operands (they must exist in a segment with execute permission), execute them, then jump back to the instruction following the breakpoint. This procedure is highly dependent upon opcode sizes, but has the advantage that it does not inhibit parallelism; the exe-

[27] using the **setjmp/longjmp** mechanism, with the flag **ae_save_stmt_err_jmp** set See Subsection 1.4.8.

[28] in the memory image, not in the file loaded into memory by the operating system

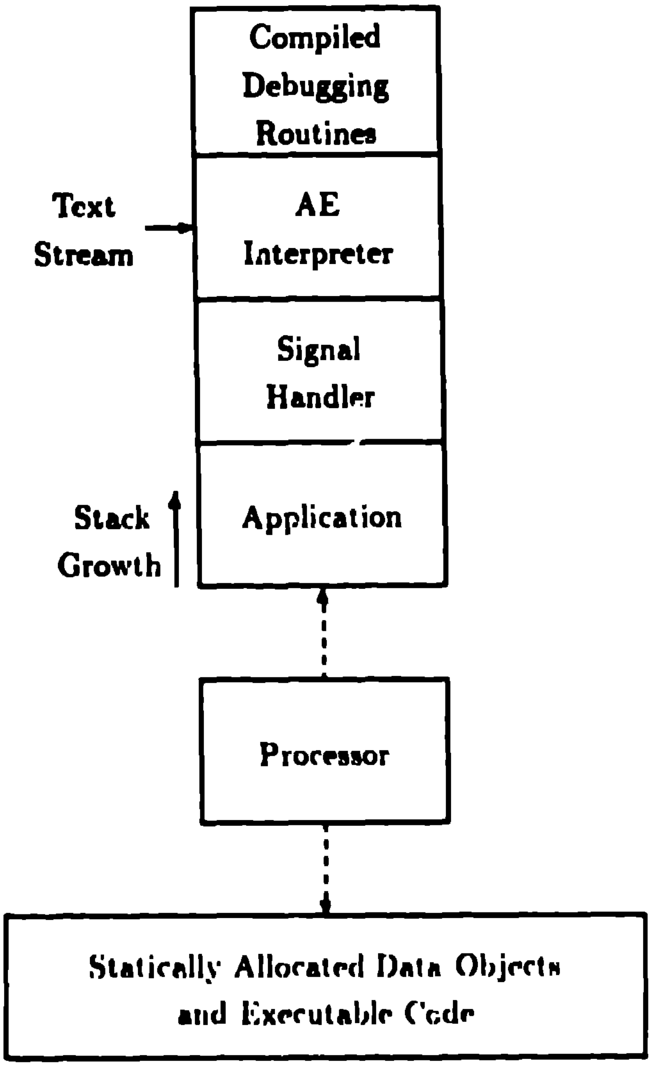

Compiled Debugging Routines
Text Stream
AE Interpreter
Signal Handler
Stack Growth
Application
Processor
Statically Allocated Data Objects and Executable Code

Figure 1: A sequential single process debugger configuration. Alternatively, the signal handler and ae interpreter may execute on a separate stack.

cutable memory image is not modified in the process of returning from the breakpoint.

3.2 Parallel Shared Memory Debugger

In order to debug a parallel program on a fully shared memory machine such at the Alliant FX/8, very few modifications are required to the sequential version.

What is required is operating system support. When an exception occurs, it is only necessary that one processor enter the signal handler and interpreter so long as it has access to the stack frames of the other concurrently executing processors. The operating system must store the values of the registers in each processor, and allow the debugger access to these. It should inform us which processor actually caused the exception; this may or may not be the processor that actually executes the signal handler. If the threads are multitasked, that is, an arbitrary number of threads of control are executed by and switched between the available processors, the operating system should also allow access to the stored values of the registers for the suspended threads.

After entering the interpreter, the user will typically call the stack trace routine. The stack trace must show the status of all threads of control. The user may then examine data objects local to a given stack frame, for any thread of control, by calling the routine **ae_load_dyn_scope()** to load the symbols for the given frame into ae's internal symbol tables. (**ae_load_dyn_scope()** is described in Subsection 1.4.7.) Even if the stack frame in question was created by a different processor from that currently executing the interpreter, it still exists in a memory region accessible by the interpreter, so there is no need to use more than one processor when debugging. If the threads are multitasked, the operating system should allow the user to change the scheduling status of a given thread.

3.3 Parallel Private Stack Memory Debugger

Should the processor that executes the interpreter following an exception not have access to the stack frames of the other processors, all processors may execute the interpreter, with an arbitration scheme multiplexing the input code stream between the separate invocations. Most all versions of the UNIX operating in use today that support parallel programming allow only one processor to execute the signal handler; it is necessary to "trick"

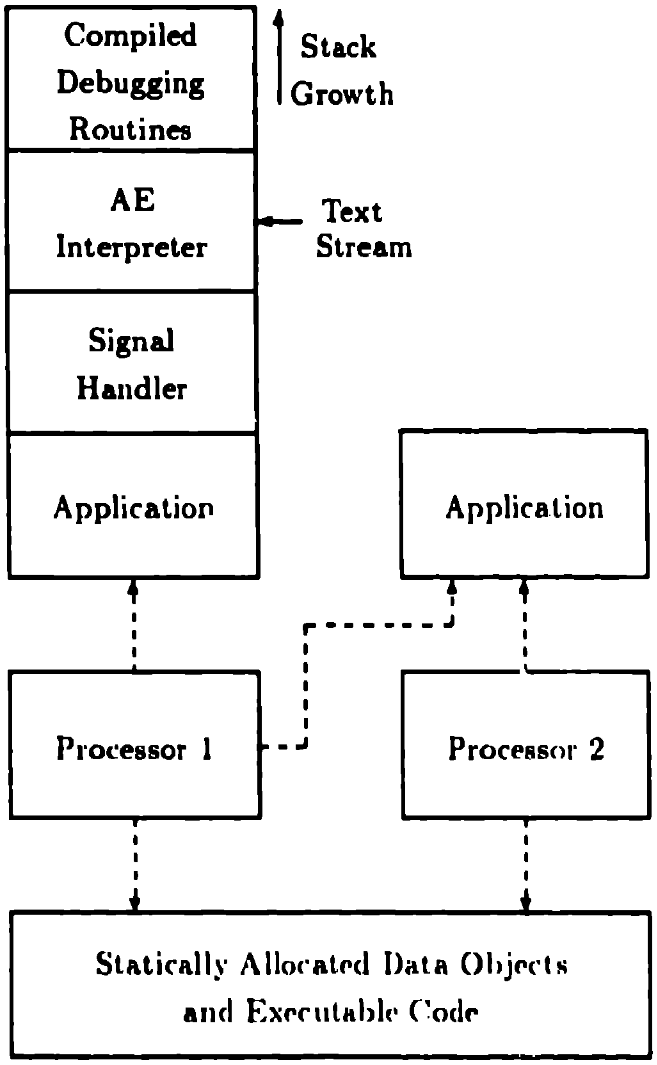


Figure 2: A parallel shared memory debugger configuration. Alternatively, the signal handler and ae interpreter may execute on a separate stack.

the others into executing the interpreter by modifying their stored states, causing all processors to jump to the handler immediately after the single processor executing it returns and parallel execution resumes.

An alternative to this scheme is to have a single processor execute the interpreter. The others execute a data access mechanism communicating with the interpreter, copying data objects to a region of memory where the interpreter may access it. The application executive does not currently support this.

The input stream arbitration must not allow the entire process to block unless all invocations of the interpreter are currently waiting on input. This would normally dictate that the arbitration exist within the process itself. Figure 3 shows the debugger configuration for a parallel machine where processors are denied access to each other's stacks. This architecture inhibits true multitasking (without extensive copying of stack memory). Here, there usually exists a one-to-one correspondence between the processors and stacks for a given process.

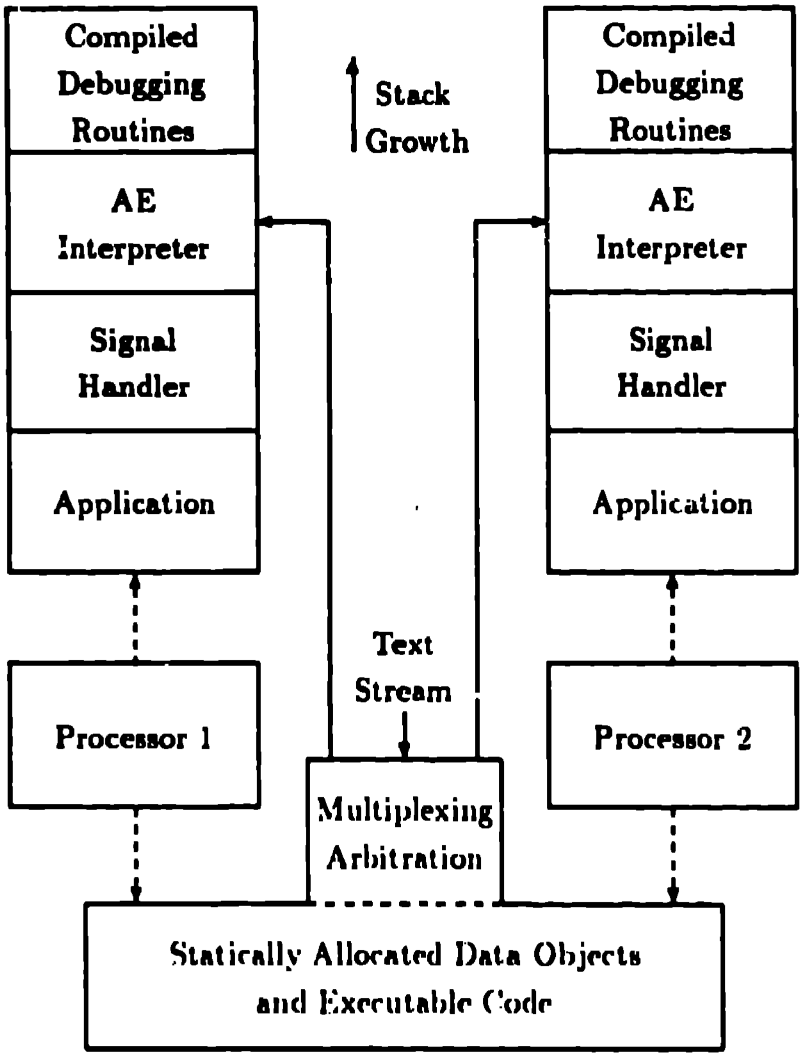


Figure 3: A parallel private stack memory debugger configuration. Because a processor may not access the stack of another processor all processors execute the interpreter.

4 Conclusion

The application executive is a versatile tool that allows the user to control the execution of his program at run-time without recompilation; because this is a major requirement for a debugger, it also forms the basis for a number of debugging configurations for widely varying computer architectures, both sequential and parallel.

The application executive is useful in any context where a general locus of control is desired within an executing program, especially when the user can be assumed knowledgable of the C language and possibly the UNIX operating system. The use of ae greatly facilitated the debugging of the Vista project, in addition to being an integral part of it.

Unless otherwise noted, all features of ae described in Section 1 have been implemented and tested on an Alliant FX/2800; should any of them fail to perform as described, it is considered a bug.

Thanks to Allan Tuchman for his input during the development of the application executive, for critiquing this document, and last but not least, for coining the name "application executive".

References

[DS88] Charles Donnelly and Richard M. Stallman. *Bison*. Free Software Foundation, October 1988.

[JT91] David Jablonowski and Allan Tuchman. Vista users manual. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Superco mputing Res. & Dev., May 1991. CSRD Report No. 1068.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Law90] Lawrence Berkeley Laboratory and Vern Paxson. *flex*, May 1990.

[Lin90] Mark A. Linton. The Evolution of Dbx. In *Usenix Summer Conference Proceedings*, pages 211–220, June 1990.

[Sta89a] Richard M. Stallman. *GDB Manual*. Free Software Foundation, October 1989.

[Sta89b] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, April 1989.

[TCJ$^+$91] Allan Tuchman, George Cybenko, David Jablonowski, Bri an Bliss, and Sanjay Sharma. Vista: A system for remote data visualization. *Presented at the Fifth SIAM Conference on Parallel Processing for Scientific Computing, Houston, TX*, March 25-27, 1991.

[TJC'91a] Allan Tuchman, David Jablonowski, and George Cybenko. A system for remote data visualization. Technical report, Univ. of Illinois at Urbana-Champaign Center for Supercom puting Res. & Dev., June 1991. CSRD Report No. 1067.

[TJC'91b] Allan Tuchman, David Jablonowski, and George Cybenko. Run-time visualization of program data. *To appear in the Proceedings of Visualization '91, San Diego, CA*, October 22-25, 1991.

[Uni87a] University of California, Berkeley. *CC*, 4.3 bsd edition, June 1987. in Unix User's Manual Reference Guide.

[Uni87b] University of California, Berkeley. *DBX*, 4.3 bsd edition, June 1987. in Unix User's Manual Reference Guide.

Here is my list of features for my dream debugger of the future:

Fast conditional breakpoints and fast memory watch points.

Bounds checking.

A memory map in a user friendly format.

2 and 3d plots of multidimensional data simply e.g. plot A

reverse execution

Why is x=3? Dynamic backward chaining of dependences.

Dynamic linking of new procedures (useful in debugging long executions).

A memory reference trace of shared variables.

A trace of message passing activity (in nice graphic format of course:).

Structured display of user data (lists etc. - go SGI).

Race detection.

Dynamic display of data (i.e. continuously updated).

Performance displays.

Deterministic replay.

Multiprocess event detection (e.g. stop when p1 is at x and p2 is at y).

Source level debugging of optimized/parallelized code.

Algorithm level debugging (e.g. better abstraction to higher levels).

Debug aids for numerical problems (e.g. instability due to limited precision).

Range checking of arbitrary values (not just array indicies).

Compare intermediate states of new and old versions of a program during execution.

Charlie McDowell, ***University of California at Santa Cruz***

3.1 Automated Analysis

In parallel programs, the most common and complex errors involve the order of events. Behavioral abstraction simplifies matters. It is intended to aid the user in reasoning about relationships with respect to time. The most fundamental of these relationships is *happened-before* [28] and is denoted by an arrow (->).

Figure 3 shows a partial order diagram for a two-process application. The arc between the two timelines corresponds to a message transmission. Events A and C represent the sending and receiving of the message, respectively. Since a message must be sent before it can be received, event A must have occurred before event C. This fact can be represented as A -> C, or "A happened-before C." This relationship also exists for each pair of events on the same timeline. Thus, A -> B and C -> D. Because the happened-before relationship is transitive, we can also say that A -> D. However, events B and D are on separate timelines with no arcs between them. Since no happened-before relation exists between these events, we simply say they are unordered, or *concurrent*. This approach results in an event sequence that is said to be partially ordered because not all event pairs are ordered.

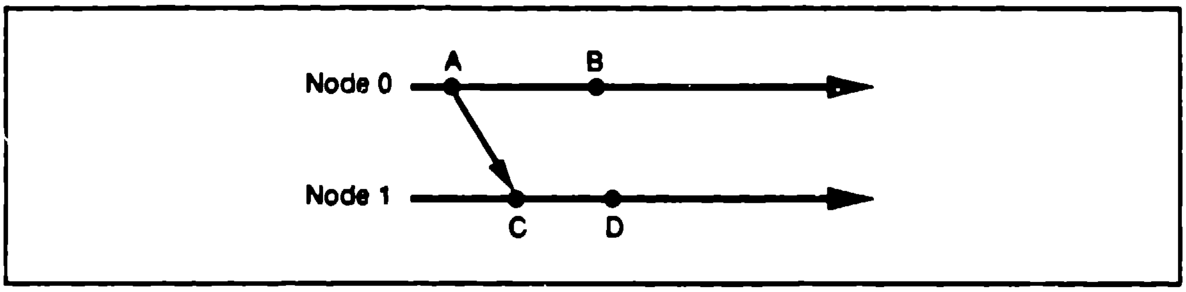


Figure 3. Partial Order Diagram

An alternative approach used by many behavioral abstraction systems is to order each pair of events. In a system based on total, or linear, ordering, the timeline in figure 3 might be described as A->B->C->D. Note that this implies a relationship between events B and D that does not actually exist.

A behavioral abstraction system based on partial ordering yields only those relationships that are actually observed. Furthermore, during the recording phase, the execution replay system extracts information that is essentially a partial order diagram, and can be used directly to guide analysis.

To date, the most complete and practical collection of partial order operators are those developed by Hough [22]. These are *precedes* (->), *precedes closure* (*), *parallels* (||), and *parallels closure* (||*). Precedes (->) is similar to the happened-before relation. Precedes closure (*) is used to describe any number of consecutive instances of an event.

Parallels (||) is similar to the concurrent relation. Parallels closure (||*) is used to describe any number of unordered instances of an event.

These temporal operators are used to model program behaviors. For example, imagine a distributed simulation that advances lock step in a series of rounds. Imagine, also, that each round consists of five phases. Four correspond to communication with north, south, east, and west neighbors, while the fifth involves the computation for that round. Figure 4 shows this sequence as described using Hough's operators. Abstract events A, B, C, and D have been defined previously as the four communication events, and E as the computation step. The parallels closure operator describes any number of events in parallel. In figure 4, (A||*) denotes any number of northbound message exchanges. Consequently, the abstract event X is a model of the entire execution round.

```
X is (A||*) -> (B||*) -> (C||*) -> (D||*) -> (E||*)
Y is X*
```

Figure 4. Execution History Consisting of a Series of Five-Phase Rounds

Using the rules in figure 4, the debugging environment can now be told to track each X, or round of execution, for us. Alternatively, the debugger can be used to verify assertions about program behavior. With this approach, the programmer first uses modeling to define a set of conditions that must hold throughout the execution. Next, the program is run and monitored by the debugger. Any violations of the assertions trigger predefined actions such as breakpoints or notifications. For example, the entire simulation, Y, is defined as a series of rounds in figure 4. This permits any discrepancy between the expected behavior Y and the actual execution to be detected automatically.

To Hough's temporal operators, PARADIGM adds the concept of a *sliding interval*. This mechanism enables the analysis facilities to obtain and reason about performance-related information. A sliding interval's leading edge always coincides with the current point in the execution. Its trailing edge falls some number of events behind. Since abstract events may correspond to phases, or even rounds, a sliding interval can span a substantial sample of the execution.

As an example of the use of sliding intervals, consider the distributed simulation once again, and imagine that load balancing is being added. This requires that the simulation first be broken into granules. Each of these hosts some number of simulation objects. The granules are then distributed so that each processor manages roughly the same number of simulation objects. An imbalance occurs if simulation objects later migrate and leave the granules on some processors depleted, while crowding others. Balance can be restored by swapping granules so that each processor again manages roughly the same number of simulation objects.

To tune the simulation, the programmer must weigh the degree of balance achieved against the overhead incurred. The key parameter is granule size. Decreasing granularity improves load balance, but also results in more message exchanges during each round. Determining the appropriate granule size may require careful observation over several runs. Sliding intervals can be used to facilitate this type of performance assessment.

To determine optimum granule size, the user must observe both the uniformity of the object distribution and the number of messages exchanged. A simple measure of uniformity can be obtained by comparing the number of simulation objects resident on the least and most populous processors. Load balancing overhead is directly indicated by message volume. During any single round of execution, however, either measure may be misleading. For example, an object redistribution causes an immediate spike in message traffic but may reduce overall execution time. Ideally, both measurements should be taken over the entire execution. Unfortunately, this could be a considerable length of time. Furthermore, post-mortem analysis precludes interactive study.

Sliding intervals permit interactive analysis while mitigating the inaccuracy of instantaneous measurements. In the load balancing example, a sliding interval may be defined over some number of execution rounds. The distribution delta and message overhead would be averaged across the entire interval. As the execution progresses, the interval slides forward in time. During any interval, an exceptionally large average could trigger a predefined action, such as a breakpoint. The user could then study the conditions that prompted the action. The programmer may control the accuracy and responsiveness obtained—conflicting goals that are determined by interval duration. Increasing interval length increases response time, but reduces the impact of anomalous measurements.

3.2 Automated Control

During replay, program events are represented as tuples. Changes made to the parameters of a tuple are reflected in the corresponding event. For example, consider the event tuple shown in figure 5, which represents a message send event. The tuple's class is *comm* and its type is *send*. An arbitrarily long list of attribute/value pairs may follow these fields. In the figure, the first attribute, *destination*, identifies the target of the message as process 0 on node 31. The next pair gives the message type. Finally, the message itself appears. (In the figure, NULL indicates a zero-byte message.) By changing the *message* attribute, a different message may be substituted. Similarly, a new target may be specified by modifying the *destination* field.

```
(comm send destination 31:0 type 4 message NULL)
```

Figure 5. A Send Event

During a breakpoint, the user may alter or even delete any event manually. Alternatively, the analysis system can modify and delete events on the fly. This ability permits extensive experimentation with the execution without the need for recompilation.

3.3 Reflective Queries

During execution, behavioral abstraction can be used to track the progress of an application automatically, to monitor its performance, and to detect any deviations from its expected behavior. Often, however, a programmer will need to interrogate the environment directly. For example, following a breakpoint, he may query the system to determine the sequence of events that preceded an erroneous behavior. This type of query is said to be *reflective*, since it refers to past events.

Reflective queries are made in the same notation as that used for behavioral abstraction rules. For example, the rules in figure 4 define the expected behavior of a hypothetical simulation. Presented as a query, they should match all the past behavior of the execution. If only a partial match is made, the point of deviation may indicate the origin of an error. If the match fails entirely, further queries can be used to unmask the error. In this way, a programmer can test assertions about his application. Failed assertions indicate discrepancies between expected and actual behavior.

3.4 Cellular Displays

The analysis system can be used to drive simple state-based displays. These consist of a matrix, or plane, of cells. Each cell represents a program component. The color, shade, texture, or icon of a given cell corresponds to some aspect of that component's state. Figure 6 presents a display that might result from the hypothetical simulation discussed previously. Here each cell represents a simulation process. Shading is used to indicate the number of simulation objects managed by a process. Darker shades indicate larger numbers of simulation objects. By consulting the display, the programmer can determine immediately that the object distribution is nonuniform. In addition, he can identify particularly aberrant processes for further study. In experiments involving similar displays, test subjects succeeded in identifying patterns in thousands of pieces of multi-dimensional data [29].

PARADIGM will offer predefined views of system state such as message traffic, memory consumption, and idle time. The programmer may also use the analysis system to drive application-specific displays. Lastly, cellular displays may be used in conjunction with reflective queries. In this case, a display's cells are highlighted if the components they represent satisfy the conditions of a query.

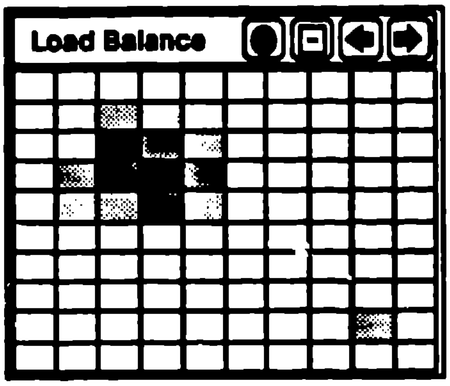


Figure 6. Cellular Display of Load Balance

4. REPLAY ANALYSIS

Execution replay and behavioral abstraction fit together naturally and have been combined previously. LeBlanc and Mellor-Crummey [30] describe an environment that offers both replay and LISP-based analysis of the execution history. In their system, replay can be used to drive conventional debugging environments. Analysis, however, is carried out separately from replay, and so cannot be used to dynamically control and alter execution.

Replay and analysis are closely coupled in the system described by Elshoff [13]. Although specific to the Amoeba operating system, this environment supports execution replay, behavioral abstraction, and source-level debugging. Monitoring information is collected by an instrumented library and sent to analysis tasks. This requires debugger and application messages to be interleaved. Consequently, the environment can perturb the execution, though not beyond the possibilities permitted within the partial ordering. In addition, monitoring data is maintained in the application's address space, where it can be modified by a faulty program. Finally, the environment cannot be used with conventional debuggers.

In PARADIGM, the combination of execution replay and behavioral abstraction is termed *replay analysis*. PARADIGM offers four principle extensions over previous works. These are: (1) examination, analysis, and modification of events that have yet to occur; (2) cooperative analytical strategies, including conventional debugging, graphic state mapping, and behavioral abstraction; (3) experimentation with event ordering in suspect program fragments; and (4) communication-related performance measurement. Each of these capabilities is discussed in the following sections.

4.1 Future Tense Query

Conventional source-level debuggers provide considerable information about the present. This includes such detailed information as the values of program variables and processor registers, and the currently executing statement in the application. Though less detailed, information about the past is also provided in the form of a stack frame history. However, no information about the future is available. The programmer must either infer future events or wait for them to be realized.

PARADIGM differs from conventional environments in its knowledge of future synchronization events. This information is captured during the recording phase and is made available at replay. The programmer is thus able to inspect and even modify future synchronization events. For example, following a breakpoint, a programmer can examine the next messages to be received. If he suspects that an error arises from their particular ordering, he may swap or even delete future message events before resuming execution.

4.2 Mixed Mode Debugging

Both PARADIGM's replay and analysis capabilities are transparently extended to conventional source-level debuggers. The replay facilities relieve traditional debuggers of the problems of the probe effect and intermittent errors. Similarly, behavioral abstraction can be used to augment conventional environments with event-based breakpoints. This analysis can also be used to drive the cellular display during execution or to support reflective queries following a conventional breakpoint.

For example, a programmer may use a conventional debugger to set a breakpoint on entry into a particular procedure. Following the breakpoint, he may consult cellular displays to obtain a global picture of the system state. These may identify particular components with suspect states. Next, he may make reflective queries of the analysis system to determine the sequence of events that preceded the breakpoint. He may also query the system for future events, possibly altering them. Finally, he can resume execution from the conventional debugger.

PARADIGM permits the programmer to use familiar, conventional debuggers more effectively. Alternatively, he may employ PARADIGM's analysis facilities alone. However, these two approaches can also be used cooperatively to achieve capabilities beyond either technique alone.

4.3 User-Directed Sequencing

In parallel programs, many errors are dependent on the sequence of events. PARADIGM enables the programmer to investigate the effects of alternative sequencing. Consequently, he may eliminate errors that have been observed or unmask those that have not.

User-directed sequencing requires that the program first be instrumented with labels visible to the debugger. Each of these serve as a possible synchronization point. The debugger can then be used to enforce mutual exclusion, barriers, or synchronization based on constraints provided by the user.

4.4 Performance Measurement

PARADIGM's analysis facilities permit measurement of elapsed physical time between any pair of synchronization events. For example, a programmer may determine the overall time required to send a request, await a consequent remote computation, and receive a reply. He may also track and time similar actions included in the remote computation.

These measurements are based on timestamps made during the recording phase. Consequently, they are immune to the time dilation and execution skew effects caused during replay.

5. Status

To date, much of the event interface and part of the monitor agent have been implemented on the Intel iPSC/2 multicomputer. These consist of a set of modifications woven into the operating system kernel that runs on each node of the iPSC/2. Currently, a wide variety of programs can be replayed. The analysis facilities are centralized, running on the iPSC/2's front end, and are still being developed. Thus far, they have been useful in providing performance measurements. The cellular display has not yet been implemented.

The kernel modifications result in three new operating system components. These are the event interface, monitor agent, and double agent. The relationship between these components is shown in figure 7.

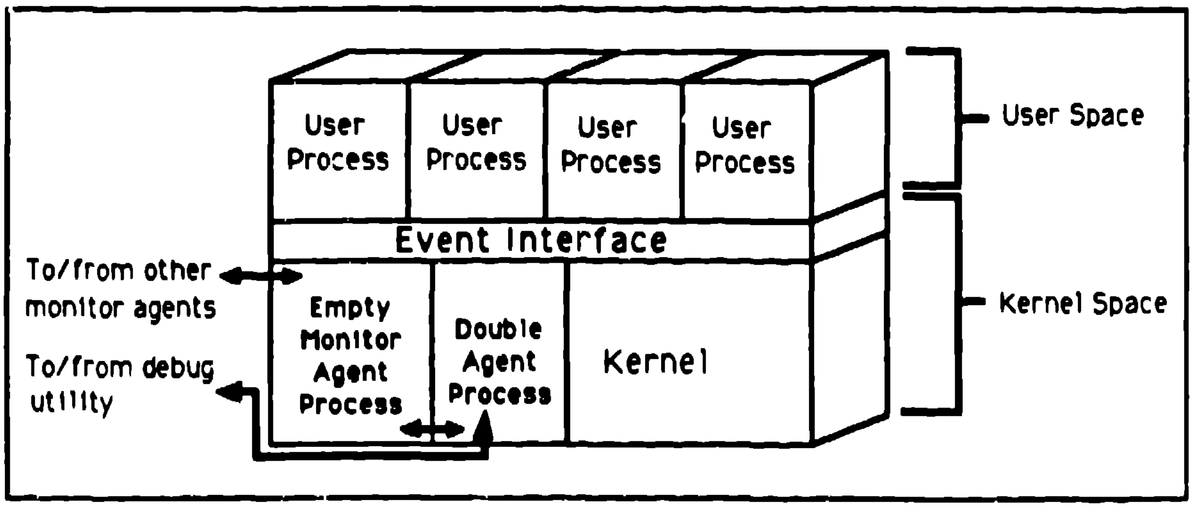


Figure 7. Relationship Between Node Level Debugging Components

Encapsulated in the event interface is all of the debugging environment's machine dependent code. During the recording phase, the event interface captures events which are ƿart of the partial order diagram needed for replay. This amounts to a small subset of events involving communication, and the impact per event is kept sufficiently low. For example, on the iPSC/2, the minimum time required to satisfy a receive request is 308 microseconds. The time required to timestamp a receive event is 21 microseconds. A further 18 microseconds are required to extract the event parameters and complete a receive record.

Event information is collected in a kernel buffer for later recovery, should debugging be required. Considerably less perturbation results from writing to a buffer than to a file. However, the buffer's limited size constrains the number of events, and hence the length of execution, which can be recorded.

The most communication intensive execution yet replayed involved an average of 382 recorded events per node per second. At this rate, event buffer space is exhausted after recording approximately 8 minutes and 32 seconds of execution. A method of extending recording time is presented in section 6.

The event interface cannot be manipulated directly. A kernel-resident process called the double agent serves as an intermediary. As shown in figure 8, the double agent can be commanded from a *debug* utility located anywhere on the network. From his remote session, a user can extract event buffer data using the save command.

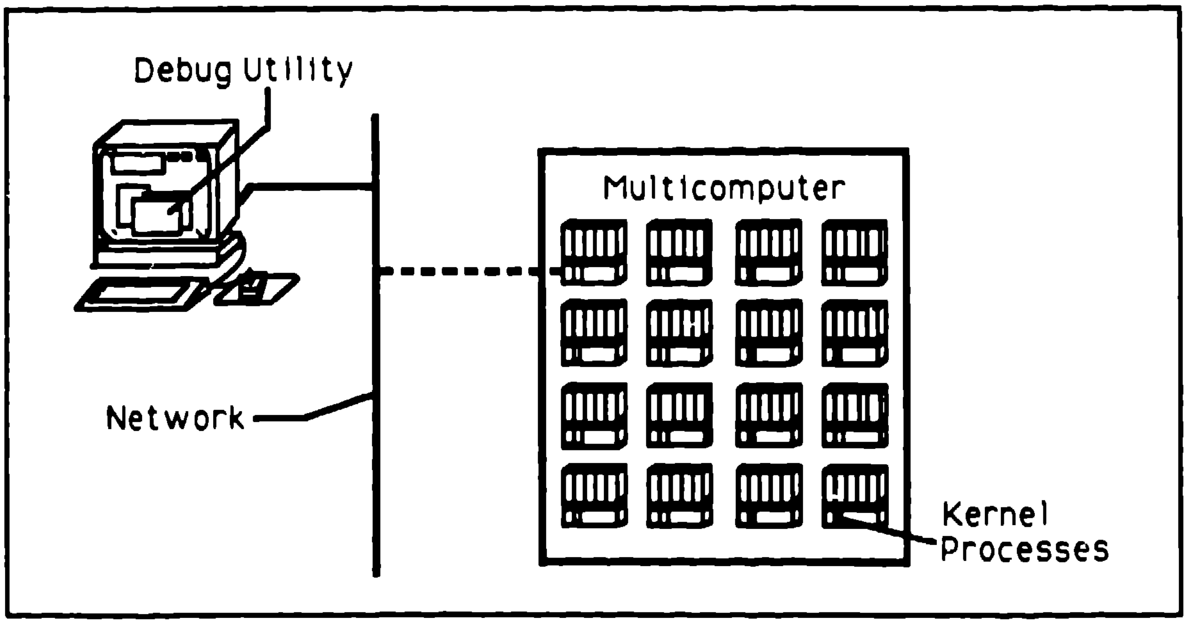


Figure 8. New Host-side and Node-side Software

As an example of replay, consider the sample program in figures 9 and 10. Figure 9 shows one of two processes comprising a simple test program. This "sender" process is loaded on nodes 1, 2, and 3 and results in three messages immediately being dispatched to node 0. A "receiver" process, shown in figure 10, is then loaded on node 0. The program completes when the receiver displays the order in which it consumes the three messages.

```
main()
{
   char buffer[10];

   csend(0, buffer, sizeof(buffer), 0, 0);
}
```

Figure 9. The Sender Process

```
main()
{
      char buffer[10];
      int  i;

      for (i=0; i<2; i++) {
            crecv(-1, buffer, sizeof(buffer));
            printf("Received message from node %d\n", infonode());
      }
}
```

Figure 10. The Receiver Process

Upon completion of the program, the debug utility is used to extract a *transcript* of critical events from the double agent on each node. The recorded critical events provide all the necessary information to reproduce the execution. The transcripts contain only individual event histories. They must be correlated to become the *schedules* which guide both execution replay and analysis. Schedule generation is performed by a mksched utility which is transparently invoked by the debug utility's `save` command.

The following are excerpts from a replay session. Here the user allocates four nodes, loads a sender process on three nodes, and then loads a receiver process on a fourth.

```
SRM> getcube
getcube successful: cube type 4ml6n0 allocated
SRM> load 1 sender
SRM> load 2 sender
SRM> load 3 sender
SRM> load 0 receiver
SRM> Message from node 1
Message from node 2
Message from node 3
```

The receiver process displays the sequence in which the three sender messages are consumed. This is the same as the loading order: node 0 followed by nodes 1 and 2. Next, transcripts are recovered and replay schedules generated using the debug utility.

```
SRM> debug save schedl
Node  0- Recovering transcript.
Node  1- Recovering transcript.
Node  2- Recovering transcript.
Node  3- Recovering transcript.

Building schedules
4 transcripts of 128 Kbytes

Node 0
------
 (getcube, physnode 0, @248849)
 (load, pid 0, @279339)
 (crecv, node 2, seq 18, #0, @280240->280240)
 (csendrecv, r_seq 182, peer 256, s_seq 25, #1, @280240->280244)
 (crecv, node 4, seq 11, #4, @280278->280278)
 (csendrecv, r_seq 185, peer 259, s_seq 28, #5, @280279->280302)
 (crecv, node 6, seq 11, #6, @280303->280303)
 (csendrecv, r_seq 186, peer 259, s_seq 29, #7, @280303->280326)
 (terminate, pid 0, @280363)

Node 1
------
 (getcube, physnode 2, @248858)
 (load, pid 0, @265927)
 (csend, seq 18, #0, @267211)
 (terminate, pid 0, @267259)

Node 2
------
 (getcube, physnode 4, @248801)
 (load, pid 0, @269944)
 (csend, seq 11, #0, @270028)
 (terminate, pid 0, @270195)
```

```
Node 3
------
(getcube, physnode 6, @248862)
(load, pid 0, @274819)
(csend, seq 11, #0, @274905)
(terminate, pid 0, @275060)

------------------------------------------------------------------------
No unused nodes
No unterminated processes
No incomplete receives

Built 4 schedules
SRM>
```

Having extracted a transcript from each node, debug transparently invokes mksched. At this time, the recovered transcripts are displayed and massaged into replay and analysis schedules. To minimize perturbation, no event processing occurs during the recording phase. As a result, transcript times are absolute, and both node and sequence numbers are physical values. The necessary translation takes place in mksched during schedule generation.

The first event in a transcript is always a `getcube` record, corresponding to the allocation of a four node cube. This does not constitute an event, since the associated information is not required for execution replay. Consequently, no event number is assigned. Of interest to mksched are the `getcube` entry's two attributes, `physnode` and `@` (at). `Physnode` has as its value the physical number of the local node. Thus, mksched can determine that the nodes logically numbered 0 through 3 correspond to the physical numbers 0, 2, 4, and 6. The second attribute of `getcube`, `@`, marks the creation time of the allocation in milliseconds. Mksched subtracts this from all other absolute timestamps to yield relative timestamps.

Each transcript has a subsequent `load` entry. As with `getcube`, `loads` are not given event status, but do provide important information. The `load` record's attributes identify the process ID and creation time of a newly loaded process.

In the transcripts for nodes 1 through 3, `csend` records correspond to the sole statement in the sender process. The attributes identify the message sequence number and transmission time. As an occurrence of `csend` is an event, an event number (preceded by '#') is also given.

A final `terminate` entry appears in the transcript of every node which runs a process that exits, either normally or abnormally. The record's two attributes identify and timestamp the terminating process.

Node 0's transcript records the actions of the receiver process. The first `crecv` entry corresponds to the receipt of node 1's message. The entry's attributes give the event number along with the sender's node and sequence numbers. Unlike the `csend` entry, `crecv`'s `@` attribute is an interval. The interval's first value denotes the time at which the `crecv` system call was made. The second value marks the time at which the `crecv` consumed the message and completed. Following the `csend` is a `csendrecv` event. This is generated as a result of the receiver's `printf()` statement. The next two `csend`/`csendrecv` pairs correspond to the receipt of node 2 and 3's messages on subsequent passes through the receiver process' loop. Finally, the receiver process exits, resulting in a terminate entry.

The debug utility displays the message "`Built X schedules`" once all replay and analysis schedules have been written. These are saved to a file named by the `save` command's optional argument. The execution can subsequentlv be replayed using the debug utility:

```
SRM> debug replay sched1
sched1.rs.I: 4 schedules of 128 Kbytes
SRM> load 2 sender
SRM> load 1 sender
SRM> load 3 sender
SRM> load 0 receiver
SRM> Message from node 1
Message from node 2
Message from node 3
```

Debug's `replay` command transmits a replay schedule to the double agent on each node. The execution can then be reproduced by loading the appropriate executables. In the example above, sender processes are loaded in a different sequence than in the recorded run. Consequently, the order in which sender messages queue at node 0 is also different. Nonetheless, the event interface guides execution through the same path as specified in the replay schedules. The result is output identical to that of the original run.

Figure 7 shows a second system process, the monitor agent. During replay, the event interface intercepts system calls and other program actions and presents these to the monitor agent as events. Ultimately, monitor agents will cooperate to collectively analyze execution. Currently, the information they receive is sent to centralized analysis facilities on the iPSC/2's front end. These facilities use the previously generated analysis schedules to determine temporal relationships between reproduced events.

Significant overhead is incurred each time a replayed event is presented to the monitor agent process. This consists of the two context switch times required to schedule and unschedule the monitor agent. Further, four context switches are required for alterable events: two preceding event commitment and two following. This scheduling penalty is the price paid for ease of development. As the monitor agent is a self-contained process, it is easily loaded from the debug utility, and possesses privileges between those of user and system processes. This obviates the time-consuming kernel compilation, linking, and rebooting that were previously required.

Although th< monitoring cost cannot be reduced, the number of events processed can be. This is the intent behind the *interest* and *control* sets. The interest set contains a list of all events which will be considered by the monitor agent. The control set lists the subset of these events which can be modified.

The system queues governing the schedule/analyze/deschedule cycle are shown in figure 11. Initially, the queues are empty. When debugging is to be performed, the monitor agent process is loaded and finds its way onto the *runq*. There, the agent will carry out its initialization before suspending itself on the *agentwaitq*. Once the replay schedules have also been loaded, the user process can be faithfully re-executed.

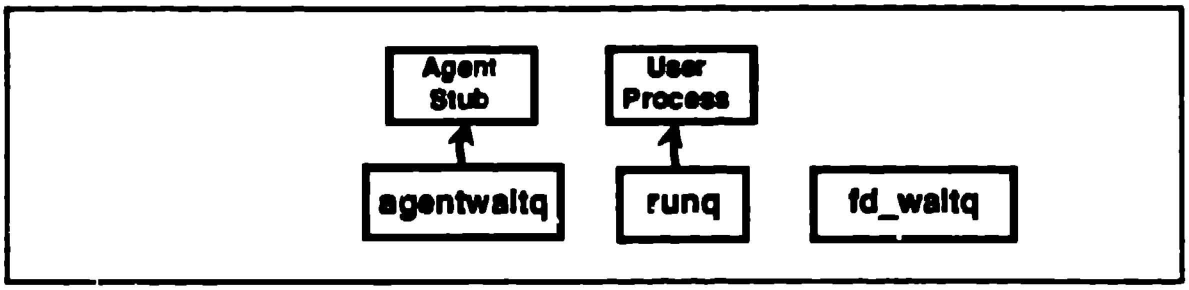


Figure 11. System Queues Involved in Schedule/Analyze/Deschedule Cycle

The user process will continue executing and remain on the runq until encountering an event. At that time, it will trade places with the monitor agent process and be suspended on the agentwaitq. With the monitor agent then in control, the event may be analyzed and possibly altered. Finally, the monitor agent and user process again exchange queues, and execution is resumed until the next event. During the course of its execution, the user process may be suspended on a variety of other system queues. Of these, the most common is the *fd_waitq*, which is involved in message passing. Unlike conventional processes, the monitor agent process will not be dispatched during these idle periods.

No debugging instructions are inserted into the user process, nor is it altered in any way. PARADIGM's monitoring is restricted to the mechanism presented here. Consequently, any source-level debugger may be used independently of, or in conjunction with, PARADIGM's analysis facilities. Event- and flow control-based breakpoints may thus be used interchangeably between PARADIGM and a source-level debugger. Similarly, PARADIGM's replay capability has also been transparently used with other debuggers such as DECON.

6. Future Work

PARADIGM addresses the problems of the probe effect with execution replay. Consequently, it should be possible to distribute the analysis along with the application. In our first attempts, distribution will be entirely the user's responsibility. Ultimately, however, we hope to automate this task. The result should be a more scalable system in which analysis approaches the rate of event generation.

Another area to be addressed is the recording time limitation imposed by in-core event transcription. The solution typically employed is a distributed checkpoint. Periodically, all nodes simultaneously suspend processing and generate checkpoints. Event buffers are also flushed to a file at this time. All nodes then simultaneously resume execution until the next checkpointing interval is reached. With this approach, recording duration is limited only to out-of-core storage capacity. The solution requires only that nodes remain well synchronized and that there be sufficient in-core storage to buffer events between checkpoints.

A problem introduced by distributed checkpoints is perturbation of the network state. Each time the system globally halts for a checkpoint, the network is evacuated. When execution resumes, messages which would have been in transit will instead be queued at their destinations. Consequently, execution may differ between checkpointed and uncheckpointed runs.

A possible solution is the *cursive checkpoint* shown in figure 12. As with the previous approach, execution is globally suspended at regular checkpointing intervals. At each checkpoint, however, the system passes through five phases. During the first phase, en route messages are collected at their destination nodes. In the second phase, the messages are returned to their originating nodes. The third phase involves the actual checkpointing and buffer flushing. Nodes then compensate for clock drift in the fourth phase. Finally, in the fifth phase, each node resends its returned messages at the same intervals as it did prior to the checkpoint. Consequently, network state should be restored when execution resumes.

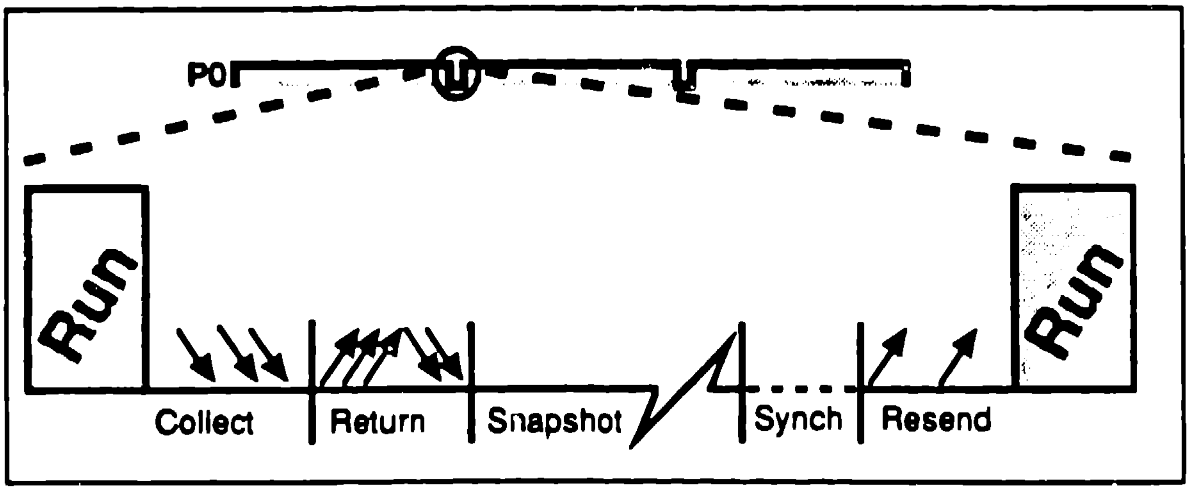


Figure 12. Five Phases of Cursive Checkpoint

The cursive checkpoint requires significant clock precision. This requirement can be relaxed by decreasing the interval between checkpoints. It is not yet known whether reasonable checkpoint intervals can be obtained under this approach.

ACKNOWLEDGEMENTS

This work would not have been possible without the generous assistance of several people. I would like to offer my gratitude to each of them: to Thom ыrando, for helping to make the project a reality; to Al Hough, for pointing me in the right direction; to Dale Wade for helping to build PARADIGM; and to John McCoubrey for remembering all the grammar I have long forgotten.

The work described in this paper was funded by The MITRE Corporation at its offices in Bedford, MA.

REFERENCES

[1] Brown, A., and W. Sampson, 1973, *Program Debugging: The Prevention and Cure of Program Errors*, London: Macdonald.

[2] Garcia-Molina, H., F. Germano, Jr., and W. Kohler, 1984, "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, SE-10(2), 210–219.

[3] McDowell, C., and D. Helmbold, 1989, "Debugging Concurrent Programs," *Computing Surveys*, 21(4), 593–622.

[4] Cournois, P., F. Heymans, and D. Parnas, 1971, "Concurrent Control with 'Readers' and 'Writers,'" *Communications of the ACM*, 14(10), 667–668.

[5] Gait, J., 1985, "A Debugger for Concurrent Programs," *Software Practices and Experience*, 15(6), 539–554.

[6] Leblanc, R., and A. Robbins, 1985, "Event-Driven Monitoring of Distributed Programs," *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, 515–522.

[7] LeBlanc, T., and J. Mellor-Crummey, 1987, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, C-36(4), 471–482.

[8] Wittie, L., 1988, "Debugging Distributed C Programs by Real Time Replay," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 57–67.

[9] Curtis, R., and L. Wittie, 1982, "BugNet: A Debugging System for Parallel Programming Environments," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami, FL, 394–399.

[10] Feldman, S., and C. Brown, 1988, "IGOR: A System for Program Debugging via Reversible Execution," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 112–123.

[11] Pan, D., and M. Linton, 1988, "Supporting Reverse Execution of Parallel Programs," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 124–129.

[12] Forin, A., 1988, "Debugging of Heterogeneous Parallel Programs," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 130–139.

[13] Elshoff, I., 1988, "A Distributed Debugger for Amoeba," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 1–10.

[14] Bates, P., and J. Wileden, 1982, "EDL: A Basis For Distributed System Debugging Tools," *Proceedings of the 15th Hawaii International Conference on System Sciences*, 86–93.

[15] Hough, A., and J. Cuny, 1987, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, 498–506.

[16] Harter, P., Jr., D. Heimbigner, and R. King, 1985, "IDD: An Interactive Distributed Debugger," *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, 498–506.

[17] Kubin, R., L. Rudolph, and D. Zernik, 1988, "Debugging Parallel Programs in Parallel," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 216-225.

[18] Helmbold, D., and D. Luckham, 1985, "TSL: Task Sequencing Language," *Proceedings of the Ada International Conference*, Paris, France, 255–274.

[19] Goldszmidt, G., S. Katz, and S. Yemini, 1988, "Interactive Black Box Debugging for Concurrent Languages," *Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, 24(1), 271–282.

[20] Snodgrass, R., 1988, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, 6(2), 157–196.

[21] Schwan, K., R. Ramnath, S. Vasudevan, and D. Ogle, 1988, "A Language and System for the Construction and Tuning of Parallel Programs," *IEEE Transactions on Software Engineering*, SE-14(4), 455–471.

[22] Hough, A., 1991, *Debugging Parallel Programs using Abstract Visualizations*, Ph.D. thesis, COINS Department, University of Massachusetts, Amherst, MA.

[23] Smith, E., 1985, "A Debugger for Message-based Processes," *Software Practice & Experience*, 15(11), 1073-1086.

[24] Malony, A., J. Arendt, R. Aydt, D. Reed, D. Grabas, and B. Totty, 1989, "An Integrated Performance Data Collection, Analysis, and Visualization System," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, Monterey, CA, 229–236.

[25] Tsai, J., K. Fang, and H. Chen, 1990, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems," *Computer*, 23(3), 11–23.

[26] Mink, A., R. Carpenter, G. Nacht, and J. Roberts, 1990, "Multiprocessor Performance-Measurement Instrumentation," *Computer*, 23(9), 63–75.

[27] Reilly, M., 1990, *A Performance Monitor for Parallel Programs*, Boston: Academic Press, pg. 12.

[28] Lamport, L., 1978, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), 558–564.

[29] Williams, M., S. Smith, and G. Pecelli, 1989, "Experimentally Driven Visual Language Design: Texture Perception Experiments for Iconographic Displays," *Proceedings of the 1989 IEEE International Workshop on Visual Languages*, Rome, Italy, 62–67.

[30] LeBlanc, T., J. Mellor-Crummey, and R. Fowler, 1990, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, 9(2), 203–217.

On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism

John Mellor-Crummey

CRPC-TR91133
1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

On-tl e-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism[*]

John M. Mellor-Crummey[†]
(johnmc@rice.edu)
Center for Research on Parallel Computation
Rice University, P.O. Box 1892
Houston, TX 77251-1892

August 1991

Abstract

Detecting data races in shared-memory parallel programs is an important debugging problem. This paper presents a new protocol for run-time detection of data races in executions of shared-memory programs with nested fork-join parallelism and no other inter-thread synchronization. This protocol has significantly smaller worst-case run-time overhead than previous techniques. The worst-case space required by our protocol when monitoring an execution of a program P is $O(VN)$, where V is the number of shared variables in P, and N is the maximum dynamic nesting of parallel constructs in P's execution. The worst-case time required to perform any monitoring operation is $O(N)$. We formally prove that our new protocol always reports a non-empty subset of the data races in a monitored program execution and describe how this property leads to an effective debugging strategy.

1 Introduction

Parallel programs for shared-memory multiprocessors can exhibit *schedule-dependent* bugs, which cause erroneous behavior on some, but not all, execution schedules. The principal cause of such errors is unsafe or inadvertent communication through shared variables. If one thread of execution updates a shared variable concurrently with another thread's access to that variable, the program's behavior may depend on the temporal order of the accesses. Such concurrent accesses are known as "data races" or "access anomalies".

Pinpointing data races is difficult since adding diagnostic statements to a program can alter the relative timing of operations and change the set of execution schedules likely to occur. The act of trying to isolate a data race responsible for a schedule dependent error can cause the error to vanish. Thus, the technique used to debug sequential programs re-executing them with instrumentation to provide information about program variable values is likely to be ineffective for pinpointing data races in parallel program executions.

Three principal strategies have been proposed for isolating data races in parallel programs: static analysis, post mortem analysis, and on the fly analysis.

Static analysis relies on classical dependence analysis of a program's text to determine when two references may refer to the same shared variable. Static techniques conservatively report dependences that include all potential data races that could occur during parallel execution. Strategies

[*] To appear in *Proc. of Supercomputing '91*, Albuquerque, NM, November 1991

[†] This work was supported in part by National Science Foundation Cooperative Agreement CCR 9045252

range from those that consider loop parallelism [1, 4], to those that consider more general tasking models [3, 14]. The conservative nature of static techniques, however, often leads to reports of data races that could never occur during execution. Experience with static analysis tools has shown that the number of false positives reported using these techniques is too high for programmers to rely exclusively on static methods for isolating data races. Combining static analysis with symbolic execution offers hope for reducing reports of infeasible races [15].

Post-mortem techniques for detecting data races involve collecting a log of events that occur during a program's execution and post-processing the log to look for evidence of data races [2, 5, 10]. If exhaustive logs are recorded, post-mortem techniques will report only feasible races. The primary drawback with post-mortem techniques is that execution logs can be enormous for parallel programs that execute for more than a trivial amount of time.

On-the-fly techniques involve augmenting a program to detect and report data races as they occur during its execution [6, 7, 9, 11, 12, 13]. These techniques maintain additional information at run-time to determine when conflicting accesses to a shared variable have occurred. Like post-mortem techniques based on exhaustive logging, on-the-fly techniques report only feasible races. In general, on-the-fly techniques require less space than post-mortem techniques since much information can be discarded as an execution progresses.

On-the-fly techniques for detecting data races fall into two classes: summary methods [9, 12, 13] that report the presence of a data race with incomplete information about the references that caused it, and access history methods [7, 11] that can precisely identify each of a pair of accesses involved in a data race. From a programmer's standpoint, the precision of the information possible using access history methods is desirable for debugging. In the remainder of this paper, we focus on access history methods.

To pinpoint accesses involved in data races, access history methods maintain two types of information at run time: the threads (along with annotations identifying the source code statements involved) that have accessed each shared variable, and information that enables determination of whether any two threads are logically concurrent. When a thread t accesses a shared variable, the thread

1. determines if any thread in the history list performed an access that conflicts with t's current access,

2. reports a data race if a thread that made a conflicting access is logically concurrent with t,

3. removes from the history list the names of any threads that sequentially precede t in the execution and adds t to the list.

A drawback of previous access history protocols (*i.e.*, those used by Dinning & Schonberg's *Task Recycling* [6, 7] and Nudler & Rudolph's *English Hebrew Labeling* [11]) is that in the worst case, each shared variable's access history must contain names for as many as T threads where T is the maximum amount of logical concurrency in the program to guarantee that these protocols will never certify a program execution as *race free* when it actually contains a data race. The space requirements for maintaining such long access histories limit the usefulness of these techniques. In practice, approximations to these protocols have been implemented that maintain abbreviated access histories of length one or two [6, 7]; however, using abbreviated histories, these protocols can erroneously certify program executions as being free of data races.

In this paper we present a new access history protocol for detecting data races on the fly in executions of programs with nested fork join parallelism. In contrast to previous access history protocols, our protocol bounds the length of each variable's history list by by a small constant that is program independent, yet our protocol ensures that if any data races exist in an execution, at least one will be reported. With this condition, an execution will never be erroneously certified as race free.

Bounding the length of history lists has two advantages. First, it reduces the worst-case space requirements. Second, it reduces the worst-case number of operations necessary to determine whether a thread's access is logically concurrent with any prior conflicting accesses.

Section 2 presents a graph model of fork-join program executions. This model serves as a framework for proving the correctness of our access history protocol. Section 3 presents *Offset-Span Labeling*, an on-line method for assigning names to threads in executions of programs with nested fork-join parallelism. Using Offset-Span Labeling, the concurrency relationship between any pair of threads can be inferred by comparing their names. Although similar to English-Hebrew Labeling [11], in the worst-case, Offset-Span Labeling assigns asymptotically shorter thread names, which lead to improved space and time bounds for access history protocols that use them. Section 4 presents our new protocol that uses bounded access histories to detect data races. Using properties of fork-join graphs and their respective Offset-Span labelings, we prove that if any data races exist in an execution of a program with nested fork-join parallelism but no other inter-thread synchronization, our protocol will report at least one data race for each shared variable involved in a race. Section 5 compares the time and space overhead of using our access history protocol and Offset-Span labels against the overhead with incurred using other access history methods. Section 6 describes the current status of this work and directions for future work.

2 A Model of Concurrency in Fork-Join Program Executions

This section defines fork-join graphs that model the run-time concurrency structure possible using closed, nestable fork-join constructs. Parallel Fortran programs that use nested parallel loops and sections are an instance of this programming model.

A *fork* operation terminates a thread and spawns a set of logically concurrent threads. Each *fork* operation has a corresponding *join* operation; when all of the threads descended from a fork terminate, the corresponding join succeeds and spawns a single thread. A thread participates in no synchronization operations other than the fork that spawned it and the join that terminates it. Each vertex in a fork-join graph represents a unique thread executing a (possibly empty) sequence of instructions. Each edge in a fork-join graph is induced by synchronization implied by a fork or join construct. A directed edge from vertex t_1 to vertex t_2 indicates that thread t_1 terminates execution before thread t_2 begins execution. Figure 1 shows a fragment of parallel Fortran and a fork-join graph that models the concurrency present during an execution of the code. Entering a parallel loop corresponds to a fork; exiting a parallel loop corresponds to a join. Each vertex in the fork-join graph is labeled with the sequence of code blocks whose execution it represents.

Before formally defining fork-join graphs, we define some useful notation for directed acyclic graphs (DAGs). In a DAG $G = (V, E)$, the *path relation* $x \rightsquigarrow_G y$ is true for $x, y \in V$ *iff* there is a path from x to y along edges in E; similarly $x \not\rightsquigarrow_G y$ is true *iff* there exists no directed path from x to y along edges in E. The *path star relation* $x \rightsquigarrow_G^* y$ is true for $x, y \in V$ *iff* $x \rightsquigarrow_G y \vee x = y$, namely there is a path from x to y along edges in E, or x and y are the same vertex.

Definition 1 constructively defines fork-join graphs which represent the concurrency relationships among threads in an execution of a fork join program. Fork-join graphs are a subset of series parallel graphs. The rules for constructing fork join graphs ensure that no vertex has a singleton predecessor with outdegree 1. Such a pair of vertices would represent a pair of threads that execute sequentially. The rules for composing fork join graphs collapse such pairs since their concurrency relationship is

```
[code block A]
PARALLEL DO I=2,4
  [code block B]
  IF (I.EQ.2) THEN
    PARALLEL DO J = 1,2
      [code block C]
    ENDDO
  ENDIF
  [code block D]
  PARALLEL DO J=1,I
    [code block E]
  ENDDO
  [code block F]
ENDDO
[code block G]
```

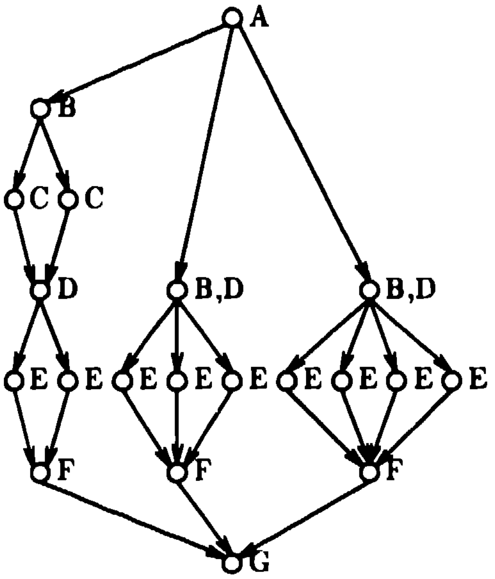


Figure 1: A fragment of parallel Fortran and its corresponding fork-join graph.

trivial.

Definition 1 *A* fork-join graph $G = (V, E, v_{src}, v_{snk})$ *is a DAG that*

- *has a designated source vertex v_{src} such that $v_{src} \leadsto_G^* v$, for all $v \in V$.*
- *has a designated sink vertex v_{snk} such that $v \leadsto_G^* v_{snk}$, for all $v \in V$.*
- *can be constructed using the following rules:*

  1. *A singleton vertex v denotes a trivial fork-join graph $G = (\{v\}, \emptyset, v, v)$.*
  2. *A compound fork-join graph can be formed in two ways:*

     parallel composition

     A set $S = \{G_i = (V_i, E_i, v_{src_i}, v_{snk_i}) | i = 1, n\}$ of $n \geq 2$ disjoint fork-join graphs can be linked in parallel to form a new fork-join graph $G = (V, E, v_{src}, v_{snk})$ where

$$\begin{aligned} V &= \{v_{src}, v_{snk}\} + \bigcup_{i=1,n} V_i \\ E &= \bigcup_{i=1,n} (E_i + \{(v_{src}, v_{src_i})\} + \{(v_{snk_i}, v_{snk})\}) \end{aligned}$$

     series composition

     A pair of disjoint fork-join graphs, $G_1 = (V_1, E_1, v_{src1}, v_{snk1})$ and $G_2 = (V_2, E_2, v_{src2}, v_{snk2})$, can be linked in series by merging vertices v_{snk1} and v_{src2} to form a new fork-join graph $G = (V, E, v_{src1}, v_{snk2})$, where

$$\begin{aligned} V &= V_1 + V_2 - \{v_{src2}\} \\ E &= E_1 + E_2 - \{(v_{src2}, v) | (v_{src2}, v) \in E_2\} + \{(v_{snk1}, v) | (v_{src2}, v) \in E_2\} \end{aligned}$$

The parallel composition rule describes how to link a set S of arbitrary fork join graphs in parallel by nesting them inside a new, closed fork join construct. The parallel composition rule adds two new threads v_{src}, the thread before a new fork, and v_{snk}, the thread after the corresponding new join, as well a synchronization edge from v_{src} to the source node of each fort. join graph in S, and a synchronization edge from the sink node of each fork join graph in S to v_{snk}. In figure 1, the fork-join graph for each parallel loop is formed by parallel composition of the fork-join graph for each loop interation.

The series composition rule describes how to link a pair of arbitrary fork-join graphs in sequence by merging the sink vertex of the first graph with the source vertex of the second graph and retaining all of the edges. In figure 1, each node labeled "B,D" is the result of series composition of trivial fork-join graphs representing code blocks B and D respectively. Similarly, the fork-join graph that represents iteration I=2 of the outermost parallel loop is the series composition of the fork-join graphs for the two loops nested inside.

Two vertices v_1 and v_2 in a fork-join graph G represent logically concurrent threads in an execution of a fork-join program *iff* $v_1 \not\rightsquigarrow^*_G v_2 \wedge v_2 \not\rightsquigarrow^*_G v_1$. The only ways this formula can be falsified is if v_1 and v_2 are not distinct, or if $v_1 \rightsquigarrow_G v_2 \vee v_2 \rightsquigarrow_G v_1$. If the vertices are not distinct, the threads are the same and thus not concurrent. In the second case, the vertices are related by a path of directed edges. The interpretation of a directed edge (as described earlier) as temporal precedence and the transitivity of this precedence relation for paths of edges means that v_1 and v_2 could not in fact be concurrent if they are connected by a path of directed edges.

To facilitate inductive proofs about fork-join graphs, we define $rule(G)$ to be the minimum number of applications of the series and parallel composition rules needed to construct a fork-join graph G from a set of trivial fork-join graphs. (It is important to define $rule(G)$ to be the minimum number of rule applications since applying series composition to a pair of trivial fork-join graphs results in another trivial fork-join graph.)

3 Offset-Span Labeling

Offset-Span labeling is an on-line scheme for labeling each thread in a fork-join program execution. Each thread's label contains information that identifies its position in a corresponding fork-join graph. By comparing the labels of two threads, their concurrency relationship can be deduced. Offset-Span labeling is similar to Nudler and Rudolph's *English-Hebrew labeling* [11]. In both Offset-Span (OS) and English-Hebrew (EH) labeling, a thread in an execution of a fork-join program computes its own unique label using only local information — specifically, the label(s) of its immediate predecessor(s) in a fork-join graph. (In contrast, the Task Recycling technique [6, 7] requires a centralized data structure to maintain information about free task descriptors. It is preferable to avoid use of centralized data structures in parallel programs since they tend to introduce serial bottlenecks.) In both EH and OS labeling, the length of a thread's label increases along with the nesting depth of fork-join constructs. Also, both schemes use a lexicographic-style comparison of labels to determine if the threads they represent are concurrent.

An advantage of OS labeling is that its definition guarantees that the length of a thread's OS label is *always* proportional to the current nesting depth of the fork-join pair surrounding the thread. The length of the OS label for a thread following a join is always equal to the length of the OS label for the thread that executed the matching fork. Using EH labeling (as described in [7]), the length of a thread's label can grow in proportion to the number of fork operations encountered along the execution path leading to the creation of the thread; the length of an EH label following a join is greater than the length of the EH label for the thread that executed the matching fork. Dinning and Schonberg mention the existence of a heuristic [7, p. 4] that reportedly limits the length of EH labels to the level of nesting. It is important to minimize the length of labels used by these methods since shorter labels reduce the space required to store them at execution time as well as the time spent comparing them.

Definition 2 *An* Offset-Span labeling *of a fork-join graph G assigns a label consisting of a non-null sequence of ordered pairs to each of the vertices of G. Each ordered pair $[o,s]$ consists of two components: the* offset *and the* span. *The span indicates the number of threads spawned by an s-way fork from which this label pair is descended. The offset distinguishes among relatives descended from the same parent. An OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is computed as follows given an initial OS label for v_{src} that consists of a non-null sequence of offset-span pairs:*

1. *For a vertex $v \in V$ of outdegree $n > 1$ (v is the source node of some fork-join subgraph of G) that has an OS label of L, where L is some non-empty sequence of label pairs (hereafter, we use the notation $OSL(v) = L$): let v_i denote the ith child of v, $0 \leq i < n$ (the ordering of the children is insignificant). Assign $OSL(v_i) = L[i,n]$, where juxtaposition of L and $[i,n]$ implies concatenation.*

2. *For a vertex v of indegree $n > 1$ (v is the sink node of some fork-join subgraph of G) that has some labeled vertex v' as a predecessor: $\exists_{L,u,w,x,y} OSL(v') = L[u,w][x,y]$, where L is a (possibly null) sequence of label pairs. (In a fork-join graph, $OSL(v')$ must be of this form. Any node in G other than v_{src} or v_{snk} has an OS label consisting of at least two label pairs. By the definition of fork-join graphs, v_{src} cannot be the predecessor of any vertex of indegree > 1 and v_{snk} cannot be the predecessor of any vertex.) Assign $OSL(v) = L[u+w,w]$.*[1]

The labeling is complete since by the composition rules no vertex in a fork-join graph can have a predecessor of outdegree 1 and itself be of indegree 1. The labeling is consistent since the composition rules guarantee that any vertex that is a successor of a vertex with outdegree > 1 has indegree 1. Comparisons between two labels are made by comparing the corresponding ordered pairs in the label sequences from left to right. Each thread's OS label in an execution of a program with closed, nestable fork-join parallelism can be computed on line efficiently from the label of its predecessor. Figure 2 shows an OS labeling of the fork-join graph shown in figure 1.

The following lemma shows the relationship between the labels assigned to the source and sink of a fork-join graph. Note that this lemma also implies that the length of each thread's OS label assigned using the rules of definition 2 is directly proportional to the nesting depth of fork-join constructs surrounding the thread.

Lemma 1 *In an OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$, if v_{src} has a label $P[o,s]$, where P is an arbitrary (possibly null) sequence of ordered label pairs and o and s are arbitrary constants, then v_{snk} has label $P[o',s]$, for some o' such that $o \bmod s = o' \bmod s$.*

Proof Induction on the size of G as measured using $rule(G)$.
Base Case. For any trivial graph G ($rule(G) = 0$), the lemma is satisfied with $o' = o$.
Induction Hypothesis. Assume that the lemma holds for every fork-join graph G with $rule(G) < k$.
Induction Step. Show that the lemma holds for each fork-join graph G with $rule(G) = k$. We consider applying each of the composition rules to a collection of $G_i, i = 1, n$ ($n \geq 2$) fork-join graphs with $\sum_{i=1}^{n} rule(G_i) = k - 1$.

series An application of the series composition rule to form $G = (V, E, v_{src1}, v_{snk2})$ from 2 disjoint fork-join graphs $G_1 = (V_1, E_1, v_{src1}, v_{snk1})$ and $G_2 = (V_2, E_2, v_{src2}, v_{snk2})$, where $rule(G_1) + rule(G_2) = k - 1$. By the induction hypothesis, the lemma holds for both G_1 and G_2 separately. Let v_{src1} have OS label $P[o,s]$. By the induction hypothesis, v_{snk1} has label $P[o_1,s]$, where $o \bmod s = o_1 \bmod s$. Let v_{src2} have OS label $P[o_1,s]$, by the induction hypothesis v_{snk2} has label $P[o_2,s]$, where $o_1 \bmod s = o_2 \bmod s$. The series composition rule

[1] This label is the same regardless of the predecessor v' chosen. The label of a sink node for a fork-join subgraph is determined by the label of the corresponding source node. (See Lemma 1.)

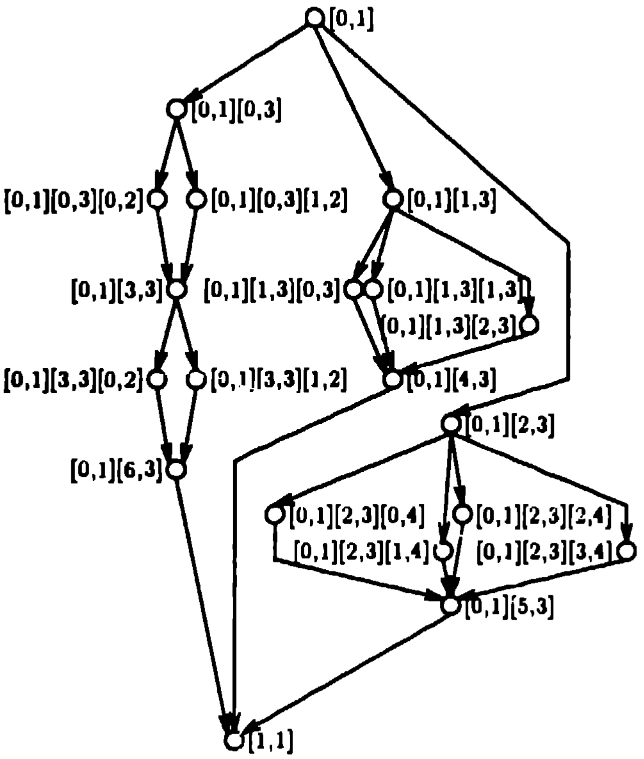


Figure 2: An Offset-Span Labeling of a fork-join graph.

merges v_{snk1} with v_{src2}. After the merge, the labeling remains consistent. Since v_{src2} has no incoming edges in G_2, the label of the merged node is completely determined by the labels of the ancestors of v_{snk1} in G_1; thus, the label of the merged node remains $P[o_1,s]$. Since v_{snk1} has no outgoing edges in G_1, the outdegree of the merged node in G remains the same as that of v_{src2} in G_2. Therefore, the labels of the descendants of v_{src2} remain the same. By transitivity, $o \bmod s = o_2 \bmod s$ and the lemma is satisfied for graph G.

parallel An application of the parallel composition rule to form $G = (V, E, v_{src}, v_{snk})$ from a set $S = \{G_i = (V_i, E_i, v_{srci}, v_{snki}) | i = 1,n\}$ of $n \geq 2$ disjoint fork-join graphs, where $\sum_{i=1}^{n} rule(G_i) = k - 1$. By the induction hypothesis, the lemma holds for both each G_i separately. Let v_{srci} have OS label $P'[i-1,n]$. By the induction hypothesis v_{snki} has label $P'[o_i,n]$, where $(i-1) \bmod n = o_i \bmod n$. The parallel composition rule links v_{src} to each v_{srci}, $i = 1,n$ and links v_{snki}, $i = 1,n$ to v_{snk}. Let the OS label of v_{src} be $P[o,s]$. Letting $P' = P[o,s]$ makes the all of the labels of nodes in subgraphs G_i, $i = 1,n$ consistent with the labeling rules. By labeling rule 2, v_{snk} is assigned label $P[o+s,s]$ since its ancestors v_{snki}, $i = 1,n$ have OS labels $P'[o_i,n] = P[o,s][o_i,n]$ respectively. The lemma is satisfied for G since $o \bmod s = (o+s) \bmod s$.

The lemma follows by the principle of induction. □

In the following lemma, we show that by comparing the OS labels for a pair of threads in an execution, it is straight-forward to determine if one thread has finished before a second thread begins (*i.e.*, the vertices representing the threads are related by $\rightsquigarrow_G$ in the fork join graph G representing the execution).

Lemma 2 *Given the OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$, $x \rightsquigarrow_G y$, is true for*

$x, y \in V$ iff *one of the following properties holds for their respective OS labels, $OSL(x)$ and $OSL(y)$*

case 1 $\exists_{P,S}(OSL(x) = P) \wedge (OSL(y) = PS)$ *where both P and S are any non-null sequence of ordered label pairs.*

case 2 $\exists_{P,S_x,S_y,o_x,o_y,s}(OSL(x) = P[o_x, s]S_x) \wedge (OSL(y) = P[o_y, s]S_y) \wedge (o_x < o_y) \wedge (o_x \bmod s = o_y \bmod s)$ *where P, S_x, and S_y are (possibly null) sequences of ordered pairs.*

Proof Any fork-join graph that contains more than one vertex must have been constructed through some sequence of applications of the parallel and series composition rules. Let $G_s = (V_s, E_s, v_{src_s}, v_{snk_s})$ be the smallest fork-join subgraph of G that contains both x and y. Case 1 holds iff G_s was constructed from a set of disjoint fork-join graphs using the parallel composition rule, $x = v_{src_s}$, and $y \in V_s - \{v_{src_s}, v_{snk_s}\}$. Case 2 holds iff (a) G_s was constructed from a set of disjoint fork-join graphs using parallel composition, $x \in V_s - \{v_{snk_s}\}$, and $y = v_{snk_s}$, or (b) G_s was constructed by linking some pair of disjoint fork-join graphs using series composition. In case 2, P is *null iff* $G_s = G$, S_x is *null iff* $x = v_{src_s}$, and S_y is *null iff* $y = v_{snk_s}$. The enumeration of ancestor relationships covered by these cases is complete. Case 1 and 2a cover all ancestor relationships if the last rule applied to form G_s was the parallel composition rule. In these cases x has to be v_{src_s} or y has to be v_{snk_s}, otherwise G_s would not be the smallest subgraph that contains both x and y with $x \leadsto_G y$. Case 2b covers all possible ancestor relationships if the last rule applied to form G_s was the series composition rule. □

Below, we define a *left of* relation that defines a partial ordering of vertices in a fork-join graph that are not related by the $\leadsto_G^*$ relation (*i.e.*, vertices that represent concurrent threads). The access history protocol described in section 4 requires a labeling scheme for which a left-of relation can be defined. English-Hebrew labels contain sufficient information to compute a left-of relation, but labels assigned by the Task Recycling technique do not Here we define a left-of relation for OS labels.

Definition 3 *For an OS labeling of a fork-join graph $G = (V, E, v_{src}, v_{snk})$, the* "left of" relation, *denoted $x \prec_G y$, is true for $x, y \in V$* iff *the following property holds for their OS labels $OSL(x)$ and $OSL(y)$*

> $\exists_{P,S_x,S_y}(OSL(x) = P[o_x, s]S_x) \wedge (OSL(y) = P[o_y, s]S_y) \wedge (o_x \bmod s < o_y \bmod s)$, *$P$ is a non-null sequence of ordered label pairs, S_x and S_y are (possibly null) sequences of ordered label pairs.*

The left-of relation establishes a canonical ordering of relatives with respect to their lowest common ancestor.

4 A Protocol for Detecting Data Races

Two accesses to the same variable are *conflicting* if at least one of them is a write. A data race in the execution of a fork-join program exists when two or more concurrent threads perform conflicting accesses to the same shared variable. In terms of the fork-join graph model, a data race exists in an execution if two threads represented by vertices v_i and v_j in a fork-join graph G perform conflicting accesses to the same shared variable and $v_i \not\leadsto_G^* v_j \wedge v_j \not\leadsto_G^* v_i$ (the threads are unordered by synchronization, and thus their executions are logically concurrent).

To detect data races on the fly, each potentially unsafe access to a shared variable during a parallel program execution must be monitored. A program transformer must allocate access history storage for each shared variable with a reference that is the endpoint of a dependence carried by a parallel construct (*i.e.*, static analysis was unable to prove that some reference by a logically concurrent thread does not result in a conflicting access to the variable). At each variable reference that is an endpoint of a dependence carried by a parallel construct, the transformer must add a call to a monitoring protocol that inspects and updates the variable's access history. The transformer must also insert statements that enable each thread to compute a label that reflects its concurrency relationship to other threads. At execution time, the monitoring protocol reports any logically concurrent, conflicting accesses to a shared variable.

For an execution of a fork-join program, the existence of a data race involving a shared variable is solely a function of which threads access it and the concurrency relationship between the threads that is implied by the fork and join constructs in the program. Therefore, we can consider data races for each shared variable independently.

We define an *access interleaving* to model a set of accesses to a shared variable by threads in a fork-join program.

Definition 4 *An* access interleaving *for a shared variable X by threads whose run-time concurrency relationship is modeled by a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is denoted I_G^X. I_G^X consists of a totally ordered sequence of accesses $A_1, \ldots, A_n$. Each access is performed by some thread; let $v_G(A) \in V$ be the vertex in G that represents the thread that performed the access A. An access $A_i \in I_G^X$ marks vertex $v_G(A_i)$ with either an X_{read} or an X_{write} token. Multiple accesses in I_G^X may mark the same vertex, and a vertex can be marked with both X_{read} and X_{write} tokens. No access $A_j \in I_G^X$ may mark a vertex $v_1 \in V$ if some $A_i \in I_G^X$, $i < j$ previously marked a vertex $v_2 \in V$ such that $v_1 \rightsquigarrow_G v_2$.*

The definition of an access interleaving assumes *sequentially consistent* [8] shared memory. We refer to an access in I_G^X as a *read* if it marks a vertex with an X_{read} token, or as a *write* if it marks a vertex with an X_{write} token.

In the remainder of this section, we present protocols for detecting data races caused by conflicting accesses to a single shared variable and prove their correctness. We formulate the problem of on-the-fly detection of data races as detecting conflicting, logically concurrent accesses in an access interleaving for a shared variable. An access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$ is *checked* if for each access $A \in I_G^X$

- if A is a read the **checkread** protocol (figure 3) is called with a pointer to X's access history and the label for thread $v_G(A)$ (the thread performing the access), and

- if A is a write the **checkwrite** protocol (figure 4) is called with a pointer to X's access history and the thread label for $v_G(A)$.

The **checkread** and **checkwrite** protocols determine whether an access by the current thread is involved in a data race with any access earlier in the interleaving. Any thread labeling scheme is suitable for use with this protocol as long as the $\rightsquigarrow_G$, $\rightsquigarrow_G^*$, and $\prec_G$ relations can be determined using label comparisions.

If **checkread** is invoked when any thread reads a shared variable X, the protocol guarantees that the $\mathbf{R}_{1r}$ component of X's access history contains the label for the "lowest", "rightmost" thread, and $\mathbf{R}_{1l}$, the label for the "lowest", "leftmost" thread. The concepts of "lowest", "rightmost", and "leftmost" are well-defined for threads in an execution modeled by a fork-join graph G in terms of the $\rightsquigarrow_G$, $\rightsquigarrow_G^*$, and $\prec_G$ relations. If **checkwrite** is invoked when any thread writes to X, the protocol guarantees that the $\mathbf{W}_{last}$ component of X's access history contains the label for the thread that last performed a write to that variable.

The theorems that follow in this section show that the **checkread** and **checkwrite** protocols guarantee that if an access interleaving contains one or more data races, at least one of these races

```
checkread(access_history, thread_label)
   if access_history^.W_last $\not\rightsquigarrow^{*}_{G}$ thread_label then
      report a WRITE-READ data race
   endif
   if thread_label $\prec_G$ access_history^.R_ll or
         access_history^.R_ll $\rightsquigarrow_G$ thread_label then
      access_history^.R_ll := thread_label
   endif
   if access_history^.R_lr $\prec_G$ thread_label or
         access_history^.R_lr $\rightsquigarrow_G$ thread_label then
      access_history^.R_lr := thread_label
   endif
end checkread
```

Figure 3: Monitoring protocol for a read.

```
checkwrite(access_history, thread_label)
   if access_history^.W_last $\not\rightsquigarrow^{*}_{G}$ thread_label then
      report a WRITE-WRITE data race
   endif
   if access_history^.R_ll $\not\rightsquigarrow^{*}_{G}$ thread_label or
         access_history^.R_lr $\not\rightsquigarrow^{*}_{G}$ thread_label then
      report a READ-WRITE data race
   endif
   access_history^.W_last := thread_label
end checkwrite
```

Figure 4: Monitoring protocol for a write.

will be detected and reported. Thus, using these protocols, an execution will be reported free of races *iff* no data races are present.

Theorem 1 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$,* **checkwrite** *will report a data race for a write in I_G^X if it is logically concurrent with some earlier read in I_G^X.*

Proof Suppose $r \in I_G^X$ marks $R \in V$ with an X_{read} token, $w \in I_G^X$ marks $W \in V$ with an X_{write} token, r precedes w in I_G^X, and R and W are logically concurrent, but **checkwrite** fails to report a data race for w.

Without loss of generality, assume that vertices in V are named by their thread labels. If **checkwrite** reports no race for w, then it must be the case that $R_{lr} \rightsquigarrow^{*}_{G} W \wedge R_{ll} \rightsquigarrow^{*}_{G} W$ when **checkwrite** is called for w (*i.e.*, W is not logically concurrent with previous readers R_{lr} or R_{ll} saved by the **checkread** protocol).

Since **checkread** has been executed for each *read* preceding w in the interleaving (including r), we are guaranteed that

$$R_{ll} \prec_G R \wedge R_{ll} \not\rightsquigarrow_G R \wedge R \prec_G R_{lr} \wedge R_{lr} \not\rightsquigarrow_G R \wedge R_{ll} \not\rightsquigarrow_G R_{lr} \wedge R_{lr} \not\rightsquigarrow_G R_{ll} \qquad (1)$$

It must be the case that $R \not\rightsquigarrow^*_G \mathrm{R_{1r}} \wedge R \not\rightsquigarrow^*_G \mathrm{R_{1l}}$; otherwise, by transitivity of the $\rightsquigarrow^*_G$ relation, $R \rightsquigarrow^*_G W$, which violates the supposition that R and W are logically concurrent. This implies $R \neq \mathrm{R_{1l}} \wedge R \neq \mathrm{R_{1r}}$. Using this to refine (1) we can conclude that if such an R exists,

$$\mathrm{R_{1l}} \prec_G R \wedge R \prec_G \mathrm{R_{1r}} \tag{2}$$

If $\mathrm{R_{1r}} = \mathrm{R_{1l}}$, then (2) is not satisfiable and there can be no R concurrent with W; therefore, if such an R exists

$$\mathrm{R_{1r}} \neq \mathrm{R_{1l}} \tag{3}$$

Let $G_s = (V_s, E_s, v_{src_s}, v_{snk_s})$ be the smallest fork-join subgraph of G that contains both $\mathrm{R_{1l}}$ and $\mathrm{R_{1r}}$. By (1) and (3), $\mathrm{R_{1l}} \not\rightsquigarrow^*_G \mathrm{R_{1r}} \wedge \mathrm{R_{1r}} \not\rightsquigarrow^*_G \mathrm{R_{1l}}$; therefore, $v_{src_s} \neq \mathrm{R_{1l}} \wedge v_{src_s} \neq \mathrm{R_{1r}}$. A corollary of this is that $|V_s| > 1$ which implies $rule(G_s) > 0$. The composition rule last applied to construct G_s could not have been the series composition rule. The condition that G_s is the smallest fork-join graph containing both $\mathrm{R_{1l}}$ and $\mathrm{R_{1r}}$ would imply that one vertex must be in each of the components linked in series; this contradicts (1) since $\mathrm{R_{1l}}$ and $\mathrm{R_{1r}}$ would be related by $\rightsquigarrow_G$. Therefore, G_s must have been formed from some set S of disjoint fork-join graphs using the parallel composition rule. Both $\mathrm{R_{1l}}$ and $\mathrm{R_{1r}}$ cannot belong to the same element of S, otherwise G_s would not be the smallest fork-join graph containing them both. Therefore, v_{src_s} is the closest common ancestor of $\mathrm{R_{1l}}$ and $\mathrm{R_{1r}}$, and v_{snk_s} is their closest common descendant. Since $\mathrm{R_{1r}} \rightsquigarrow_G W$ and $\mathrm{R_{1l}} \rightsquigarrow_G W$, then $v_{snk_s} \rightsquigarrow^*_G W$. As justified below, v_{src_s} must be an ancestor of R (*i.e.*, $v_{src_s} \rightsquigarrow_G R$):

- If $R \rightsquigarrow^*_G v_{src_s}$, then $R \rightsquigarrow_G \mathrm{R_{1l}} \wedge R \rightsquigarrow_G \mathrm{R_{1r}}$. By transitivity of the path relation, $R \rightsquigarrow_G W$, contradicting the supposition that R and W are logically concurrent.

- If R is to the left of v_{src_s}, then by definition of $\prec_G$, $R \prec_G \mathrm{R_{1l}}$, contradicting (1).

- If v_{src_s} is to the left of R, then by definition of $\prec_G$, $\mathrm{R_{1r}} \prec_G R$, contradicting (1).

Also, $v_{snk_s} \not\rightsquigarrow_G R$, otherwise, by transitivity $\mathrm{R_{1r}} \rightsquigarrow_G R \wedge \mathrm{R_{1l}} \rightsquigarrow_G R$, contradicting (1). By the definition of closed, nestable fork-join graphs, every descendant of a source vertex that is not a descendant of the corresponding sink vertex must be an ancestor of the sink vertex. Therefore, since $v_{src_s} \rightsquigarrow_G R \wedge v_{snk_s} \not\rightsquigarrow_G R$, then $R \rightsquigarrow_G v_{snk_s}$. But then by transitivity, $R \rightsquigarrow_G W$, contradicting the supposition that R and W are logically concurrent.

By showing a contradiction in every case to the supposition that there can exist some read r that precedes a write w in I_G^X such that they mark logically concurrent vertices but **checkwrite** fails to report a data race for w, the theorem is proven. □

Theorem 2 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, if any two writes in I_G^X are logically concurrent, then* **checkwrite** *will report a data race.*

Proof Suppose vertices $V_w \subseteq V$ are marked with X_{write} tokens by accesses in I_G^X and at least one pair of vertices in V_w is concurrent. Two writes in an access interleaving are *adjacent* if there is no other write between them in the sequence. If the vertices marked by each pair of adjacent writes in I_G^X are related by the path star relation $\rightsquigarrow^*_G$, by transitivity of $\rightsquigarrow^*_G$ no pair of writes in V_w would be concurrent. By the original supposition, at least two of the vertices in V_w are concurrent; therefore, some pair of vertices $v_1, v_2 \in V_w$ that are marked by a pair of adjacent writes in I_G^X must not be related by $\rightsquigarrow^*_G$. Without loss of generality, let v_1 be the vertex marked by the first of the adjacent writes; thus, $v_1 \not\rightsquigarrow^*_G v_2$. Since the writes by v_1 and v_2 are adjacent, $\mathbf{W_{last}}$ will contain the thread label for v_1 when **checkwrite** is called for the following write by v_2; **checkwrite** will report a data race since $v_1 \not\rightsquigarrow^*_G v_2$. We have shown that if write accesses in I_G^X mark any two concurrent vertices in G, then a data race will be reported, thus proving the theorem. □

Theorem 3 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, a data race will be reported if a read in I_G^X is logically concurrent with some earlier write in I_G^X.*

Proof Suppose $w \in I_G^X$ marks $W \in V$ with an X_{write} token, $r \in I_G^X$ marks $R \in V$ with an X_{read} token, W precedes R in I_G^X, and W and R are logically concurrent, but no data race is reported.

Without loss of generality, assume that vertices in V are named by their thread labels. If there is no intervening write between w and r in I_G^X, when **checkread** executes for r, $\mathtt{W_{last}} = W$ and **checkread** will report a data race since by supposition W and R are concurrent.

If there is some sequence of writes $w_1, \ldots, w_n$ between w and r in I_G^X then it cannot be the case that $W \rightsquigarrow^*_G v_G(w_1)$, $v_G(w_i) \rightsquigarrow^*_G v_G(w_{i+1})$ for $1 \leq i < n$, and $v_G(w_n) \rightsquigarrow^*_G R$; otherwise by transitivity of the $\rightsquigarrow^*_G$ relation $W \rightsquigarrow^*_G R$, contradicting our original supposition that they are concurrent. If $W \rightsquigarrow^*_G v_G(w_n)$, then $v_G(w_n) \not\rightsquigarrow^*_G R$, otherwise W and R could not be concurrent. In this case, at vertex R, $\mathtt{W_{last}}$ would contain the label for $v_G(w_n)$ and **checkread** would report a data race between $v_G(w_n)$ and R. Otherwise, if $W \not\rightsquigarrow^*_G v_G(w_n)$, then w is concurrent with w_n and by theorem 2 **checkwrite** will report at least one data race for some pair of adjacent writes in the subsequence of I_G^X beginning with w and ending with w_n. □

Theorem 4 *In a checked access interleaving I_G^X for a variable X and a fork-join graph $G = (V, E, v_{src}, v_{snk})$, at least one data race will be reported if there are any conflicting, logically concurrent accesses in I_G^X.*

Proof There are three cases of conflicting accesses to consider,

1. a read is concurrent with a write, and the read precedes the write in I_G^X,

2. two writes are concurrent,

3. a read is concurrent with a write, and the write precedes the read in I_G^X.

By theorem 1, a data race will be reported for any concurrent accesses in case 1. By theorem 2 a data race will be reported for any concurrent accesses in case 2. Finally, by theorem 3, a data race will be reported for any concurrent accesses in case 3. □

Theorem 4 shows that if any data races are present in an access interleaving for a shared variable, at least one will be reported using our **checkread** and **checkwrite** access history protocols. By applying the solution to detect any races for an individual shared variable to each of the shared variables in a program, we can guarantee that if a program execution exhibits any data races given a particular input, then the **checkread** and **checkwrite** protocols will report at least one data race for each shared variable that is actually involved in a race during that execution.

Using the monitoring protocol described in this section leads to an effective debugging strategy for eliminating data races from a program execution for a given input. Run the program on the given input with the monitoring protocol in place. Each time a data race is reported (the access history protocol precisely reports both endpoints of the race), fix the cause of the data race, and re execute the program with the same input. Since the access history procotols given in this section will report data races (if any exist) regardless of the interleaving order, the protocol can be used to check for races in a program that is executed in a canonical serial order. Executing programs in a canonical serial order while debugging is often convenient as it provides the user with deterministic behavior that simplifies the task of determining the origin of variable values that indirectly caused a data race to occur.

If no race is detected in an execution, then no race will occur in *any* execution of the program for that particular input and the program is guaranteed to be deterministic *for that input*. The key insight behind this observation is that the only thing that could cause an execution for the given

Algorithm	Space	Time	
		Thread Creation & Termination	Per Access
Task Recycling	$O(VT + T^2)$	$O(T)$	$O(T)$
EH Labeling	$O(VT + \min(BN, VTN))$	$O(N)$	$O(NT)$
OS Labeling	$O(V + \min(BN, VN))$	$O(N)$	$O(N)$

Table 1: Comparison of Worst Case Time and Space Requirements.

input to behave differently would be if there were some form of non-determinism present. Data races are the sole source of non-determinism in programs that have nested fork-join parallelism but no other inter-thread synchronization. Therefore, if no data race is detected in one execution of such a program for a given input, then no data race can exist in any execution for that input.

Practical implementations of the **checkread** and **checkwrite** protocols described in this section must respect the underlying assumptions upon which the correctness proofs are based. In particular, all updates and inspections of an access history by the **checkread** and **checkwrite** protocols must be coordinated. Without coordinating updates to a variable's access history, the **checkread** protocol could not correctly maintain the invariants with respect to R_{1r} and R_{1l}. The simplest coordination strategy is enforcing mutually exclusive access. Such coordination could cause bottlenecks if there is pervasive read sharing of a variable among concurrent threads. By using dependence analysis to limit monitoring instrumentation to only the cases in which read-write conflicts seem imminent, hopefully such bottlenecks could be avoided. Other less restrictive coordination strategies appear possible, but it would be necessary to relax some of the invariants maintained by the protocols and show that data races are guaranteed to be detected even with relaxed invariants.

5 Analysis

In this section we examine the space and time complexity of using our access history protocol with Offset-Span labels and compare it to the complexity of the protocols described in the literature for English-Hebrew Labeling [11] and Task Recycling [6, 7]. To be consistent with the notation of Dinning and Schonberg [6], we present our analysis in terms of the following parameters:

T - maximum logical concurrency
V number of monitored shared variables
N maximum level of fork-join nesting
B total number of threads in an execution

Table 1 compares the worst case time and space complexity of the earlier access history methods, English Hebrew Labeling and Task Recycling, with the worst case time and space complexity of our access history protocol using Offset Span labels.

For the EH Labeling and Task Recycling access history protocols described in the literature, each monitored variable has an access history that may contain as many as T thread names if the variable is accessed by each thread that is active when the program attains its maximum logical concurrency; this leads to the VT term in the their space complexities. The second term in the space complexity of Task Recycling arises because each thread has an associated "parent vector" of length T that is used to summarize the concurrency relationships between a thread and its ancestors.

Since T threads may be active simultaneously, T^2 space may be needed. In EH Labeling, the size of an EH label for a thread is proportional to the nesting depth of fork-join constructs which is bounded by N. (This analysis assumes the existence of an effective heuristic alluded to by Dinning and Schonberg [7, p. 4] that limits the length of labels to $O(N)$. Without the heuristic, labels can grow arbitrarily long. A description of the heuristic was unavailable to the author of this paper at the time of this publication.) If access histories store pointers to EH labels, each label is at most of length N, and there can be at most VT distinct pointers to labels. If reference counting garbage collection is used, the maximum space used to store EH labels is bounded by $O(VTN)$. If the number of threads in a program execution B is less than VT, then this places a tighter bound on the space to store the labels of $O(BN)$ since at most one label per thread needs to be stored.

In the expression for the worst-case space complexity for our new access history protocol using Offset-Span labels, the first term accounts for the constant size access history for each monitored variable. The second term reflects the space needed to store OS labels. If access histories store pointers to OS labels, each label is at most of length N, and there can be at most $O(V)$ distinct pointers to OS labels. If reference counting garbage collection is used, the maximum space used to store OS labels is bounded by $O(VN)$. If the number of threads in a program execution B is less than V, then this places a tighter bound on the space to store the labels of $O(BN)$ since at most one label per thread needs to be stored.

The worst case time to verify whether an individual access to a variable is involved in a data race is $O(TN)$ for the EH Labeling protocol since an access may need to be compared against T entries in the variable's access history and each comparison may take $O(N)$ time. For Task Recycling, the worst case time to verify whether an individual access to a variable is involved in a data race is $O(T)$; the parent vector representation in Task Recycling enables access comparisons in constant time, but a comparison may be needed for each of T entries in a variable's access history. For our new access history protocol with Offset-Span labels, the corresponding time is only $O(N)$ since the label for the current access need only be compared with a constant number of other labels.

The worst-case time overhead at thread creation for EH and OS labeling is $O(N)$ for assignment of a label of size $O(N)$ to a thread. Task Recycling incurs worst-case overhead of $O(T)$ at thread creation and termination since a parent vector of size $O(T)$ may need to be created for a new thread, and when theads meet at a join, their parent vectors of size $O(T)$ must be merged.

Since T is typically greater than 2^N, using our new access protocol represents a significant worst-case savings in both space and time over earlier protocols for on-the-fly detection of data races.

6 Status and Future Work

A prototype system for dependence based instrumentation of potential data races in parallel FORTRAN programs has been developed as part of the debugging system in the ParaScope Programming Environment [4]. The instrumentation system inserts calls to a run time library that uses Offset-Span Labeling and the access history protocol described in section 4. The prototype instrumentation system currently handles simple programs with loop based parallelism. Currently, procedure calls from within parallel loops are not handled. Ongoing implementation efforts are focused on extending interprocedural analysis in ParaScope so that the dependence based instrumentation can interprocedurally propagate requirements for instrumentation into procedures called from within parallel constructs. Once the interprocedural instrumentation system is complete, the on the fly debugging system will be useful for more than toy programs.

Future work includes extending the access history protocol and proofs to handle regular patterns of synchronization such as sections in DOACROSS loops and the Pcf Fortran generalization of this construct: ordered sequence synchronization. Preliminary indications are that the protocols will extend naturally to accommodate this larger class of programs.

Acknowledgments

I thank the referees for the improvements they suggested and I am indepted to the referee who pointed out several erratta. Robert Hood and Seema Hiranandani participated in early discussions of these ideas. Robert Hood implemented the prototype dependence-based instrumentation system.

References

[1] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 164–170, Aug. 1986.

[2] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. of the 1987 International Conference on Parallel Processing*, pages 721–727, Aug. 1987.

[3] W. F. Appelbe and C. E. McDowell. Anomaly reporting – a tool for debugging and developing parallel numerical applications. In *Proc. First International Conference on Supercomputers*, FL, Dec. 1985.

[4] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Sublok. The ParaScope editor: An interactive parallel programming tool. In *Proc. Supercomputing '89*, pages 540–550, Reno, NV, Nov. 1989.

[5] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 1991.

[6] A. Dinning and E. Schonberg. An evaluation of monitoring algorithms for access anomaly detection. Ultracomputer Note 163, Courant Institute, New York University, July 1989.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, Mar. 1990.

[8] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[9] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Palo Alto, CA, Apr. 1991.

[10] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1991. Also in *Proc. of the 3rd Workshop on Prog. Langs. and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

[11] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1988. Cited in [7].

[12] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, June 1989.

[13] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proc. of the 1990 Symposium on the Principles of Programming Languages*, pages 218–231, Jan. 1990.

[14] R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[15] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct. 1988.

Block-Structured Control of Parallel Tracing[1]

Cherri M. Pancake
(Visiting Scientist, Cornell Theory Center)
Department of Computer Science and Engineering
Auburn University, Alabama 36849
email: pancake@ducvax.auburn.edu

Despite the increasing attention being given to the design and implementation of parallel debuggers (see [19, 12]), users continue to be dissatisfied [15, 13, 3]. Some of the criticisms reflect the technological difficulties of monitoring parallel execution in nonn-intrusive ways, or of reproducing behavior in an inherently unstable environment. Other complaints, however, address a more fundamental problem: providing execution information that relates meaningfully to program development activities.

Techniques for portraying parallel behavior graphically have been the focus of a number of recent research efforts [18, 20]. To date, however, little attention has been given to the problem of how debugging tools should support interaction with the user. Existing breakpoint-style debuggers (e.g., Intel's IPD [10], CONVEX's CXdb [1], or Sequent's Pdbx [23]) rely on extensions of serial debugger technology. The user manually specifies where execution should be halted or monitored, typically through breakpoints (positions in the instruction stream where processing should halt), watchpoints (data elements whose values should be monitored, with execution halting when the value is touched or if a specific condition is met), and/or tracepoints (instruction or data locations whose access should trigger generation of a message). Trace-based tools, as the term indicates, rely on just the tracepoint mechanism (e.g., the trace analysis facilities of SCHEDULE [2], GMAT [22], IBM's Parallel Fortran [8], CONVEX's CXpa [6], or Paragraph [7]); during execution, messages are logged to a trace file for real-time or post-mortem analysis. The disadvantage of this approach is that the user cannot interact with or alter program execution On the other hand, the software hooks required to implement tracing are relatively straightforward, and can be inserted automatically by the compiler (e.g., CXpa) or in a preprocessing step (e.g., SCHEDULE).

Regardless of the mechanism used, the user is confronted with an all-or-nothing support situation. If monitoring is controlled by automatic instrumentation or features in the run-time library, copious amounts of data are generated, much of which may be irrelevant to the programming task at hand. In contrast, manual control over monitoring requires that the user specify where information should

[1] A portion of this research was conducted using the Cornell National Supercomputer Facility, a resource of the Center for Theory and Simulation in Science and Engineering, which receives major funding from the National Science Foundation and IBM Corporation. The work was carried out as part of a joint development study with IBM Palo Alto Scientific Center

be gathered; this entails predicting what data will be useful and at what locations, and then either adding new statements to the program (which will need to be eliminated later) or issuing commands at run-time (which may be difficult to duplicate in a subsequent session).

This paper suggests a compromise approach, whereby the user and tool collaborate to establish an optimal level of instrumentation for a given program and tracing task. The user indicates very generally the type of information desired and the areas of the program for which trace records should be generated, by annotating the block-structured organisation already present in the source code. Since the specifications are tied unambiguously to program structure, the appropriate software hooks can then be inserted by a compiler, preprocessing tool, or the debugger itself.[1] Although the techniques are described in terms of tracing tools, they could also be employed in breakpoint-style debuggers if rudimentary source-code analysis facilities were available.

The proposed strategy explots three concepts which have been largely neglected in the past, but could go a long way in making debugging tools more palatable to the user community:

- Flexible ways to limit the potentially huge amounts of data generated during execution of a scientific application.
- Clear correlation of dynamic/multi-stream behavior with the static/single-stream program manipulated by the user.
- Adaptation to changing requirements during the program development cycle.

The scheme is based on user-defined *event regions*, used to establish the portions of execution during which events are reported, and *event levels*, which determine the types of events to be monitored within a region. The two orthogonal controls interact to provide flexible control over monitoring. The advantages of this approach derive from its clear relationship to program structure. The model for specifying trace output matches that used for program code, so it is easier for the programmer to arrive at a useful trace. The reduction in the number of event records also makes trace interpretation faster.

The discussion begins with an analysis of the requirements for program behavior information at different points in the development cycle. This establishes the need for independent levels of trace support, outlined in the next section. The section which follows describes how the scope of tracing can be varied to fit cyclic patterns of debugging and program analysis. By way of example, the region and level mechanisms are applied to a program written in PCF Fortran [14].

Requirements for Program Behavior Information

As shown in Figure 1, the development cycle for parallel applications typically begins with a correct serial version [16]. The programmer has a general idea of which portions of the program might be performed in parallel, but it is not always clear if parallelisation will be cost-effective. With the help of a profiling tool or hand-coded instrumentation, timing statistics are gathered to determine which of those areas are sufficiently compute-intensive to warrant the effort of restructur-

[1] This could also be accomplished through postprocessing (e.g., application of a filter to the trace file). As the effects would be transparent, no specific implementation mechanism is described here.

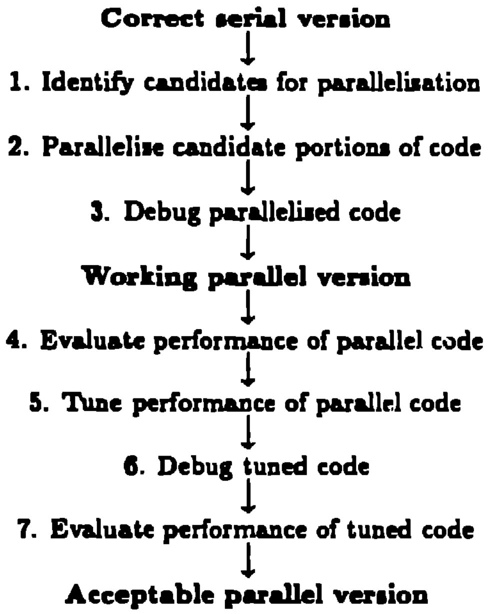

Correct serial version
1. Identify candidates for parallelisation
2. Parallelise candidate portions of code
3. Debug parallelised code
Working parallel version
4. Evaluate performance of parallel code
5. Tune performance of parallel code
6. Debug tuned code
7. Evaluate performance of tuned code
Acceptable parallel version

Figure 1. Development Cycle for Parallel Scientific Applications

ing. Parallelisation then begins. As new structures are added to the program and old ones modified, the code is tested to determine if the results match those obtained from the serial baseline. When they diverge, a period of cyclic debugging intervenes. This alternation of testing and debugging is necessary even when software tools have been used to guide parallelisation activities. Once a functional parallel version hrs been achieved, its performance can be tuned to maximise speedup. The tuning process often results in the discovery of additional bugs, precipitating new bouts of debugging activities. Eventually, the programmer is satisfied that further improvements are impossible or unprofitable.

Execution tracing, as a source of dynamic information on program behavior, is potentially useful at all stages in the developr ent cycle. Although certain steps are repeated more than once (as shown in Figure 1), they may Le grouped into four categories of activities: performance profiling, debugging, benchmarking, and performance tuning.

Prior to initiating parallelisation, the programmer needs a high-level *profile* of computational activities in order to determine where to focus efforts. The principal requirement here is timing information, which can be used to confirm or contradict intuitive notions of program hot-spots. As a minimum, entry to and exit from all user-supplied program units should be reported so that timing statistics can be calculated and compared.

As parallelism is introduced, run-time errors will surfece. In *debugging*, the primary concern i. to determine where program behavior does not match that expected by the programmer. Because the program is thought of as a sequence of manipulations on data structures, such as multi-dimensional arrays, the programmer assures correctness by tracking changes to those structures. In parallel sections of code, this activity takes on an added dimension: tracking the order in which parallel processes access the data. Not only must value changes be noted, but also the source of each change (i.e., which process made it and at what point in its activities). Determining access order often entails the analysis of synchronisation events, such as which process entered a critical section last.

Once all obvious bugs have been eliminated, tracing can be used to determine the effectiveness of parallelisation efforts in terms of performance. *Benchmarking* requires a finer granularity than subprogram profiling. Activities within the parallelised section of code are timed, to verify that parallelism has achieved some degree of speedup and to ascertain the possibilities for further improvement. Programmers are concerned with quantifying the execution cost or benefit of each parallelising transformation. Moreover, they draw a distinction between the system overhead involved in starting up and terminating process (referred to here as *system costs*) versus that incurred when processes are idle because of barrier waits, failure to obtain locks, etc. (*waiting costs*). The former represents the fixed costs associated with parallelism, while the latter can be manipulated — at least indirectly — by the programmer.

During *tuning*, the primary concern is to identify situations which can be improved by code manipulation. The programmer needs detailed information on load balancing: the order in which work is distributed, time required to distribute shared data, time spent by each process at a barrier, etc. Since the programmer must rely on this data to fine-tune the degree of parallelism, the specifics of which work (i.e., which loop iterations or other subtasks) was assigned to each process is also important. Finally, as tuning modifications are made to the source code, additional benchmarking is needed to verify that the timings improved or to compare the effects of different tuning strategies.

Matching Trace Information to Programming Activities

A recent survey of the trace facilities available with IBM's Parallel and Clustered Fortran compilers [8, 9] revealed that users are remarkably unaware of the potential of parallel program traces [21]. Many programmers, for example, who employed traces for benchmarking or performance tuning activities had never considered using them to isolate program errors. Others underestimated their reporting capabilities, resorting to hand-coded instrumentation to acquire data already available (albeit obscured) in the trace files. This situation results in a great deal of unnecessary programmer effort and may introduce new sources of error which are extremely difficult to isolate.

The extremely large quantities of data generated for a full program trace are daunting to most programmers. In some cases, there are mechanisms available to reduce trace volume; CXpa, for example, allows selective profiling at the routine, loop, or parallel region level [6], while IBM's trace facility offers nine levels in a number of permutations [8, 9]. Users claim, however, that the mechanisms are unusable, either because they are inappropriate for the need at hand or because their use is incomprehensible or inconsistent. Moreover, the type of information reported in most traces reflects the requirements of systems programmers, not scientific users. Much of the data reflects system factors that are irrelevant to program development, while common programming needs are left unsatisfied. Consequently, existing tools are under-utilised and under-valued by the user community.

How can the situation be improved? The first step is to organise the type of data reported in order to correspond with typical programming activities. In our block-structured approach, the type of trace records generated is controlled through *trace levels*. A level defines which execution-time events are of interest and should be reported; it therefore functions as a masking mechanism to reduce the amount of trace output. We propose five levels, reflecting the most common uses for traces:

- to establish timings for entry to and exit from subprogram units (PROFILE)
- to isolate the portion of the program where an error has occurred (DEBUG1)
- to identify the error and determine the efficacy of repairs (DEBUG2)
- to tune program performance for maximum efficiency (TUNE)
- to benchmark and compare program performance (BENCHMARK)

Normally, one level will apply to the entire program, reflecting the activity in which the programmer is engaged, be it debugging, tuning, or performance analysis. In some cases, however, it may be desirable to combine multiple levels during a single execution. The effects of each level are described in relation to typical parallel language constructs, amplified by the concept of user-defined trace messages (arbitrary text emitted in the trace file at the specification of the user).

The results of applying levels are illustrated by a brief program for the computation of π with the rectangle rule (Figure 2), written in PCF Fortran [14] and adapted from the example in [11]. The trace output shown is generalised and does not reflect any particular trace format. The columns present timestamp, process ID, source code location, and minimal messages, respectively; such information is compatible with most existing formats, as well as the suggestions for a standardised trace format summarised in [17].

PROFILE: This level results in a minimal number of trace records (Figure 3). It is intended primarily for summarising the amount of time spent in each program unit (main program/subroutine/function, or finer-grained blocks of code), as an indication of where parallelisation or improvement efforts should be directed. The flow of program control into and out of each unit is reported in the trace file. User-defined trace messages may identify the organisation of logical activities within a unit, so these are recorded as well.

DEBUG1: This also results in a restricted number of trace records (Figure 3), and is particularly useful during initial attempts to localise a program error. Only events marking the very general progress — or lack of progress — of parallelism are reported. Thus, the user is able to obtain an overview of which portions of the program executed and in what general order they occurred.

For parallel loops, the trace records include each process's arrival at the start and end of the construct, plus any waits caused by unsuccessful attempts to enter critical sections. Similar information is reported for parallel sections, except that waits occur due to the explicit ordering of sibling sections. User manipulation of synchronisers (such as lock and event variables) is also reported in terms of unsuccessful attempts which resulted in waits. This information gives the programmer an extremely rough idea of the extent to which contention may be affecting program behavior. Subroutine-level parallelism is also traced in terms of coarse-grained activities: the start and end of each process's work, and the satisfaction of barrier synchronisation. Access to shared variables is reported only in the most general way, via lists identifying which ones were accessed by each process. Entry to and exit from subprogram (whether the invocations were serial or in parallel) continue to

```
      PROGRAM PI
      DO I=1,3
          READ(*,*) NRECS
          CALL INTEG(NRECS,RN)
          WRITE(*,*)'Number of rectangles:',NRECS
          WRITE(*,*)'Number of processes available:',NPRTOT
          WRITE(*,*)'Approximation:',RN
      END DO
      END

      SUBROUTINE INTEG(N,SUM)
      GATE ADDUP GUARDS(SUM)
      SUM = 0.0
      UNLOCK(ADDUP)
C parallel region and scoping declarations
      PARALLEL
          PRIVATE(PSUM,H,X)
C parallel initializations (redundantly executed, once per process)
          PSUM = 0.0
          H = 1.0/N
C parallel work (groups of iterations executed by each process)
          PDO INDEX=1,N
              X = (INDEX-0.5)*H
              PSUM = PSUM + 4 0/(1.0+X*X)
          END PDO
C reduction executed once per process and one process at a time
          CRITICAL SECTION (ADDUP)
              SUM = SUM + H*PSUM
          END CRITICAL SECTION (ADDUP)
      END PARALLEL
      RETURN
      END
```

Figure 2. Example PCF-Fortran Program

be traced in order to indicate the general flow of program control. User-defined trace messages are recorded as well.

DEBUG2: Like DEBUG1, this level is intended to facilitate the isolation and correction of program errors. It provides the level of detail most likely to reveal the sources of behavioral anomalies (Figure 4), but does not include performance-related information. Since DEBUG2 has the potential for generating considerable volume, it will be most useful when restricted to small portions of the program, such as those suspected (through analysis of previous DEBUG1-level output) of containing anomalies or those where code modifications have been made.

Tracing for a parallel construct reflects its progression through execution: construct entry, privatisation of variables, start of each process's work, assignment of iteration groups or sections, end of each process's work, and construct exit when the barrier is satisfied. When critical section occurs, detailed information on this is reported as well, including successful and unsuccessful attempts to

```
          PROFILE level                                      DEBUG1 level

00000000 1    1  BEGIN PROGRAM               00000000 1    1  BEGIN PROGRAM
00000041 1   10  ENTER INTEG                 00000041 1   10  ENTER INTEG
00000249 1   26  EXIT INTEG                  00000042 1   14  SHARED (SUM,ADDUP)
00000321 1   10  ENTER INTEG                 00000042 2   14  SHARED (SUM,ADDUP)
00000549 1   26  EXIT INTEG                  00000043 3   14  SHARED (SUM,ADDUP)
00000630 1   10  ENTER INTEG                 00000047 2   18  BEGIN PDO
00000812 1   26  EXIT INTEG                  00000048 1   18  BEGIN PDO
00000896 1    9  END PROGRAM                 00000048 3   18  BEGIN PDO
                                             00000202 1   21  END PDO
                                             00000203 3   21  END PDO
                                             00000204 3   22  WAIT CRIT SECT
                                             00000225 2   21  END PDO
                                             00000249 1   26  EXIT INTEG
                                             00000321 1   10  ENTER INTEG
                                                        ...
                                             00000896 1    9  END PROGRAM
```

Figure 3. Trace Output for PROFILE and DEBUG1 Levels

obtain access, as well as exit from the section. The level of detail is similar for parallel sections, except that process suspension and resumption, due to ordered execution, is reflected. All user-defined synchroniser operations are now reported in the trace, whether or not a delay was involved. Thus, the creation, termination, and freeing of a lock are reported as well as attempts to gain control of it. This fine level of granularity allows the programmer to observe every transaction on synchronisers. Subroutine-level parallelism is also traced at the lowest level manipulatable by the programmer: process creation and termination, start and end of work, arrival at barriers, and barrier satisfaction. Updates and accesses to shared data are reported in terms of the value assigned or read. Finally, subprogram entry/exit and user-defined trace messages are still recorded.

TUNE: Unlike the DEBUG levels, TUNE is intended for programs which function correctly (or appear to function correctly). This level reports on program performance (Figure 4), specifically those aspects of performance which can be tuned by the programmer to achieve maximum efficiency. Its focus, therefore, is the "variable" overhead due to poor load balancing, lock contention, etc. Information on the "fixed" costs incurred by the system during process initiation and cleanup will be reported at the BENCHMARK level.

The events of interest for parallel loop and cases constructs include the start of the construct, start of each process's work, assignment of iteration groups or cases, termination of each process's work, and end of the construct. From this information, the programmer (or a trace analysis tool) can determine to what extent "slow" or improperly balanced processes are provoking long barrier waits. He or she can also observe the effects of attempts to tune loop/sections performance by controlling iteration groups, etc. When the construct includes synchronisation constructs (critical section or ordered case execution), this is traced too, as described below for synchronisers. The record produced for subroutine-level parallelism include the start and end of each process's work, arrival at barriers, and barrier satisfaction. In addition, the distribution of shared data is reported

```
00000000 1   1  BEGIN PROGRAM
00000041 1  10  ENTER INTEG                 00000210 3  22  ENTER CRIT SECT
00000042 1  11  GATE (ADDUP)                00000211 1  25  WAIT BARRIER
00000044 1  12  SHARED (SUM = 0.0)          00000214 3  23  SHARED (SUM = 2.0135)
00000044 1  13  UNLOCK (ADDUP)              00000217 3  24  EXIT CRIT SECT
00000045 1  15  PRIVATE (PSUM,N,X,INDEX)    00000218 3  25  WAIT BARRIER
00000047 2  18  BEGIN PDO (INDEX = 1,10)    00000225 2  21  END PDO
00000048 1  18  BEGIN PDO (INDEX = 11,20)   00000227 2  22  ENTER CRIT SECT
00000048 3  18  BEGIN PDO (INDEX = 21,30)   00000231 2  23  SHARED (SUM = 3.1416)
00000176 2  18  BEGIN PDO (INDEX = 31,33)   00000237 2  24  EXIT CRIT SECT
00000202 1  21  END PDO                     00000238 2  25  WAIT BARRIER
00000203 1  22  ENTER CRIT SECT             00000242 1  25  PASS BARRIER
00000203 3  21  END PDO                     00000249 1  26  EXIT INTEG
00000206 1  23  SHARED (SUM = 0.7880)       00000321 1  10  ENTER INTEG
00000207 3  22  WAIT CRIT SECT                         ...
00000209 1  24  EXIT CRIT SECT              00000896 1   9  END PROGRAM
```

Figure 3. Trace Output for DEBUG2

so that the programmer can observe the delays associated with data distribution.

For user-defined synchronisers, tracing at this level reports all accesses, but not creation/termination (which cannot be tuned for efficiency). Successful and unsuccessful attempts to obtain locks, lock releases, event posting, and event waits are included. The programmer thus can observe first-hand the causes and costs of synchroniser contention. Entry to and exit from functions and subroutines are not reported at this level, but user-defined trace messages are included for the convenience of programmers who use this technique to mark or measure general program activities.

BENCHMARK: The benchmarking level is intended to provide information that will be useful in the analysis of system (as opposed to program) performance. Its events report on systems-related overhead such as process start-up time. The data will also be of interest to programmers who wish to compare the performance of alternative program versions in detail — for example, to determine where the cost breakoff point is between loop-level and subroutine-level parallelism for a particular section of code.

Tracing for parallel loop or cases constructs now reflects the system startup time incurred between entry to the construct and the initiation of process work, as well any lag time between the arrival of the last process at the barrier and final barrier satisfaction. The full set of trace records therefore includes construct start, process creation, start of process's work, end of process's work, and each process's arrival at construct end. For user-defined processes and subroutine-level parallelism, tracing records the system overhead for process management activities. These include the amount of time spent originating and terminating processes, as well as the time elapsed between arrival of the last process at a barrier and barrier satisfaction. The tracing of lock and event synchronisations is identical to that performed under TUNE, since it allows the determination of how much system overhead time elapses between, say, the release of a lock and the re-activation of a waiting process. Again subprogram entry/exit are ignored, but any user defined trace messages are reported.

```
                TUNE level

00000000 1   1  BEGIN PROGRAM
00000042 1  12  SHARED (SUM,ADDUP)
00000045 1  15  PRIVATE (PSUM,H,X,INDEX)
00000047 2  18  BEGIN PDO (INDEX = 1,10)
00000048 1  18  BEGIN PDO (INDEX = 11,20)
00000048 3  18  BEGIN PDO (INDEX = 21,30)
00000176 2  18  BEGIN PDO (INDEX = 31,33)
00000202 1  21  END PDO
00000203 1  22  OBTAIN (ADDUP)
00000203 3  21  END PDO
00000207 3  22  TRY (ADDUP)
00000209 1  24  RELEASE (ADDUP)
00000210 3  22  OBTAIN (ADDUP)
00000211 1  25  WAIT BARRIER
00000217 3  24  RELEASE (ADDUP)
00000218 3  25  WAIT BARRIER
00000225 2  21  END PDO
00000227 2  22  OBTAIN (ADDUP)
00000237 2  24  RELEASE (ADDUP)
00000238 2  25  WAIT BARRIER
00000242 1  25  PASS BARRIER
00000322 1  12  SHARED (SUM,ADDUP)
           ...
00000896 1   9  END PROGRAM
```

```
                BENCHMARK level

00000000 1   1  BEGIN PROGRAM
00000042 1  12  SHARED (SUM,ADDUP)
00000044 3  14  BEGIN PARALLEL
00000045 3  15  PRIVATE (PSUM,H,X,INDEX)
00000046 1  18  BEGIN PDO
00000047 2  18  DISPATCH PDO
00000048 1  18  DISPATCH PDO
00000048 3  18  DISPATCH PDO
00000170 2  21  COMPLETE PDO
00000176 2  18  DISPATCH PDO
00000196 1  21  COMPLETE PDO
00000197 3  21  COMPLETE PDO
00000202 1  21  DONE PDO
00000202 1  22  OBTAIN (ADDUP)
00000203 1  22  ENTER CRIT SECT
00000203 3  21  DONE PDO
00000206 3  22  TRY (ADDUP)
00000207 3  22  WAIT CRIT SECT
00000208 1  24  RELEASE (ADDUP)
00000209 1  24  EXIT CRIT SECT
00000209 3  22  OBTAIN (ADDUP)
00000210 3  22  ENTER CRIT SECT
00000210 1  25  TEST BARRIER
00000211 1  25  WAIT BARRIER
           ...
00000896 1   9  END PROGRAM
```

Figure 4. Trace Output for TUNE and BENCHMARK

Restricting the Scope of Analysis Information

Tracing levels alone will not reduce to manageable proportions the amount of trace data generated by scientific applications. Researchers at CONVEX, for example, found that a 10-minute program run generated 1.3 gigabytes of profiling statistics [6]. Organizing levels in terms of program development activities decreases the number of records that are extraneous to the task at hand, but it should be clear that large traces will still result.

One aspect of program development that merits closer attention in this respect is the hierarchical approach employed by most users. Empirical studies suggest that programmers "funnel in" on the code, starting with a high-level view of overall program behavior and progressively moving to more specific levels of detail [4, 5]. This procedure, which allows the programmer to put off complex issues as long as possible, mimics the top-down approach to program development. Take, for example, the way hand-coded instrumentation is added to a program to detect the source of an error (Figure 2). The programmer first investigates general behavior at the level of subprogram units. The focus is then narrowed to a particular block of code. Finally, code modification is performed at the level of individual statements. A similar procedure is followed for benchmarking and performance

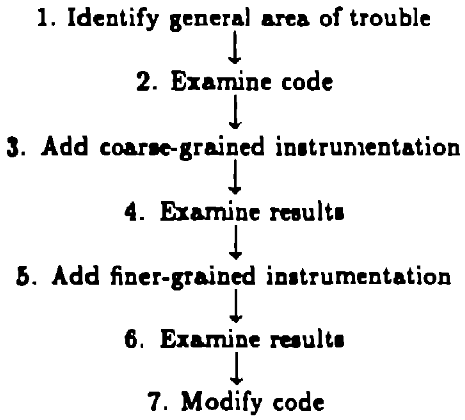

1. Identify general area of trouble
2. Examine code
3. Add coarse-grained instrumentation
4. Examine results
5. Add finer-grained instrumentation
6. Examine results
7. Modify code

Figure 2. Hierarchical Approach in Hand-coded Debugging

improvement activities. In pre-improvement benchmarking, for example, the first order of business is determining which subprogram units account for the greatest proportion of execution time. Within those units, analysis is then refined to pinpoint the areas which have the greatest potential for yielding improvements.

To support this approach, a second mechanism interacts orthogonally with the trace level controls. *Trace regions* limit the scope of tracing, or the period of time during which event records are generated. Because the program already represents a block-structured expression of problem logic, it makes sense that tracing scope relate directly to source code organisation. A region, therefore, corresponds to a subprogram unit (SUBPROGRAM and IGNORE controls), a block construct (CONSTRUCT), or an arbitrary area (BEGIN and END). The first three control static (lexical) scope, while the other two delimit dynamic regions. The number and nature of the regions were established through extensive interviews with scientific users [21].

Each type of region is described below. For convenience, the controls are shown in the form of compiler or preprocessor directives. It is intended, however, that regions be specified graphically through the use of a program editor or other interactive tool. Facilities for highlighting regions with shading or color will allow the user to pinpoint the areas of interest quickly and accurately. They will also emphasise the distinction between "step-over" (static) and "step-down" (dynamic) tracing of subordinate program modules.

SUBPROGRAM: The programmer uses this region to indicate interest in a particular subprogram or portions thereof. Tracing will be active during the execution of all statements within the region (in this case, after the occurrence of the T$SUBPROGRAM directive). Its effect is limited to the immediate static (lexical) scope; that is, tracing is deactivated at calls to subordinate functions or subroutines. For example, the region defined in Figure 6 begins in the middle of the subroutine and encompasses all subsequent statements, but does not "step down" to include the code executed by the invocation of INITGLX.

CONSTRUCT: This region provides finer granularity than SUBPROGRAM, corresponding to the execution of a program block. Block constructs include all block-structured elements in the language, but typically only parallel blocks (e.g., parallel loops and sections) are of interest for tracing. Since the programmer uses CONSTRUCT to indicate interest in a particular construct or

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$SUBPROGRAM
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
      ...
      RETURN
      END
```

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$CONSTRUCT
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
      ...
      RETURN
      END
```

Figure 6. Examples of SUBROUTINE and CONSTRUCT Regions.

group of constructs, its effect is limited to the immediate static scope. In the example of Figure 6, tracing begins just prior to execution of the PARALLEL DO and continues until the loop has terminated; it is deactivated during the invocation of INITGLX.

BEGIN and END: The user can also define arbitrary regions that are not restricted to construct or subprogram boundaries, and that reflect the dynamic flow of program control through subprograms. A BEGIN/END region effectively toggles tracing on and off, as shown in Figure 7. Note that in this case, trace records are generated from the start of the parallel loop until after the UNLOCK operation, including during all subprograms invoked within the scope of the region (INITGLX and any subordinates it might have). Due to the nesting of subprograms during execution, a previous BEGIN/END region may be active when a new region is encountered, although it will be more common that regions are closed for the duration of subordinate routines, as described below.

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$BEGIN
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
C$ T$END
      ...
      RETURN
      END
```

```
      SUBROUTINE INIT(SUM,N)
      ...
         PSUM = INITGLX(IPLANE,PSUM)
      ...
      END

      SUBROUTINE INITGLX(I,SUM)
C$ T$IGNORE
      PARALLEL DO J=1,I
         ...
      END PARALLEL DO
      RETURN
      END
```

Figure 7. Examples of BEGIN/END and IGNORE Regions.

IGNORE: Because user-delimited regions transcend invocation boundaries, they have the potential for generating considerable amounts of trace data. An IGNORE region therefore offers a convenient mechanism for temporarily closing a region for the duration of a subprogram. By spec ifying that a subprogram should be ignored, the programmer disables all tracing at that level of invocation; tracing is resumed after return to the caller. In Figure 7, any region which was open at the calling site to INITGLX will be temporarily closed during execution of that subroutine. The scope of the IGNORE region is static, so tracing will again become active within any of its subordinate routines. The effects of this region are antithetical to those of SUBPROGRAM; where SUBPROGRAM initiates statically-scoped tracing at the indicated point and continues until the end of the subprogram unit, IGNORE disables tracing for the same area.

It is also possible to combine regions of different types. Their interaction provides a tight control over exactly which portions of code are traced. Returning to the CONSTRUCT region in Figure 6, for example, the specification of a SUBPROGRAM region containing INITGLX would have the effect of suppressing all records except those in the subroutine or in the parallel loop.

Conclusions

Structured programming techniques offer the scientific programmer ways to make source code structure reflect the underlying design logic. As a result, it has become commonplace for users to apply cyclic and hierarchical approaches in code development. Block-structured tracing capitalises on this observation. It allows the programmer to control the number and type of run-time events in a structured fashion that reflects both source code organisation and changing requirements during the program development cycle.

The cyclic and hierarchical approaches interact throughout the parallel program cycle. Most programmers develop or parallelise their applications one section at a time. A full cycle — converting code to parallel form, testing and debugging it, benchmarking the results, then fine-tuning it it to achieve the best possible performance — is applied to a subportion of the program. Once it is complete, the programmer moves on to another area, typically returning only if a latent bug emerges or if later work generates a new idea for performance improvement. This suggests that tracing tools should provide separate controls for (a) indicating the program area of current interest and (b) identifying what type of information should be reported for that area. The controls should be easy to specify and easy to change.

The orthogonal trace and region mechanisms provide direct support for this approach. Hierarchical patterns indicate that at any stage during program development, a single trace region or collection of trace regions is likely to be of interest for a length of time. For that reason, the region mechanism is potentially fine-grained, while level provides a simple, coarser control. Cyclic patterns, on the other hand, indicate that varying collections of trace data will be desired for the region as the user progresses through different programming tasks. By organising trace events according to typical activities, the level mechanism eliminates the tedium of discarding irrelevant records and clarifies the contribution of each record type. Together, the two controls interact to make parallel debugging tools easier and more effective for user applications.

References

[1] CONVEX Computer Corporation. 1991. *Convex CXdb User's Guide*. Convex Press, publication DSW-473.

[2] Dongarra, J. J. and D. C. Sorensen. 1987. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In *The Characteristics of Parallel Algorithms*, Jamieson, *et al.*, eds. MIT Press.

[3] Fox, G. C. 1989. Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech. *Concurrency: Practice & Experience*, 1 (1): 94.

[4] Gould, J. D. 1975. Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies*, 7: 151–182.

[5] Gould, J. D. and P. Drongowski. 1974. An Exploratory Study of Computer Program Debugging. *Human Factors*, 16 (3): 258–277.

[6] Hansen, G. J., C. A. Linthicum and G. Brooks. 1990. Experience with a Performance Analyzer for Multithreaded Applications. *Proceedings of Supercomputing '90*, pp. 124–131.

[7] Heath, M. T. and J A. Etheridge. 1991. *Visualizing Performance of Parallel Programs*. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory.

[8] IBM Corporation. 1990. *Clustered Fortran Language and Library Reference*. IBM Corporation, publication SC23-0523-0.

[9] IBM Corporation. 1988. *Parallel Fortran Language and Library Reference*. IBM Corporation, publication SC23-0431-0.

[10] Intel Supercomputer System. 1991. *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual*. Intel Corporation, publication 312043-001.

[11] Jakob, R. 1990. Parallel Programming Models for Shared Memory Multiprocessors. M.S. Thesis, Department of Electrical and Computer Engineering, University of Colorado.

[12] McDowell. C. E. and D. P. Helmbold. 1989. "Debugging Concurrent Programs." *ACM Computing Surveys*, 21 (4): 593–622.

[13] Moriarty, K. J. M. 1989. Parallel Processing of Large-Scale Applications on Powerful Multiple Processors. *International Journal of Supercomputer Applications*, 3 (1): 82-87.

[14] Parallel Computing Forum. 1990. *PCF Fortran, Version 3.1*. (Prepared for ANSI X3H5)

[15] Pancake, C. M. 1991. Software Support for Parallel Computing: Where Are We Headed? *Communications of the ACM*, 34 (11): 52-64.

[16] Pancake, C. M. and D. Bergmark. 1990. Do Parallel Languages Respond to the Needs of Scientific Programmers? *IEEE Computer*, 23 (1): 13-23.

[17] Pancake, C. M., D. Gannon, S. Utter and D. Bergmark. 1991. Supercomputing '90 BOF Session on Standardizing Parallel Trace Formats. Technical Report CTC91-TR53, Cornell Theory Center.

[18] Pancake, C. M. and S. Utter. 1991. Debugger Visualizations for Shared Memory Multiprocessors. In *High-Performance Computing II*, ed. M. Durand and F. El Dabaghi, North Holland, pp. 145-158.

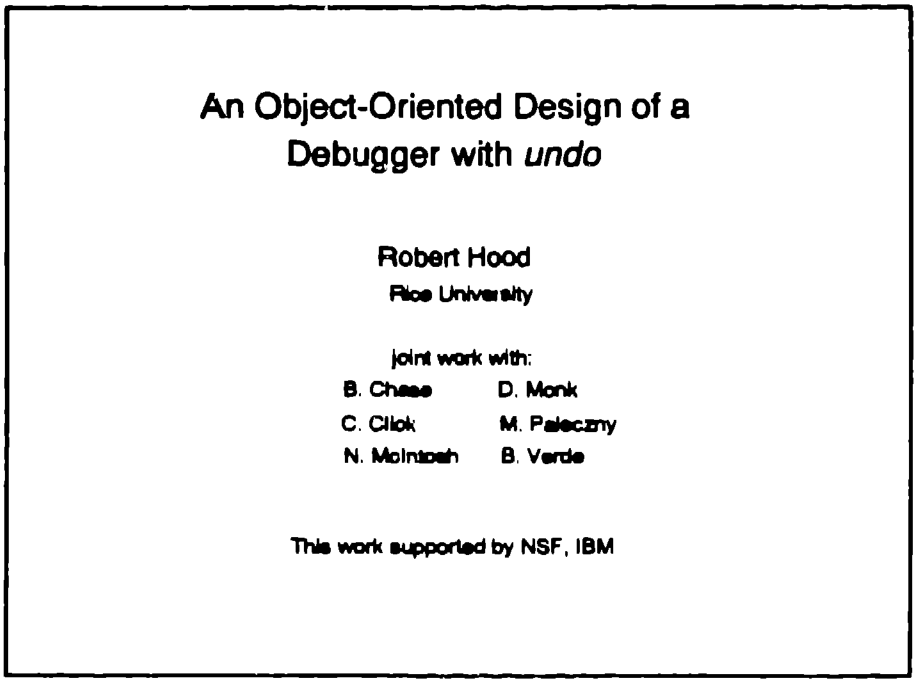
[19] Pancake, C. M. and S. Utter. 1991. "A Bibliography of Parallel Debuggers - 1990 Edition," *ACM SIGPLAN Notices*, 26 (1): 21-37. [The bibliographic database is available in electronic form through the Cornell National Supercomputer Facility. For information, contact the author.]

[20] Pancake, C. M. and S. Utter. 1989. Models for Visualisation in Parallel Debuggers. *Proceedings of Supercomputing '89*, pp. 627-636.

[21] Pancake, C. M. and S. Utter-Honig. 1991. Improving the Effectiveness of the Parallel Fortran Trace Facility. Internal report prepared for IBM Palo Alto Scientific Center.

[22] Seager, M. K. *et al.*. 1989. *Graphical Multiprocessing Analysis Tool (GMAT)*. Technical Report UCID-21345, Lawrence Livermore National Laboratory.

[23] Sequent Computer Corporation. 1986. *Pdbx Parallel Debugger for Sequent Systems*. Sequent Technical Publications.

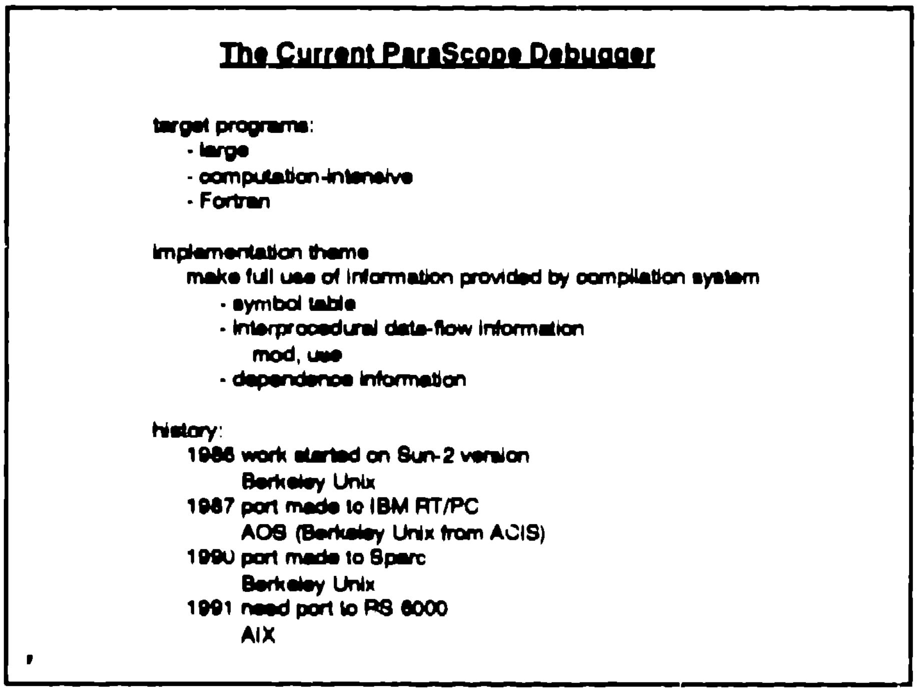


An Object-Oriented Design of a Debugger with *undo*

Robert Hood
Rice University

joint work with:
B. Chase D. Monk
C. Click M. Paleczny
N. McIntosh B. Verde

This work supported by NSF, IBM



The Current ParaScope Debugger

target programs:
- large
- computation-intensive
- Fortran

implementation theme
make full use of information provided by compilation system
- symbol table
- interprocedural data-flow information
  mod, use
- dependence information

history:
1986 work started on Sun-2 version
Berkeley Unix
1987 port made to IBM RT/PC
AOS (Berkeley Unix from ACIS)
1990 port made to Sparc
Berkeley Unix
1991 need port to RS 6000
AIX

Goals of New Debugger

portability

simplify process of accommodating new:

- architectures
- operating systems
- source languages
- compilers

also consider:

i.e., make sure design is flexible enough to permit

- remote execution capability
- multiple-threads in target
- run-time code changes
- extensibility

undo

make it possible to undo any debugger operation, even a `continue`

- design for arbitrary number of "checkpoints"
- tune implementation to find feasible maximum

save

put copy of debugging session in file system for later resumption

User's View

the debugger is an *editor*

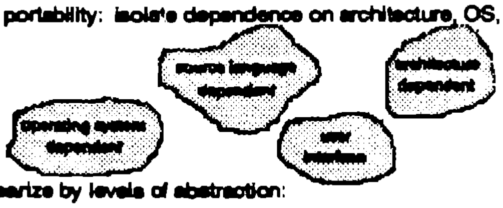
- permits us to manipulate the *execution* of a program
- supports operations:

```
stop at, continue, step, print, ...
```

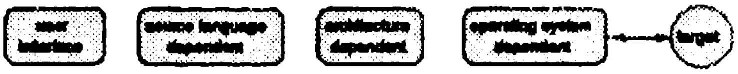
```
execution
   program
      file*
         list()
         function*
            stopIn()
            line*
               stopAt()
   state
      where()
      frame*
         prev(), next(), pcInFrame()
         param*
         local*
            print(), set()
```

How to Implement User Objects

for portability: isolate dependence on architecture, OS, ...



linearize by levels of abstraction:



what functionality should be at each level?
OS level should not have architecture dependences (if possible)
thus, would not know about *frames*

problem: *registers*

problem: *breakpoints on Sparcs*
breakpoints must be single-stepped over
no single-step in architecture: must use a breakpoint

The Operating System Layer

- contains all OS dependent code

- provides abstract access to a 'process'

```
continue()
wait()
kill()
pause()
stepWithTraceBit()     if possible
readMem(), writeMem()
readReg(), writeReg()
```

knows about *PC* , *SP*
but not *FP*

components:

```
- AddressSpace
- RegisterSet
- unix process ID
```

The Architecture Layer

contains (most of) architecture-dependent code

provides abstract access to an "instrumented process"

```
step()
wait()
continue()   (steps over breakpoint at PC, if necessary)
```

knows about

- instrumentation
  - breakpoints `insert(), delete(), ...`
  - patches
- stack frames
  - `next(), prev(), param(#), ...`

components:

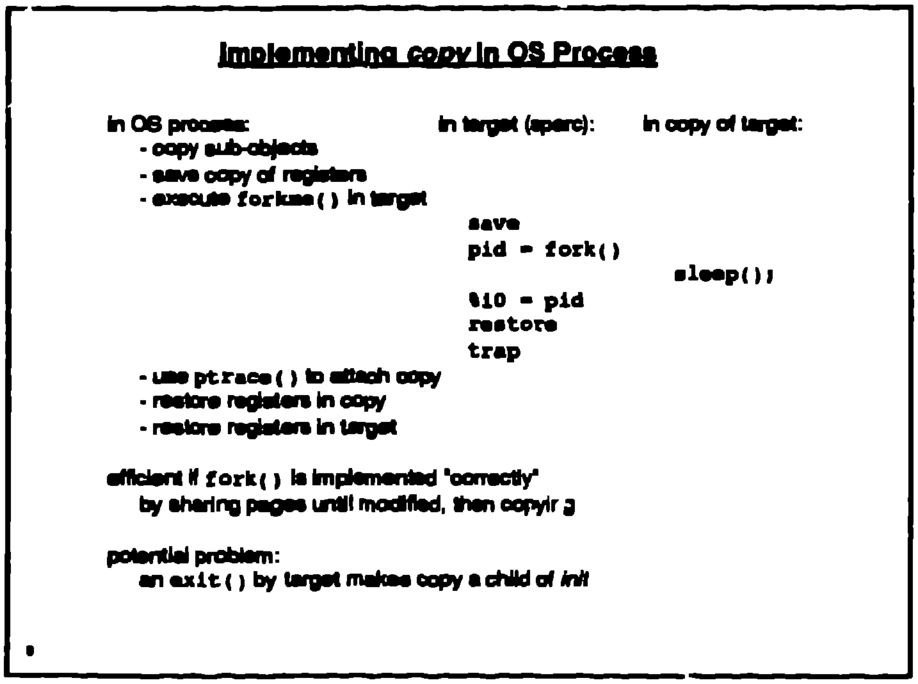
- list of stack frames
- list of breakpoints
- list of patches
- an OS process object

Implementation Strategy

use InterViews for graphical user interface
  has implications for change notification

use NIH object library

- base our abstractions on NIH `Object` class
  - for each of our abstractions must implement:
    - `copy(), save(), restore(), ...`
- get many data abstractions in return
  - sets, sequences, strings, ...
- `undo`, `save` fall out (mostly)
  - at editor level:
    - before performing next operation
      - copy top-level components
      - sub-components copied in turn

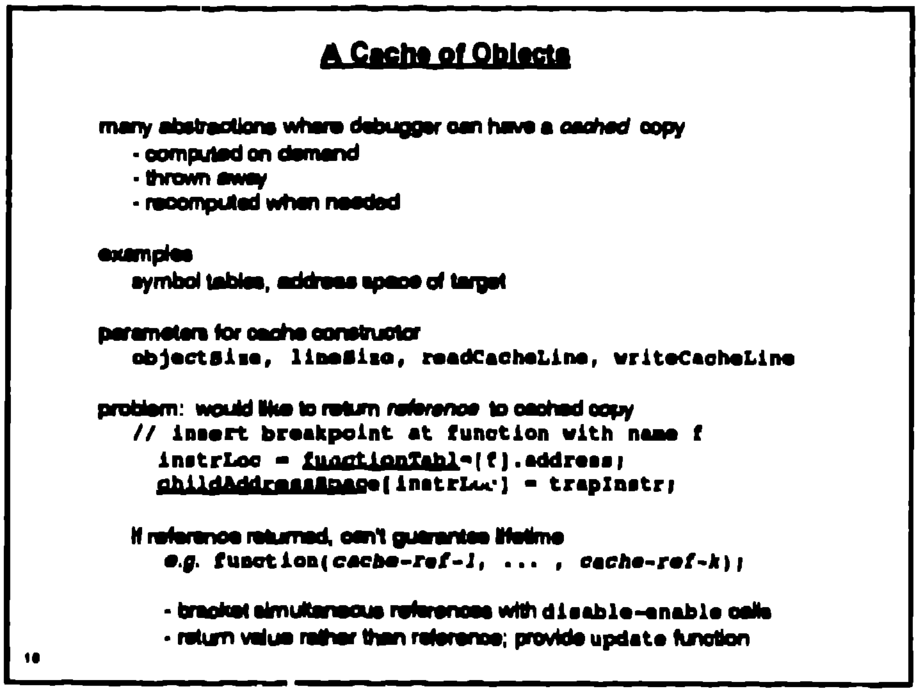


Implementing *copy* in OS Process

in OS process:	in target (sparc):	in copy of target:
- copy sub-objects		
- save copy of registers		
- execute `forkme()` in target		
	`save`	
	`pid = fork()`	
		`sleep();`
	`%i0 = pid`	
	`restore`	
	`trap`	
- use `ptrace()` to attach copy		
- restore registers in copy		
- restore registers in target		

efficient if `fork()` is implemented "correctly"
by sharing pages until modified, then copyir g

potential problem:
an `exit()` by target makes copy a child of *init*



A Cache of Objects

many abstractions where debugger can have a *cached* copy

- computed on demand
- thrown away
- recomputed when needed

examples
symbol tables, address space of target

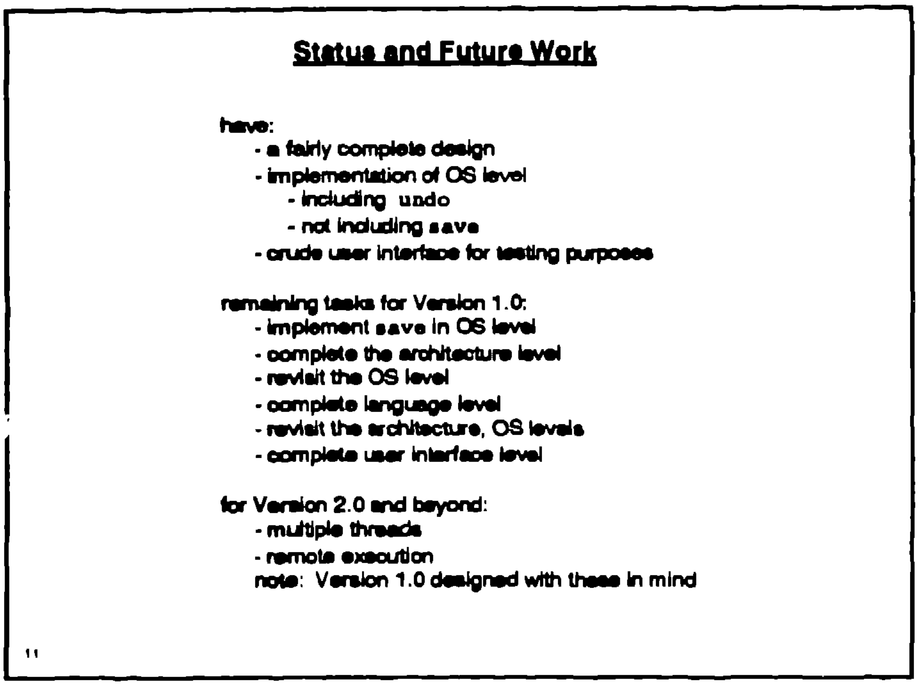
parameters for cache constructor
`objectSize, lineSize, readCacheLine, writeCacheLine`

problem: would like to return *reference* to cached copy

```
// insert breakpoint at function with name f
  instrLoc = functionTable[f].address;
  childAddressSpace[instrLoc] = trapInstr;
```

if reference returned, can't guarantee lifetime
e.g. `function(`*`cache-ref-1`*`, ... , `*`cache-ref-k`*`);`

- bracket simultaneous references with `disable-enable` calls
- return value rather than reference; provide `update` function



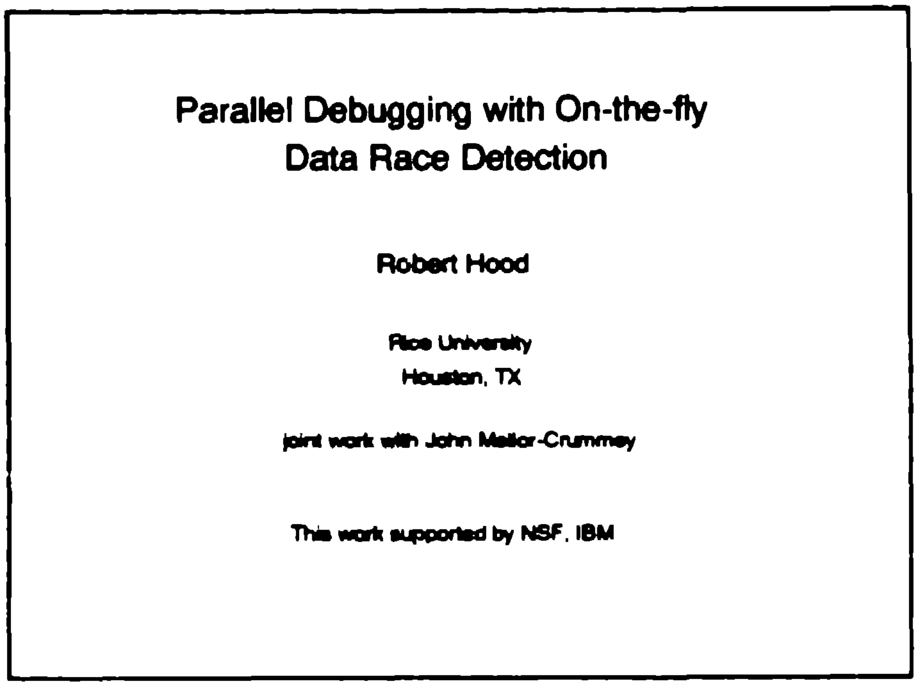
Status and Future Work

have:
- a fairly complete design
- implementation of OS level
  - including `undo`
  - not including `save`
- crude user interface for testing purposes

remaining tasks for Version 1.0:
- implement `save` in OS level
- complete the architecture level
- revisit the OS level
- complete language level
- revisit the architecture, OS levels
- complete user interface level

for Version 2.0 and beyond:
- multiple threads
- remote execution

note: Version 1.0 designed with these in mind



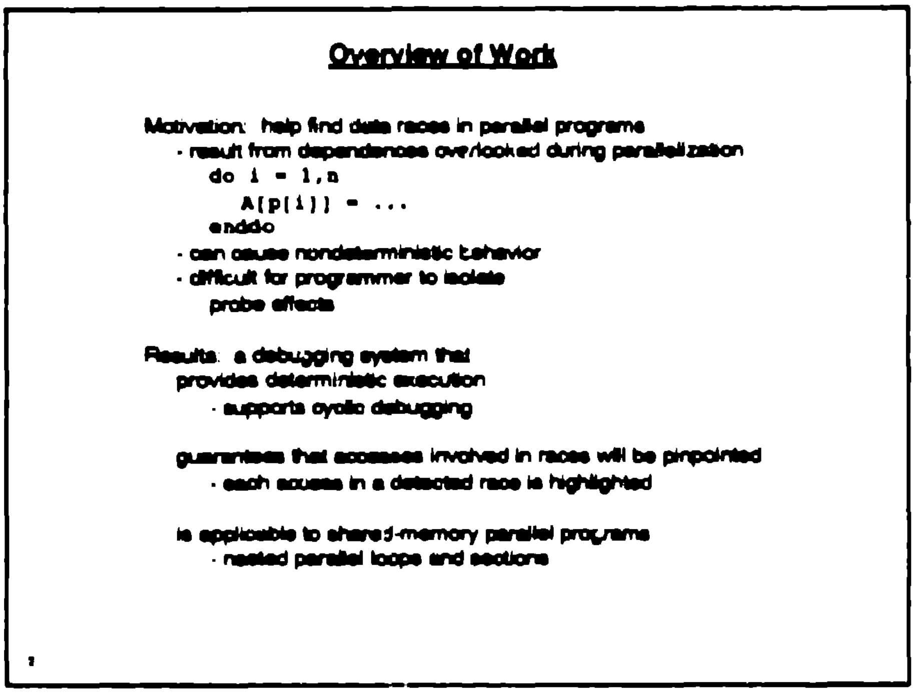
Parallel Debugging with On-the-fly Data Race Detection

Robert Hood

Rice University
Houston, TX

joint work with John Mellor-Crummey

This work supported by NSF, IBM



Overview of Work

Motivation: help find data races in parallel programs
- result from dependences overlooked during parallelization

```
do i = 1,n
   A[p[i]] = ...
enddo
```

- can cause nondeterministic behavior
- difficult for programmer to isolate
  probe effects

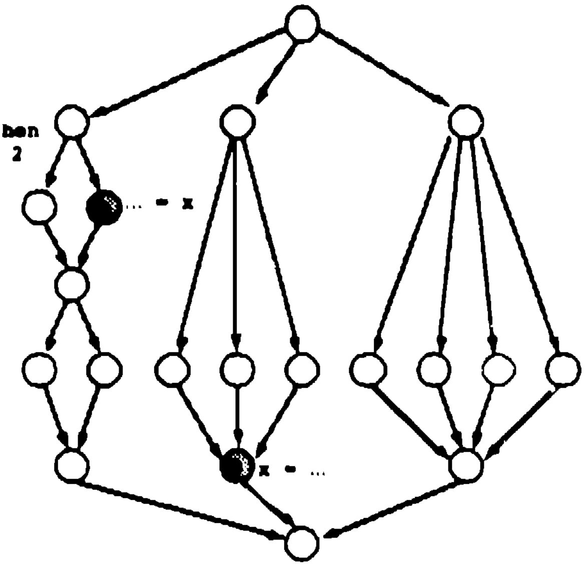
Results: a debugging system that
provides deterministic execution
- supports cyclic debugging

guarantees that accesses involved in races will be pinpointed
- each access in a detected race is highlighted

is applicable to shared-memory parallel programs
- nested parallel loops and sections

Modelling Concurrency: Data Race Example

```
...

pdo i = 2, 4
   ...
   if (i.eq.2) then
      pdo j = 1, 2

         ...

      endpdo
   endif
   pdo j = 1, i

      ...

   endpdo
   ...

endpdo
...
```



Parallel Debugging Approaches

static analysis
- conservative approximation

  PFC, PTOOL, ...

post-mortem analysis
- trace logs

  Netzer & Miller, PPoPP '91

on-the-fly analysis
- report only actual races
- no trace logs

summary methods

Steele, POPL '90
- imprecise report

precise methods

Dinning & Schonberg, PPoPP '90
- expensive, 150-1000% overhead
- suitable for interactive debugging

System Overview

implement a precise on-the-fly mechanism consisting of:

program instrumenter
- insert code to check accesses that may be involved in races
  use static analysis to identify potential races

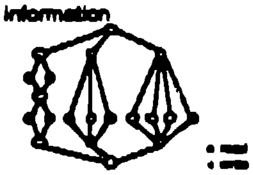
run-time library
- for each shared location: maintain access history
  conceptually, a list of all threads that have accessed it

- for each access: check to see if there is a thread racing with it
  scan history to see if concurrent thread also accessed
  (if at least 1 is a write then race exists)

user interface
- report accesses in each detected race

Our Precise On-the-fly Protocol

for each shared variable, maintain 3 pieces of information
- thread ID of last writer
- thread ID of lowest, leftmost reader
- thread ID of lowest, rightmost reader



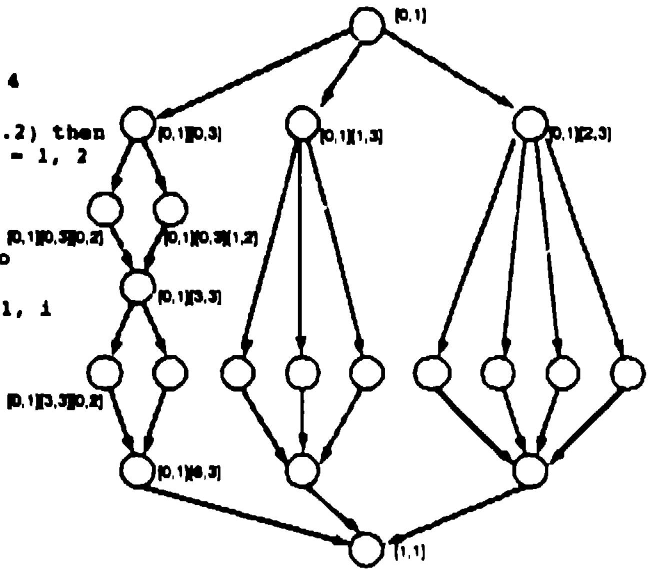
at a *read*
- if concurrent with last writer, report race
- update access history if necessary to maintain invariant

at a *write*
- if concurrent with any thread in access history, report race
- update last writer in access history

thread ID (*tag*)
  compact
  support fast concurrency test
    is thread *t1* concurrent with thread *t2* ?

Modelling Concurrency: Assigning Tags to Threads

```
...

pdo i = 2, 4
   ...
   if (i.eq.2) then
      pdo j = 1, 2

         ...

      endpdo
   endif
   pdo j = 1, i

      ...

   endpdo
   ...

endpdo
...
```



Properties of Protocol

Key Theorems: Mellor-Crummey

1. If a program execution contains one or more data races, at least one will be reported for each variable involved.

2. Any execution interleaving suffices for detecting races.

Efficiency

asymptotic improvement in worst-case time and space requirements

- time proportional to nesting level of parallel forks
  independent of maximum logical concurrency

- achieved by restricting source language
  - must have *regular* synchronization
  - cannot have *unordered critical sections*

Instrumentation System

- allocate storage for access history variables
- insert instrumentation source-to-source

one approach:
  create "shadow" variables to hold tags

```
double precision A(100, 100)
integer A_tag(8, 100, 100)
```

  for accesses at dependence endpoints:

```
B(p(i)) = ...
call WriteCheck( B_tag(p(i)) )
```

If `B` is a formal parameter
  `B_tag` must be as well

If one of the dependence endpoints is `call P(...)`
  must propagate instrumentation into body of `P`

User Interface Options

Batch Execution:

```
Data Race detected:
  Parallel construct carrying dependence: routine='main', line=284
        parallel loop 5000 i = 1, n
  Earlier Reference: iteration vector=<3> routine='value', line=511
        x1 = a(i, 2 * j - 1)
             ...............

  Current Reference: iteration vector=<4> routine='main', line=263
        a(index(i), j) = 2.d0 * a(index(0), j) - a(index(i), j)
        ...............
```

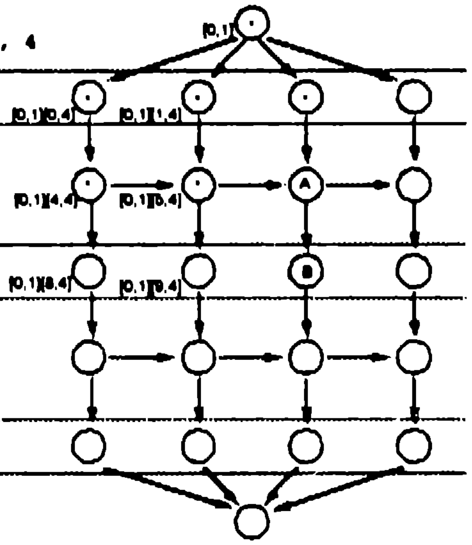
Post-mortem
  after erroneous production run
    recompile and re-execute

Interactive (with *gnu*, *dbx*, *ParaScope*)
  treat detected data race as breakpoint
  - indicate both ends of dependence + loop carrying it
    analyze state at second access

  in ParaScope
  - races displayed the same as dependences
  - user sees uninstrumented code

Handling Synchronization

```
stmt list 0
parallel pdo across I = 1, 4
    stmt list 1
    wait (a)
    stmt list 2
    send (a)
    stmt list 3
    wait (b)
    stmt list 4
    send (b)
    stmt list 5
end pdo
stmt list 6
```



Status & Future Work

have:

automatic system for inserting instrumentation code

instrumentation library for nested PDO's
sequential version

design for instrumentation mechanisms for remainder of PCF Fortran

integrated data race detection into ParaScope debugger
sequential execution only

in progress:

use of interprocedural information to improve instrumentation

implementing instrumentation mechanisms for rest of PCF Fortran
moving target!

Debugging with Lightweight Instrumentation

Benjamin Chase
Robert Hood

Rice University

Abstract

As part of a debugger project at Rice University, we wanted to provide support within our debugger for such operations as noticing modifications to a memory location. Efficiency concerns led us to choose the technique of automatically applying small machine code patches to the program.

We have experimented with this type of program instrumentation and have demonstrated the feasibility of using it for the task of watching modifications to memory. Our use of patches to monitor all stores to memory is unusual, because of the large number of patches involved. In detail, we insert a machine code patch for every machine instruction that may modify the memory locations being watched. This action does incur significant costs in program text space and running time, but provides invaluable functionality within the debugger. Our method is especially appealing if appropriate alternative support for data breakpoints is not provided by the operating system or hardware.

Introduction

We are implementing a debugger at Rice University. In addition to the more traditional operations such as location breakpoints of various kinds, we want this debugger to have some support for data breakpoints. That is, we want to be able to stop when some particular variable acquires a value. At a basic level, to implement data breakpoints without compiler support, the debugger must somehow notice when a machine instruction modifies any of the memory locations being watched.

Our project runs under the Unix[*] Operating System, and so the obvious first choice when implementing the debugger operations is to use the breakpoint facility provided by it. These routines, a collection of operations grouped under the ptrace() system call, work well enough when setting a simple breakpoint at some location in the program. However, they are far too slow for intensive, non-interactive use. Using this process tracing facility, the most obvious method for monitoring changes to process memory is to execute the program in short steps, making sure to stop the process just before each store,

[*] Unix is a trademark of AT&T Bell Laboratories.

and inspect the location being modified. Debuggers using this implementation can slow the monitored process down by a factor of up to 20,000. For a program run of any substantial duration, when faced with such a slowdown, it is usually quicker and almost as effective to simply add printing and assertions to the source code, and recompile.

To avoid the ptrace system call, we instead construct an automated way to insert code before every store instruction, and before every other kind of instruction that could modify memory. The inserted code will check the address to which the store location will write, against a table of memory locations being monitored. If the address is in the table, some special action will occur, such as pausing the program and signaling the debugger. Otherwise, the program will proceed as it normally would have, without the inserted code.

For our project, instead of actually inserting code before each store instruction, we put the new code elsewhere and connect this new code to the patch point with branches. Typically, one instruction at the patch point, usually the instruction that modifies memory, must be overwritten by a branch to the new code. This instruction is relocated to a slot in the new code. At the end of the new code, a branch back to the patch point completes the installation of the patch. This is not a new technique, but its utility is often overlooked. Our design is that of Kessler[Kessler], though some engineering is needed to make our application of this technique appealing.

Architectural Requirements

This method relies on the ability to confidently install patches to machine code, ideally even without significant support from the compiler. There are several restrictions resulting from the requirement that machine code patches are being used. In short, our method works best on RISC architectures, on systems that have an obvious separation between machine code and data.

Patching is difficult if machine instructions can be of different lengths. A uniform size of machine instruction is the easiest to handle, when inserting patches. The uniform instruction size simplifies the decoding of instructions on such architectures. Also, if the branch instruction that is used to reach the patch is large, and the instruction that modifies memory is small, it may be that many instructions surrounding the small instruction must be relocated. This increases the complexity of the case analysis in the automated patch installer.

When installing a patch, it is easier if the architecture has a branch intruction that has a generous branch distance. Ideally, this branch distance would be as large as any program that is anticipated to run on this architecture. This would allow us to place the new code for the patch where it was most convenient. It is difficult to install patches if the branch distance is limited, because space for the patch must be found close to the patchpoint, and patchpoints will be sprinkled throughout the program. In this case, spaces for patch code would have to be created (or reserved beforehand) throughout the machine code, rather than in one large arena. Also, the particular branch instruction used for linking in patches should neither require the modification of any registers to certain values, nor as a result of its execution modify any registers (other than the program counter, of course).

Typically, it will be necessary to save some of the machine registers at the beginning of each code patch and restore them at the end, without needing any free registers a priori. This problem, which we informally call "getting one's foot in the door", can be a puzzle on some architectures, but often there is a solution, albeit contorted. On architectures that lack even a contorted solution, it may be necessary to obtain some help from the compiler, such as reserving a number of registers solely for use by the debugger, or perhaps requiring that the compiler allocate for the debugger's use a conveniently located place for saving registers.

Narrow operand fields in machine instructions are problematic, if they must be relocated. An example of an instruction that might need to have its operand field(s) adjusted is a relative branch instruction. If it is moved to a distant location, the obvious adjustments to account for this move might overflow the operand fields. Even if a sequence of (possibly larger) instructions exists that is equivalent to the relocated instruction, generating this equivalent sequence greatly complicates the automated installation of patches.

Branches with delay slots can also complicate the automated patcher. On some architectures, there are restrictions against placing certain kinds of instructions in delay slots, such as branches. If a store instruction had been placed in the delay slot by the compiler, the automated patcher would normally want to overwrite that store intruction with a branch. Instead, the automated patcher must use some more complicated strategy. Similar complications occur if the patcher needs to overwrite the delayed branch instruction itself. Solutions to these problems, perhaps involving multiple patches, such as those described by Kessler [Kessler], can probably be developed for most architectures.

Register windows, such as the implemention on the SPARC [SPARC], can cause strange problems. Because register windows may be saved to the register save areas, normally located in the runtime stack, the SAVE instruction, which creates register windows, represents an instruction that can modify memory. If an address being watched is within a register save area, should the SAVE instruction be counted as modification of memory? Because register windows may be cached within a SPARC microprocessor, it may be that no write to that register save area occurs. If it does occur, it may be the result of some subsequent SAVE instruction.

The simple solution is to assume that this occurrence counts as a modification to memory. Justifications for this decision are that the modification may occur in some execution of the program, and that it is unusual for a debugger to be placing a data breakpoint on a register save area. If the user's variable resides in an area that may be overwritten by a register window save, he probably wants to know about it.

Operating System Requirements

The system under which our method will run should provide an efficient way to install the patches. If we insist that the user must declare that he will want data breakpoints before running the program, we can install the patches efficiently, simply reading and writing the program. However, this situation is very inconvenient for the user. To install data breakpoints after the program has been started, the debugger must be able to write the text segment of the running process, so that the debugger can install the patches after the program has been started.

Under the Unix operating system, writing the text segment of a running process is performed via the ptrace() system call, and each call is slow. A straightforward implementation of the automatic patch installer will need to perform many isolated single-word modifications, because store instructions occur scattered throughout the program. A simple implementation of the installer will also perform a similar number of writes of patch code, with each patch being some small number of consecutive machine words. If there is a high cost for each modification of the process text, but bulk modifications are available and comparatively cheaper than modifying one word at a time, it may be desirable for the patch installer to use this method. The installer would collect all the changes, and perform the entire installation as one bulk modification, rewriting the entire process text.

We will need space in the running process for all the patches. Again, a simple solution probably exists on most systems, if we require the user to declare before he starts the proram that data breakpoints will be required. It may be reasonable to simply assume that data breakpoints will be required, and always acquire the extra text space before debugging starts, if the costs of reserving this extra space is low. In a virtual memory environment, the main cost of this might only be extra swap space. After the process has been started, it may be difficult to obtain the needed space. This may be a serious problem if the text of the running program is restricted to a single contiguous piece of memory, and is abutted by other parts of the process address space, leaving no room for it to be enlarged.

On systems supporting mapping of files into the address space of the process, it may be possible to map a file as text rather than data. If this is available, it solves the problem of acquiring space in the running process, and may eliminate the inefficiencies involved in writing the patches. (However, it can only eliminate half of those writes, because it does not solve the problem of patching the new code into the existing code.) To actually do the file mapping in the monitored process, it may be convenient to require that a small collection of routines, callable from the debugger, be already linked into the monitored process.

Software Requirements

Late (dynamic) linking creates a problem for our method, because code that must be inspected for store instructions is not necessarily present at inspection time. To take advantage of the simple solutions above, in which the user must declare beforehand that data breakpoints will be required, we would also have to prohibit late linking. The more expensive solutions, involving installation of patches in the midst of a debugging session, will also require that every piece of code that needs to be watched is present, or that the debugger be notified later, when the additional libraries are linked. There may be no existing provision for such notification by the linker.

Ideally, we want this method to work in as many situations as possible. Our method has no great inherent need that the code be generated by a particular compiler, or just one compiler. However, one requirement that should be satisfied is that the compiler leaves sufficient information to distinguish machine code from data. This confusion between code and data commonly occurs when the machine code written by the compiler contains various kinds of immediate constants, such as jump tables, floating point constants, and sometimes immediate integer values.

If the automated patcher cannot distinguish these various kinds of data from machine instructions, it might mistake some of them for instructions that can modify memory, and instrument them. Typically, the constant would be overwritten, and relocated elsewhere to the body of the code patch. This would be a serious mistake. Thus, we want some way to reliably distinguish code from data. Note that the instructions and data need not be separate, merely distinguishable.

Programs that do not distinguish code and data, and create new code as they run, are difficult to handle using our method. As each new piece of code is generated by the program, it would have to be inspected and patched by the automated patcher. Also, when the storage for that dynamically generated and patched code is recycled for reuse, the space taken up by the patch code would need to be recycled also. We do not expect that our method of implementing data breakpoints would be easy to include in such an environment.

Results

We have implemented a simple version of the method we describe for the SPARC architecture. We have implemented watchpoints for ranges of addresses, rather than just for single addresses, as the test is almost as quick, and much more powerful. We avoid the problems of acquiring patch space by linking in an extra object module containing an arena of unused text space. The arena is sufficiently large to hold patches for all the instructions in the original program that may modify memory.

We currently install all of the patches before debugging begins, simply scanning the program for stores and rewriting the program with the patches installed. This act adds the capability for data breakpoints, but does not set any. Essentially, the modified program checks store references, but since no data breakpoints are set, none of the stores trigger any special action. Along with the arena of patch space, routines are linked into the monitored program, which will add or remove an address range from the list of address ranges to be monitored. These routines can be called from within the Unix debugger dbx [Sun Debugging] to add or remove an address range when the program is being debugged.

Our current implementation does not address modifications to memory caused by traps to the operating system. Because the machine code within the operating system cannot be patched by our routines, the best we can easily do is insert a patch after the system call, and check to see if the data at the watched addresses has changed. This is not as good as the support that we provide for store instructions, which detects the write to memory even if the value written is the same as the value previously there.

This style of patch, that detects patches after system calls by comparing data, will be far more expensive to execute than the kind we have proposed. Our reported costs do not include this cost, whatever it may be. However, we do not expect traps to be a large fraction of the instructions in a program. In addition to this lower frequency, when calculating the effect such patches will have on the program's speed, the cost of the patch is balanced against the cost of a system call, rather than the cost of executing a single store instruction.

Because we can only check after system calls, it may make more sense to change the semantics of all our patches, so that the address check always occurs after the memory is written, rather than before the write for simple modifications, and after it for modifications by the operating system. This would present a more uniform appearance of data breakpoints to the higher levels of the debugger.

Costs

Our method involves a drastic modification to the text of a program, and so there are large space and time costs involved. In programs we have examined, store instructions make up roughly 10% of the instructions, both statically and dynamically. In the worst case, every instruction in the entire program which might modify memory has to be patched. Unfortunately, this worst case is likely to be a very common case. If the user knows that the store occurs in a small subpart of the entire program, the number of patches needed can be reduced accordingly. However, the user may not be certain that only the stores in one subpart of the program need to be watched. Various kinds of dataflow information could also prove useful in reducing the number of patches, but that information may not always be available. We currently do not perform any analysis that might reduce the number of patch points.

A static frequency of store instructions of 10% means that our implementation of data breakpoints requires space for a patch for roughly every tenth instruction in the program. The patches we have designed vary in size depending upon the context of the patch location, but typically need 9 machine instructions of additional space per patch. (Shorter patches than this are possible, if sufficient registers to execute the patch are known to be available for use by the patch.) Thus, the size of a machine program when patched is roughly twice as large as the original. In comparison, many other implementations of data breakpoints have neglible space costs, and those costs do not vary with the size of the program.

When the program is running, the time of executing the instructions for a patch will be incurred roughly every tenth instruction of the original program. For our implementation, a patch takes 25 instructions to execute. An additional 6 instructions are executed for each address range against which the address of the stores is checked. Finally, an additional 15 instructions must be executed if a match occurs, to save some more registers, and fix up the process state. We do not include this last amount in calculating the slowdown of the instrumented program, on the assumption that if an address matches, the debugger will soon be performing some much more expensive action, such as interacting with the user, rather than proceeding to the next store instruction. Thus, the dynamic costs of our implementation cause the program to run 4 to 5 times slower than the original program, when only several address ranges are being monitored.

When appropriate support is provided by the hardware or operating system, data breakpoint implementations exist that affect execution speed much less than our lightweight instrumentation does. Hardware solutions [Pappas and Murray] can provide a limited number of data breakpoints at basically no cost in running speed, although the number of address which can be watched simultaneously is limited by hardware. Operating system support, such as the ability to alter page protection in virtual memory environments, can yield fast implementations of data breakpoints also, by removing write permission from pages containing the watched addresses, so that stores to these pages will generate traps.

Conclusion

Although our implementation of data breakpoints is slow, programs instrumented in this way are several orders of magnitude faster than the obvious solutions using the ptrace system call. For debuggers that rely on this system call for watching program variables, our method should be considered as a replacement, if our architectural, operating system, and software requirements can be met. On new architectures, the operating system support for data breakpoints, (that is, the ability of the debugger to modify page protection of the monitored process) may be planned but not yet implemented. In this case, our method may serve well in the interim. Finally, on hardware that provides support for data breakpoints, our method might be used as a backstop, if the hardware resources are exhausted. However, a steep degradation of speed, objectionable to the user, will occur when our method is used to relieve an overburdened hardware solution.

References

[Kessler] P. B. Kessler, "Fast Breakpoints: Design and Implementation", in Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices Vol. 25, No. 6, June 1990.

[Pappas and Murray] C. H. Pappas and W. H. Murray III, "80386 Microprocessor Handbook", Osborne McGraw-Hill, Berkeley, CA, 1988.

[SPARC] Sun Microsystems, Inc., "The SPARC Architecture Manual, Version 8", Part No. 800-1399-12, January 1991

[Sun Debugging] Sun Microsystems, Inc., "Debugging Tools", Part No. 800-1775-10, May 1988.

Integration of Performance Analysis and Debugging

Marty Itzkowitz
Krishna Kolluri
Madhavan Thirumalai
Alan Foster
Paul Sanville

November 15, 1991

Silicon Graphics Computer Systems
2011 N. Shoreline Blvd.
Mountain View, CA 94039

ABSTRACT

Performance measurement and debugging are complementary techniques used to produce a correct, efficient program. In this paper we will discuss the integration of the two in SGI's *CodeVision*™ product. We first discuss the notion of data sampling (snapshots), and sample traps. Then we will talk about the kinds of cumulative performance data that is collected, and the instrumentation used to collect it. We will also discuss tracing data, used for both performance measurement and debugging, and then we will talk about extensions to our tools to support multiprocessor applications. Finally, we present our conclusions.

1. Introduction

Performance measurement and debugging are complementary techniques whose objective is to produce an efficient program that gives the right answer. If a program fails, then it certainly takes too long to run, and hence has a performance problem. On the other hand, even if a program gives the right answer, if its algorithms or their implementation cause it to take longer than the end-user will wait, it clearly has a bug. In this paper, we will discuss the overlap of performance measurement and debugging, and describe the implementation of SGI's *CodeVision*™ *Performance Analysis* tools that exploit this commonality.

To the developer, performance measurement and debugging are quite closely related: they both involve studying the behavior of a program, and it seems quite natural that similar techniques would be used, and, more importantly, that the user model presented to do debugging tasks and that presented to do performance measurement be similar. Furthermore, the developer may start to do a "performance" experiment, but detect some problem that really is a bug. In our system, even while performance data collection is taking place, the user may exercise the full power of a debugger: stopping the process, setting breakpoints, examining memory, and so forth. These debugging features are available even if the process has been instrumented.

Another reason to integrate performance analysis and debugging is that the extraction of performance data from a process involves precisely the same sort of control operations as those needed for a debugger: start and stop the process; read data from its address-space; determine the process' call stack; note when it makes a system call, *etc*. All of these functions are provided by a process-control server that is common to the debugger and performance analysis tools, and is described elsewhere.[1]

™ *CodeVision* is a trademark of Silicon Graphics, Incorporated

[1] P. Sanville, Chang, A.M. and Foster, A. "Managing Debugger Process Execution: A Finite State Machine Approach", next paper in this session.

The next section of this paper describes the sampling paradigm that allows for performance measurements over various phases of execution of the program; the third section describes the performance data that can be collected, and the instrumentation to collect it. The fourth section discuses various kinds of trace information, which are used for both debugging and performance analysis. A fifth section discusses extensions to support multiprocessing, and finally we present our conclusions.

2. The Sampling Paradigm and Sample Traps

Many performance analysis tools give measurements that are global over the entire run of a program: implicitly, they assume that a program's performance characteristics do not change over the course of a run. However, many real programs simply do not behave that way. They execute in phases, and each phase has different performance characteristics. One objective of the CodeVision project was to be able to extract performance characteristics for each independent phase of execution of a program. To meet this objective, we defined a *Sample Trap*, which is analogous to a *Stop Trap* (program breakpoint). Data is recorded at every sample trap, and the visualization tools allow examination of the cumulative data between any two such points.

To the user, the samples that were taken are indicated by tick marks along a timeline spanning the experiment. A pair of calipers are provided that can be used to mark of the region of interest, and all of the tools can be synchronized to any particular caliper settings.

In a debugger, when you wants to see how a program is progressing, you can set (stop) traps that will stop the program at various interesting places in the code, and examine its state when it reaches those places. Such traps may be defined to fire when the program counter (PC) reaches a particular address or source line, when the program enters or exits a particular function, or when it starts or completes a system call. Other traps can be set as watchpoints, which will trigger whenever a watched memory region is accessed or changed. We also provide a pollpoint trap which is fired at regular timed intervals. And, of course, there is a manual trap, namely the "stop" button.

In our system, traps are defined as having flavors, and a stop trap is the flavor commonly used in debugging. For performance measurement, we also defined a sample trap, which can be planted in exactly the same way as a stop trap. The difference between the two is the behavior of the system when the trap fires. A stop trap stops the process, whereas a sample trap causes the performance measurement tools to extract all of the performance data up to that point from the process and the operating system, and record it in an experiment record.

Since the bulk of the performance data we can provide is maintained inside the process' address space, on its stack, whenever we stop the process, we can use the process-control server to read the data and record it in our experiment record. The performance visualization tools can look at that data, and show the net difference between any two sample points in an experiment. For example, one could set traps at entry and exit to some operation thought to be expensive, and see exactly what took place between those two points: which routines were called and how many times, how many times each line of code was executed, where the program PC was found, and so forth, but only for the interval between the two points.

Sometimes it is difficult to know exactly where in the program traps should be planted to demarcate the programs phases. For window system programs, however, it may be straightforward to detect the changes as the program runs. To that end, we also provide a manual sample trap: whenever the user clicks a button on the user interface, a sample is taken. When an interactive window system program is quiescent, the user can take a manual sample, and then request a particular function. When the results of that function appear on screen, the user can take a second manual sample, and examine the data between those two events.

3. Cumulative Performance Data and Instrumentation

The CodeVision performance tools can record a wide variety of measurements of a process. We support number of different kinds of cumulative data: measurements of resource usage by a process, as maintained by the kernel; statistical PC profiling; function execution counting, and/or basic-block counting. Each of these measurements are selectable, in any combination, from a performance panel, integrated with the debugger. The performance panel also provides for the entry of a directory name used for recoding all data in an experiment record. It may be invoked any time during a debugging session, although any data specifications must be set before starting the execution.

The CodeVision performance tools do not require that the user know about relinking with special libraries or program instrumentation (except for *malloc/free* tracing). When performance data is selected, and a run started, the system will automatically instrument the program according to the data requested by the user.

PC profiling is installed by intercepting the startup code of the process moving the stack down to provide a buffer for accumulating the counts, and invoking the kernel service before branching to the normal user startup code. It introduces a small amount of overhead at startup, and will use a fairly large region of the stack (one halfword for each instruction in the program). During normal running, however, there is no distortion of the behavior of the program.

Counting, either of basic-blocks or of function calls (a subset of basic blocks) introduces substantially more overhead: it translates the program into a substantially larger program that maintains a set of counters, also on the stack. While the actual behavior of a program instrumented for basic-block counts is significantly different from an uninstrumented program, the counts accumulated allow for exact computation of instruction counts as if the program were uninstrumented. Unlike PC sampling, these numbers are exact, not statistcal. If both PC sampling and counting are requested, they will both be performed, and both kinds of data will be extracted. The program that does the instrumentation is based on the system program, *pixie*.

4. Tracing Data and Trace Traps

For some kinds of problems, either performance or debugging, it is useful to see the pattern of behavior of the program. To that end we defined another flavor of traps, a trace trap. When a trace trap fires, rather than either stopping the process, or sampling the bulk of its performance data, we record information about the specific event triggering the trap.

We support three kinds of traces: *malloc/free* tracing, system call tracing, and page fault tracing. They may be selected individually or in combination from the performance panel, just as for cumulative data.

4.1. *malloc/free* Tracing

The tracing of *malloc* and *free* events can be used to deal with both performance anomalies and bugs. The most obvious performance problem in the use of dynamically allocated storage is a leak: a region of memory is allocated, but never freed. If the user selected *malloc/free* tracing, each call to either of these routines (or to *sbrk* or *realloc*) will record a record of where the region was allocated or freed, and its size; we also record the callstack of the program at the time of the call.

The trace can be visualized as a map of the heap, showing those areas allocated and freed, and those areas that have been allocated, but not freed. To further aid is figuring out the cause of the problem, the trace also includes indications of where sample traps fired, so that, for example, one can see exactly which regions of memory were allocated and/or freed during a particular phase of execution of the program. The user interface to this data also provides a search capability to look for events associated with a particular heap address, and a facility for listing all unmatched *free*'s, including duplicate *free*'s.

The use of *malloc/free* tracing requires linking with a special library, *-lmalloc_cv* and provides a very fast tracing mechanism, buffered within the target process.[2] The callstack traceback for the event is done by the process itself, rather than through the process control server; the trace trap fires only when the buffer within the process is full. The buffer is emptied at those times, and also whenever a sample trap fires, so that the sample events can be correlated with *malloc/free* events.

4.2. System Call Tracing

Both bugs and performance problems can be manifested in a program's sequence of system calls. Sometimes, for example, an *lseek()* on a file can be made in one place of a program, confusing a subsequent *read()* in another. Sometimes, the program makes a large number of useless system calls: it may repreatedly ask the system for its PID, when it could cache it. We have actually found programs that size a file by reading it, instead of calling *stat()*.

To support investigation of these problems, we support tracing of system calls. Whenever a call is made, the arguments to the call, and the call stack of the program at the time of the call is recorded.

4.3. Page Fault Tracing

Another source of performance problems in a program is attributable to excessive paging. To support investigation of this problem, we also defined a page fault trap. Whenever the process takes a page fault, we record the faulting address, and the call stack of the program at the time of the fault. By looking at these events, the user can see what portions of the code are responsible for paging events, and, hopefully, recode the algorithms to reduce their working set.

5. Multiprocessing Extensions

The most important extension to support multiprocessing is the extension of the notion of sampling to cover all threads of a process. Sample traps can be independently specified to trigger in one or more threads, but, whenever they trigger, all threads are stopped, and the independent data for each thread is recorded. The data for each thread may be separately visualized and compared.

Implementation of multiprocess support required the detection of those events which cause a process to be created or to exit. Code was added to nurse each process through its gestation, and then resynchronize all the processes in the MP application for future sampling.

Tracing is also be done independently on all threads in an MP application, and the trace data can be independently viewed.

6. Conclusions

Debugging and performance analysis share many common features, and many of the process-control operations needed for one are needed for the other. A combined user model allows great flexibility in performing both kinds of operations, with relatively little change in mind-set. A sampling paradigm can be used to isolate the behavior of a program during its various phases of operation, and allow the user to understand the evolution of a program's behavior. Tracing can also be used to understand performance anomalies and bugs, and both of these techniques further aid the understanding of multi-processing applications.

[2] We explored the possibility of an implementation that does not require any kind of special linking, but it had far too much overhead, as each event is separately extracted from the process.

Managing Debugger Process Execution: A Finite State Machine Approach

Paul A. Sanville
Ann Mei Chang
Alan Foster

Silicon Graphics Computer Systems
2011 N. Shoreline Blvd.
Mountain View, CA 94039

ABSTRACT

1. Introduction

One of the more complex areas in debugger implementations has been the correct handling of process execution. Traditionally, each execution control command is implemented by an independent routine or set of routines that synchronously manage the flow of control for each phase of the command's execution. Simple commands like "continue" and "stop" are initially easy to implement, but this approach becomes difficult to maintain and extend, especially when interactions between different commands and handling of multiple processes are considered.

In the *CodeVision™ Debugger*,[1] we specified the process execution handling as a finite state machine. We found that the formal model has resulted in significant improvements in robustness, extensibility, and maintainability. Furthermore, the design proved to be easily enhanced to handle multiprocess debugging features.

2. The CodeVision Debugger

The *CodeVision Debugger* is a new source-language debugger developed at Silicon Graphics as part of an integrated programming environment. Using a distributed client-server model, the low-level process control functions are contained in the *Process Control Server*, which communicates with client "views". Multiple 'views" exist as clients of the *Process Control Server*, providing a user interface for accessing underlying data and functionality. The available execution control operations include the traditional run, stop, step, and return commands as well as enhanced handling of interactive function calls, including nested interactive sequences. The trap mechanism supports traps on breakpoints, function entry or exit, signals, system calls on entry or exit, page faults, and data watchpoints.

As part of the *CodeVision* integration strategy, the *Process Control Server* also provides integrated data collection features for the *CodeVision Performance Analyzer*. Thus, the execution model is complicated by the handling of pollpoints and sample traps where the process is temporarily paused, performance data is collected, and the process is automatically resumed.

3. The Finite State Machine

The execution handling of the *CodeVision Debugger* was initially implemented using a fairly traditional model involving large case statements and separate execution control routines. Initially, we had problems achieving a high degree of robustness without a formal model. Then, as we began to integrate more complex features such as returning from a stack frame, source line single stepping, and handling of pollpoints, our initial

[1] CodeVision is a trademark of Silicon Graphics, Inc

model began to break down. Some interactions, such as incoming pollpoints for data collection during single stepping, were extremely difficult to handle reliably.

A finite state machine was designed to formally specify all of the possible states and transitions for handling the existing execution requests and traps. The initial specification involved multiple deterministic finite state machine with nineteen distinct states. Through progressive refinement, the original design was coalesced into a single nondeterministic finite state machine with three states, with a fourth being added later for handling multiprocess synchronization.

For handling transitions for a single process, three distinct states exist: *Running*, *Stopped*, and *Terminated*. The allowable set of events includes both process control requests and the triggering of various traps. In each state, transitions are defined for each of the possible events, some of which are illegal. A table defines each of the legal events, the resulting state, along with the set of actions to be performed. Additional state information is stored in the debugger's internal process object and obtained through the /proc kernel interface.

As an example, consider the case where a stopped process is given the "return from current frame" request and encounters a breakpoint before the request is completed. The process would initially be in the *Stopped* state. The event, *ReturnRequest* would cause a transition into the *Running* state after executing the actions to set the frame exit trap, resume execution, and notify the client views. Upon encountering the breakpoint trap, the process moves into the *Stopped* state, and the actions are performed to clear the frame exit trap, notify the client views, execute any attached actions for the triggered traps, and replant any tenacious traps necessary (if removed previously to be stepped over).

4. Multiprocess Debugging Extensions

The finite state machine was easily extended to handle multiprocess debugging. As the *Process Control Server* maintains control over each of the processes, it is able to handle synchronization across the entire group. In the multiprocess case, the *CodeVision Debugger* provides an extension to the trap mechanism to specify that all processes in a process group should be stopped when a process stops on the trap. A new state, *Suspended* was introduced to ensure that such traps are correctly handled when multiple traps trigger simultaneously in different processes.

A triggered trap may or may not actually stop the process due to conditionals, counts, and data collection sample traps. Furthermore, a number of different processes in the process group might encounter traps simultaneously. Thus, when a process encounters a trap that would not normally cause a stop (due to a false conditional or otherwise), it is moved into the *Suspended* state. Then, the remaining process events for the process group are handled to determine if all the processes should be kept stopped or if the suspended processes may be continued.

5. Conclusions

Although the initial finite state machine model design and implementation was fairly time consuming, the resulting architecture has provided major gains in the robustness of the execution model, ease in integrating new functionality, and the possibility for verification. Simply the exercise of formally specifying the complex interactions involved in execution handling provided insights into previously baffling problems. By further incorporating the specification into a table driven machine, enhancements and bug fixes became simple, while verification of all possible combinations became more feasible. For handling the complex interactions involved in multiprocess debugging, the use of a finite state machine was critical to our success.

A Visual Debugger Constructed by Program Generating Technique

Ming Zhao

CS Division, Asian Institute of Technology
G.P.O. Box 2754, Bangkok 10501, Thailand

Abstract

This paper presents the construction of a visual debugger in which the debugged user program is used as the carrier of visual functions. The realization is conducted by program generating technique. The existing tools are exploited without investigating into their interior structures.

Keywords: Visual debugger, Program generation
Tool integration, User interface

1. Introduction

Visual help has been used in program development long before the term "visualization" was introduced. We use flowchart to demonstrate program structure and use diagram to show data structure. With these visual help, the design is coded to program text much easier. Debugger is inevitably the next step to achieve satisfactory executable code. Yet at this stage, no longer is the visual help available. In recent years, with rapid progress in visualization, there have been works on visualizing the debugging procedure. The two aspects to be visualized are data structure and program execution sequence.

This paper proposes a new strategy to realize debugger visualization. The aim to be achieved is a concise, flexible, yet powerful debugger visual supporting package. It does not intend to become an independent debugger, but to attach visual diagnosing ability to an existing debugger. Program generating technique is adopted to achieve the goal without investigation of the underline debugger. Section 2 of this paper reviews some of the existing works. Section 3 presents the implementation strategy of a visual debugger by program generating technique. Section 4 describes the visual presentation issues of debugging diagnosis information. Section 5 concludes the paper and makes some suggestions.

II. Related Works

There are considerable works on visual programming in recent years[2,3,4,6]. The majority of the works are on language design, compilation, and various applications such as database query. Less efforts are spent on debugging procedure. This is a situation similar to that the powerful source level debuggers were developed many years later than the textual programming languages were designed. One of the reasons is that the debugger is not easy to construct. A debugger is a highly dynamic interactive procedure. To visualize a debugger, data structures and program execution sequences are to be shown visually. Yet not like data structure or program visualization, here no visual relationship is available, the debugger can only obtain indirect information from declarations of data types and variables, and function call sequences. In addition, to generate correct visual effect, the spatial relationship of data items and historical sequence of dynamic execution have to be considered. It is more difficult to organize visual presentation.

An earlier work on program visualization is the PV system[5]. It intends to provide lifetime support for software development. The aim is to support maintainers of large (10^6 lines of code) complex software system. The users of PV system are permitted to look inside and watch their programs run or "open the side of the machine", as the authors described. The PV system is designed for programs written in Ada. The system provides the individuals a variety of graphical representations for static program structures and dynamic procedure execution. The system allows simultaneously display of several different representations for the same portion of a system or the same representation for several different portions through the use of multiple screens or multiple view ports.

Animated algorithms and data structures are described in Barnett[1]. The book is written for novices in programming. It provides simple explanations and practical information for those who need to use data structures and algorithms. A number of programs written in BASIC illustrate program execution and display the data structures while execution. Although data structures are represented visually, few is written about data visualization.

A Collection of papers related to visual programming and visualization are included in a book edited by Chang[4]. The book covers theoretical presentation and practical applications of visual programming systems. The book focuses mainly on visual programming but also includes some related works on program visualization.

Seminar describes an iconic description language for data structure visualization[9]. The paper presents the language DSIL (Data

Structure Iconic Language) for specifying the content and appearance of icons depicting data structures. The prototype is presented in conjunction with the data structure editor in the integrated programming environment being developed at the University of New Burnswick. The design of the DSIL is based on the features of data structures in Modula-2 and is developed in X-window environment.

The VIPS (Visual and Interactive Programming Support) system is presented by Sadahiro[8]. VIPS is a visual debugger working on Ada intermediate languages Diana and quadruple, the syntax tree of the Ada program and the sequences of tuple of an operator and its operands, respectively. VIPS preprocessor analyzes the Diana file for information about blocks and variables. This information and the quadruple are downloaded to the workstation. When a test is executed, VIPS interprets the quadruple to depict program execution behavior. VIPS graphically presents several views of Ada program execution in eight windows: data, program code, block structure, acceleration, figure definition, interaction and editor. Since it works on the intermediate level, the data structures to be processed are simpler ones.

The integrated program development environment FIELD is developed at Brown University[7]. FIELD offers a variety of facilities to the programmer to build his/her own system. It is implemented on UNIX environment and makes use of several UNIX tools: editors, compilers, debuggers, profilers, and make facility. To integrate these tools, a central message server is used to coordinate communication among the tools, a concept called selective broadcasting. FIELD supports visualization of user data structures, including dynamic updating of these structures while the program executes. Program execution can be viewed through the source code or through the visualized source code as call graphs. The intent of FIELD is to become an integrated environment. Visual debugger is only a component of the system. The weakness of FIELD is its performance. 12 megabytes of memory is needed to ensure the minimum environment. Any extension requires consistency with its message broadcasting system. FIELD offers various visual debugging facilities, but large portion of the functions duplicate the existing ones realized in other tools.

III. Visual Debugger Generating

There are different ways to implement a software package. To write a new one is often expensive, though the developers have freedom to select functions and to decide format at their will. To modify or enhance an existing one can better exploit the existing facilities, but this needs experience, especially the knowledge of previous system. Otherwise, it might become more expensive. The third way is to develop new functions outside the existing system, and then attach these functions to it. When necessary, adjustment is made to fit environment and function requirement. This strategy can minimize the efforts spent on repetitive works. The difficulty is that it is not always easy to incorporate new functions into the old system.

In a modern programming environment, various powerful facilities are already provided to the users. To build new functions, it is not necessary to start from very beginning. Yet inconsistency always exists. The FIELD system uses a central message server to cover the differences. But for a limited application involving a small number of tools, this general form of central server is not very efficient. In this case, the program generating technique can achieve more efficient result. In this work, the functions of an existing textual debugger are exploited directly, the visual portion is developed somewhere else. The program generating technique by LEX/YACC is used to combine them together and to make up the inconsistency between them. The following can be described as the characteristics of this system:

- implementing a tool without much effort and little time and space consumption,
- exploiting existing tools as much as possible,
- covering several tools without knowing their interior structures,
- offering user access of visual data structures.

The programming language C is selected as the language for it a visual debugger is implemented. C is one of the most popularly used language. With its specific language structures, it is more flexible than other high level languages, and thus is more difficult to debug.

Dbx is a powerful debugging tool implemented on UNIX system V. With dbxtool's[10] enhanced multiple window interface, it seems there is no need to add any new function. The only thing which dbxtool lacks, and which might be badly needed, especially for novice users, is the visualization of the debugging diagnosis information. To realize visual debugging ability for dbxtool but without investigating into dbxtool code and without understanding the interior data organization of dbxtool, one possible way is to let the debugged program itself do the job. If a piece of code visually showing the data object is attached to the original program, when the program is executed under debugging, the attached code will be activated to produce visual representation. This piece of code is called a visual decorator.

Undoubtedly, if a visual decorator has to be manually inserted into the original program, no one will use it. Fortunately, there exist software tools to automate the procedure. LEX and YACC are two well known tools to generate lexical and syntactical analyzers and to help implement a compiler. In this work, a decorator generator is designed using LEX and YACC. It consists of a library of visual decorators, and a generating algorithm which automatically catches a data object from original program and attaches to it a corresponding visual decorator. The resultant program is a longer, functionally equivalent one with visual decorators.

A visual debugger is supposed to find some dynamic, difficult to be detected bugs. There is no need for it to check the correctness of the syntax. Decorator generator assumes the syntax of the original program is correct already.

Figure 1 is the overall architecture of the proposed visual debugger. It is composed of three major components. The editor can be any one to edit user program text. The modifier is the kernel of the visual debugger. It analyzes user edited source program, attaches visual presentation functions to it, and sends the augmented program for compilation. A visual library is maintained by the modifier. The debugger can also be any conventional debugger, with visual display windows created by the execution of debugged user program.

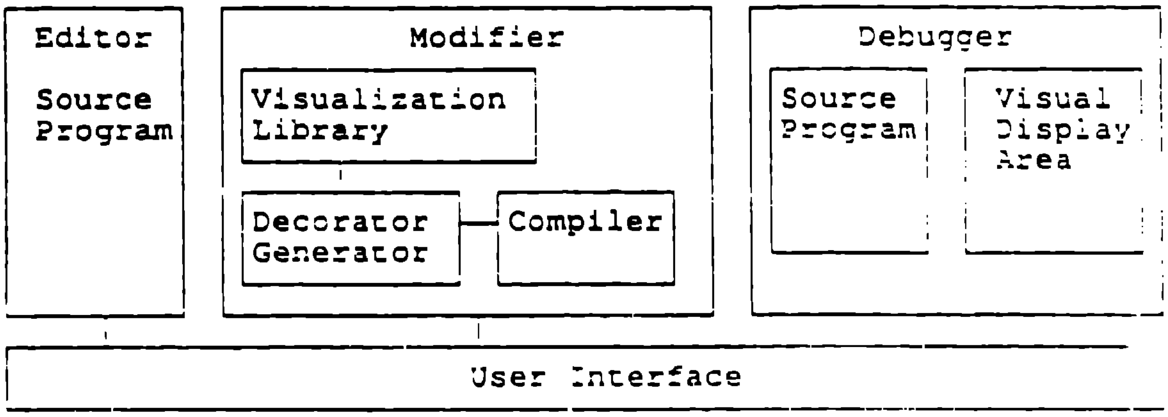


Figure 1. a model of visual debugger

IV. Visual Debugging Presentation

The data structures to be visually inspected are mainly dynamic data, namely, those built up during program executior through dynamic links, e.g. lists, trees, etc., as well as dynamic calling sequence of functions. For simpler data structures such as basic data types, simple records, dbxtool is powerful enough to offer sufficient diagnosis information. Yet with flexible type operations, no clear distinction can be made. For example, even for a simplest integer variable, by & operation, a link pointing to it is obtained. There could be operationr between this link and other compatible pointer variables. So the visual debugger should be flexible enough to let debugger users select data objects to be visually inspected.

Several typical data structures are used in programming, such as stack, list, graph. They will not cause much difficulty to be visualized in the forms familiar to people, if we do know such data structures which are being processed, as in data structure animation. For a visual debugger, however, it is very inconven ient (if not possible) to ask a user to indicate the data structures used in his/her program. A visual debugger can get data structure information only from the type definitions and variable declarations. Through addressing analysis, the data structures constructed by pointer references can be represented in the conventional forms. But for those constructed by index reference, for example, a stack composed of an array, as the storage, and an integer, as top element indicator, there is not a reasonable way to detect the stack structure. Therefore, this kind of data structures are not subjected to be suitably visualized.

In decorator generator, a variable with data type to be visually presented is captured, and a corresponding visual template is assigned to it. A visual decorator is attached to the variable each time its left value is changed. For variable i, the attached decorator will look like

```
                        if(i_vf)  v_show(i_template);
```

here i_vf is a flag controlling activation of v_show(), and v_show() is the function which manages generation of visual objects to be shown in visual area. It finds the visual template for variable i, fills in present value of i, checks links from and to i for dynamically linked data objects, and provides the result to the layout procedure.

Decorator generator maintains a visual library which records the templates for variables caught from the original program, as well as the operations needed to manipulate the templates. The library is used by visual decorators. It is also needed for dynamical visual inspection during debugging througn dbxtool's display and call commands.

The visual objects created by v_show() are floating ones. How to really arrange them in the visual area depends on the layout procedure. This procedure maintains a coordinate system for visual area, and arranges in it the data objects and links. There are conventional visual representation for some commonly used data structures, such as list, tree, as we see in a textbook. But in decorator generating, there can be no distinct indication of data structures, only types of variables are available. To keep visual representation of data structures the same as the conventional representations is not always easy. The visual effect depends on dynamic creating procedure of the data structures.

The typical program development procedure is that first the program is edited, then it is compiled and debugged. If any error is reported during compiling or debugging, the user will come back to his/her source program and modify it correspondingly. Dbxtool has incorporated in it the make facility to compile program within dbxtool environment, so only two stages are involved from a user's viewpoint. To attach visual decorators, however, an additional stage of decorator generator is needed, causing a little inconvenience. This problem is easily bypassed by batch processing. A batch file, still named make, is created, putting together decorator generator and make commands, so that the users can still work in the same manner. Moreover, using dbxtool's file facility, the program appeared in the source window can still be the user edited program with correct line indices. So the decorated program can be regarded as been hidden from the users.

Dynamic behavior is very important to a debugger. A good debugger offers various means of diagnoses to the users, helps them better understand executing procedure of the program, and quickly identify and locate the bugs. A debugger should offer the users certain kinds of control over the debugging procedure. For a visual debugger, the requirement is the same. To the visual part, it is not enough only passively visualizing data structures in a fixed manner. In this work, dynamic manipulation of visual data corresponding to text-form debugging information is offered, all the operations are fitted into the same set of dbx/dbxtool commands.

A data window for visually showing data structures is created when debugging starts. It shows only a portion of the whole visual area. The horizontal and vertical scroll bars are offered for the users to view other part of the area. The arrangement of the visual objects in the data window is decided by the layout procedure, but users can require more detailed presentation of some integrated data structures. For example, show the content of an array.

Another window showing dynamic calling sequence is also created if the users want to watch it. This window gives the hierarchy of function calls and indicates current executing functions.

Users can not directly control the layout of the visual area, but they have total freedom to decide what to be shown in this area. When the program is executed under debugger, a decorator is silent until a user issues a command to activate it. The implementation is through a control flag assigned to the decorator. The flag takes initial value 0. The user can set it to 1 using dbxtool's set command, and reset it to 0 to turn off visual display. A user can also show visual data by using a display or call command at any position of the program if the data is active. All of these are in the same way as a variable is textually displayed in dbxtool.

The difference between textual and visual representation is that, for textual output, no context relationship between data and historical sequence are concerned, the output is simply a text stream, giving present values of the variables. But for visual data, spatial and logic relationships have to be considered. For example, two objects can not overlap at the same location; the following appearances of the same data object can not create a new visual object, but modify the value of existing one. The management of visual area is getting more complex when scope rules of variables are considered.

A limited, informal semantic checking can be achieved by the visual debugger. The type checking in C is not complete. For example, C does not apply range checking for array index, though the bugs related to range checking are often difficult to detect, especially when pointer indexing is used. With dbxtool, only the content of indexing variables can be inspected, not visually expressive. With data window, since data structures are integratively represented according to the logical structures, it will be easier to visually find out of range errors of the array indexing.

A prototype is implemented on SUN3/60 workstation. In the present implementation, UNIX system functions vi, cc and dbxtool are used as editor, compiler and debugger, respectively. Separated compilation is not offered, only a single C file can be processed.

V. Conclusion and Discussions

This paper presents a new strategy to visualize a debugger tool. The existing tools are fully exploited to realize visual functioning. One interesting aspect is that by using program generation technique, the debugger visualization can be easily implemented without understanding of interior structure of original debugger. This provides an effective and efficient way to attach new characteristics to an existing software package. It is possible chat the same technique be used to other application of visualization.

The visual debugger is obtained by program generating technique, where only the generation of decorated program is language dependent. Not much effort is needed to extend it to a multi-language visual debugger. For example, to implement a Pascal visual debugger, we need only to rewrite lexical rules for LEX and grammar rules for YACC, and use compiler pc instead of cc.

A symbol table is required to record data types used in the program. This duplicates the symbol table created by the compiler with -g option. Since the symbol table in internal data structure, without inquiring into the compiler, the duplication can not be avoided. Yet with current trend in offering a standard compiler symbol table to the outside users, the decorator generator can expect to use this table when a new compiler offers it.

To be a fully functioned visual debugger, there are more things to be considered. First is to incorporate separated compilation, which is necessary to develop large application program. Another is to allow incomplete visual information. In visual presentation, spatial and historical relationship are important. Yet the users may randomly require visual display without concern of such relationships. The debugger should accept these incomplete requirements and give indication where the visual information is missing. Still another concern is the layout. To be visually attractive, a powerful layout procedure is needed. It should be able to arrange visual objects in away familiar to the users. It will be better if the users can have certain control of the display objects and select their preferred way of presentation.

Acknowledgement

Thanks are due to Manzur Alam Shaheed for his work in designing and implementing the prototype system.

References

1. Barnett M. J., Barnett S. J., Animated Algorithms, McGraw Hill Inc., Singapore, 1988.
2. Chang S. K., et. al. (eds.), Visual languages, Plenum Press, 1986.
3. Chang S. K., Visual languages: A Tutorial and Survey, IEEE Software, pp.29-39, January 1987.
4. Chang S. K.(ed.), Principles of Visual Programming Systems, Prentice-hall International, 1990.
5. Herot C. F., et al., An Integrated Environment for Program Visualization, in Schnider H. J. and Wasserman A. I.(eds.), Automated Tools for Information Systems Design, North-Holland Pub., 1982.
6. Ichikawa T., Chang S. K. (eds.), Special issue on Visual Programming, IEEE Trans. on Software Engineering, Vol.16, No.10, pp.1105-1197, October 1990.
7. Reiss S. P., Interacting with the FIELD Environment, Software-Practice and Experience, Vol.20(SI), pp.SI89-115, June 1990.
8. Sadahiro I., et al., VIPS: A Visual Debugger, IEEE Software, pp.8-19, May 1987.
9. Seminar B. K., Robson N. R., An Iconic Description Language: Programming Support for Data Structure Visualization, SIGCHI Bulletin, Vol.22, No.1, July 1990.
10. Debugging tools for the SUN Workstation, Sun Micro Systems, 1986.

Interactive Steering Using the Application Executive

Brian Bliss
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

In this paper, we describe software developed for interactively extracting and modifying typed data structures within an executing program, a package dubbed the *application executive*, or *ae* for short.

The application executive incorporates much of the functionality of the Unix breakpoint debugger *dbx* [Uni87b, Lin90] or *gdb* [Sta89a], the breakpoint debugger distributed by Free Software Foundation, namely the ability to interpret arbitrary expressions and access to data objects in different stack frames. It further supports run-time definition of new data types and data objects, and extended flow-of-control constructs.[1] There is a major difference between ae and conventional breakpoint debuggers: conventional debuggers exist as a separate process, with a different address space, whereas ae is simply a library compiled with the user's code. This allows the user the freedom to modify ae's internal data structures at will, for compiled code to call the ae interpreter and for the interpreter to return values to the compiled code, and for the user to arrange for the input text stream to come from an arbitrary source. It allows the user to create complex data structures (possibly containing pointers) in a program's address space at run-time and to pass them as arguments to compiled routines. It does not interfere with processes that communicate using signals, nor does it rely on the operating system to context switch back and forth between the debugger and the application, providing for more efficient data access. One disadvantage is that the user's program can possibly corrupt ae's internal data structures, although this rarely happens when it is used in a manner similar to conventional breakpoint debuggers.

The application executive uses C as the source for its interpreter; there is no need for the user to learn a new programming language in order to use it. Very few nonstandard constructs were added to the C interpreter. Instead, the user calls functions in the ae library (*libae.a*) using the host machine's native calling convention for I/O, control of scoping, etc. It retrieves symbolic information from a debug symbol table in the executable just as dbx does, although dbx incrementally retrieves the required information, rather than relying on a single pass to translate the information into the appropriate format as ae does. This portion of ae is referred to as the *stab scanner*.

When called from a C or Fortran application, ae provides an amount of interactive control over the application equivalent to the interpretive environments used with Lisp and other symbolic languages. This control has its cost: it is expected that the user is intimately familiar with the C language, and if ae is used with compiled Fortran code, the C equivalent of the Fortran data types and calling conventions

[1] Looping constructs are planned for future versions

The input stream may be preprocessed through the C preprocessor[2] *cpp* [Uni87a] to simplify the entry of complex code fragments.

With the addition of a signal handler and various debugging routines, the application executive can form the basis for various parallel and distributed debugger configurations. We describe several such configurations for different computer architectures.

The software has been ported to various machines running the UNIX[3] operating system or variations thereof: Sun Microsystems' Sparc[4] workstations, for use with the native C compiler or the GNU C compiler, *gcc* [Sta89b], the Alliant FX/Series[5] Computers, and the Alliant FX/2800. On parallel architectures, separate instances of the interpreter may be executed by simultaneously running threads.

[2] cpp's internal buffering must be modified if it is to be used in an interactive mode

[3] UNIX is a trademark of AT&T, Inc

[4] Sparc is a trademark of Sun Microsystems, Inc., 2550 Garcia Ave. Mountain View CA 94043

[5] Alliant, FX/Series, and FX/2800 are trademarks of Alliant Computer Systems Corporation, One Monarch Drive, Littleton, MA 01460.

Contents

1 Usage

The ae interpreter is usually invoked through the routine **fae()** or **sae()**:

```
int fae (file, return_loc, routine_name, ...)
    FILE *file;
    char *return_loc;
    char *routine_name;

int sae (string, return_loc, routine_name, ...)
    char *string;
    char *return_loc;
    char *routine_name;
```

where

file or **string** is the text stream from which the interpreted C code is read.

return_loc is a pointer to the space allocated for the return value. Data objects are returned from the interpreter using the **return** statement, a standard C construct. If no value is returned, **return_loc** may be null. **return_loc** is declared as **char *** and not **void *** as the latter construct is not accepted by certain C compilers.

routine_name is the name of the routine to interpret. If **routine_name** is null, any code appearing in the input stream is interpreted and we return to the caller only after a **return** statement is encountered. Otherwise, only the routine whose name matches **routine_name** is interpreted and we return to the caller after evaluating the body of the procedure, or earlier on a **return** statement. Unnamed routines, i.e. blocks of code surrounded by only brackets, are always interpreted, and we do not return upon exiting such a block.

extra parameters Any arguments following **routine_name** are passed to the interpreted routine. They are ignored if **routine_name** is null.

return value **fae()** returns the number of errors encountered while interpreting the text stream.

1.1 A Simple Example

Consider the following program, "**test.c**":

```
#include <stdio.h>
int global = 42;
static int static_global = 55;

main (argc, argv)
    int argc;
    char **argv;
    {
        static int static_local = 22;
        ae_stab (argv[0]);
        fae (stdin, NULL, NULL);
    }
```

The program is compiled with -g so that debugging symbol information is included in the executable, linked with libae.a or libae_g.a[6] and run.

```
ae_stab (argv[0]);
```

invokes the stab scanner, entering the type and address of all data objects and routines and type names into ae's internal symbol tables. Several diagnostic messages will be printed (and possibly some error messages) during this process:[7]

```
scanning 436 link symbols
parsing 751 debug symbols
test.c:
ae_ntrp_ref.c:
ae_stab_ref.c:
751 debug symbols parsed
```

[6] libae_g.a is the version of ae that has itself been compiled entirely with debugging symbol information.

[7] ae_ntrp_ref.c and ae_stab_ref.c are the two library files in libae.a that have been compiled with the -g option. If the user had compiled with libae_g.a, all of the application executive would contain symbolic debugging information, resulting in more symbols and filenames in the symbol tables. On Berkeley UNIX executables, the debug and link symbols reside in the same table, so the number of symbols in the debug table and in the link table are the same. The examples presented here were run on an Alliant FX/2800, with System V executables.

Once the interpreter is invoked, a diagnostic message is printed:[8]

```
entering application executive
```

Any text the user enters now is interpreted as C source code:

```
{ _printf ("%d\n%d\n", global * 2, test_c‘static_global); }
84
55
```

The source code must be surrounded by brackets, as it is considered to be the body of an unnamed function. The function declarator optionally may be present; it is required if parameters are passed to the interpreted routine. Since **printf()** exists in "**libc.a**", which does not normally contain symbolic debugging information, it is necessary to call **printf()** with the ld symbol table entry as opposed to the debug symbol table entry. The names of the ld symbol table entries for C identifiers are prepended with an underscore.[9] Since **static_global** is private to the compilation unit **test.c**, and in the entire program there may be many compilation units referring to an identifier by that name, it is necessary to resolve the potential ambiguity in a fashion similar to dbx: the identifier is prepended with with **<filename>‘**, where underscores are substituted for illegal characters in **<filename>**. Likewise, **static_local** may be referenced by the name **main“static_local**. See Subsection 1.4.6 for a more complete description of access to non-global identifiers.

1.2 The Stab Scanner

The symbol table scanner, or stab scanner, is invoked with the routine **ae_stab()**:

```
int ae_stab (filename)
    char *filename;
```

[8] Diagnostic messages can easily be turned off by resetting the global variable ae_silent to 0

[9] An ld symbol table entry does not contain type information, and if we were to use the return value of **printf()**, it would be necessary to cast **_printf** to a pointer to a function returning the appropriate type. An ld symbol table entry for a data object behaves much like an array of characters. It is promoted to **char *** when passed as a parameter or used in most expressions. Internally, it is said to have *location* type. See Subsection 1.2

where

filename is the name of the executable file. This is usually equivalent to **argv**[0] in the formal argument list of the main program, if the standard naming convention is followed.

return value ae_stab() returns the number of errors encountered while scanning the executable's symbol table.

Symbolic information for all routines and data objects which exist in a compilation unit compiled with the **-g** option is entered into ae's internal symbol tables where it may be accessed by the interpreter (and by the rest of the program). Error messages, which may be indicative of a programming error, an error in the compiler-generated symbolic debugging information, or type information inconsistent between compilation units, may be emitted during this process.

The stab scanner also creates entries for the linker symbols, which have an underscore prepended to a C identifier or an underscore prepended and another underscore appended to a Fortran identifier. These symbols are said to have *location type*, and their type is coerced to **char *** when used in an expression. A location may also be used to call a function if no value is to be returned from the call. In this case, the user must insure that the value of the symbol in question is indeed the address of a subroutine.

Should the definition of a type name differ between two or more compilation units, the former declaration takes precedence and subsequent definitions are not entered into ae's internal symbol tables, although data objects declared only with a later definition of the type name correctly assume the latter type. Global variables whose types differ across compilation units are handled in the same manner, regardless of which declaration(s) contained the **extern** storage class and which declaration actually reserved storage for the data object.

Usually, type information that is inconsistent across compilation units indicates a programming error. Should the stab scanner resolve the conflict in an undesirable fashion, one may use casts to achieve the desired data typing. dbx avoids this problem by considering type names private to each compilation unit. If this were to be the case with ae, it would greatly complicate the usage of the interpreter.

Errors encountered by the stab scanner may be resolved in one of three ways. For most errors, the identifier in question is simply discarded, and the scan continues with the next stab string. Errors concerning improperly nested include files or blocks cause the scanner to terminate and return to the caller of **ae_stab()**. Finally, the stab scanner contains the same internal error checking mechanism as described in Subsection 1.4.8 that may cause a *fatal error* to terminate program execution entirely.

1.3 Manual Insertion of Compiler Defined Objects

If the user wishes to avoid the overhead of the stab scanner, he may manually enter symbols for the desired data objects and data types by using the routines **sae_declare()** and **fae_declare()**:

```
int sae_declare (string,...)
    char *string;

int fae_declare (file,...)
    FILE *file;
```

where **string** or **file** contains the C source code necessary to declare the objects whose addresses follow it in the argument list, e.g.

```
int i, a[2];
sae_declare ("int i, a[2];", &i, a);
```

creates *static* symbol table entries for the integer variable **i** and the array **a**. If a data object declared in this manner is dynamically allocated, it is important to remove the symbol using **ae_remove_symbol()** before storage for it is deallocated. See Subsection 1.4.7.

1.4 The Interpreter

The interpreter accepts a subset of the C programming language, as defined by Kernighan & Ritchie [KR88]. It is implemented in an entirely syntax-directed scheme; expressions are evaluated as they are parsed, that is, no intermediate expression trees are explicitly formed. Currently, the C subset consists of declarations, expressions (with function calls), blocks, functions, and the **if** and **return** statements. Missing are loops and other flow-of-control constructs. In addition, the language has been extended with the **typeof** operator (described in Subsection 1.4.5), the **typedec** declarator (described in Subsection 1.4.4), and various intrinsic functions (described in Subsection 1.4.5).

When the interpreter is invoked, a *task record* containing space from which to dynamically allocate objects, the semantic stack, and other internal data objects needed by the interpreter is allocated on the run-time stack, and a pointer to this record is passed throughout the entire call sequence. Thus, the interpreter may call itself recursively, or several threads of control may execute the interpreter simultaneously.

1.4.1 Declarations

In addition to examining and/or modifying compiler-defined variables, the user may create interpreter-defined variables interactively and modify them in the same manner. This section describes the actions that take place when the interpreter encounters a declaration for a data object that it cannot currently reference. In such a case, new storage is allocated and the necessary symbolic information entered into ae's internal symbol tables.

Global Variables If the interpreter parses a declaration for a data object not previously declared outside the scope of any block, storage for the object and its symbolic representation is statically allocated, and the symbol inserted in a static hash table.[10] The user may enter and exit the interpreter and the object will remain in scope, unless manually removed by the user.[11] If further declarations for an object by the same name are encountered outside of any block, they are assumed to be redeclarations of the same object, and the specified data types must agree.

Dynamically Allocated Local Variables If the interpreter encounters a declaration for a data object (not declared **static**) inside the scope of a block, then storage for the object and its symbolic representation are dynamically allocated. The data object stays in scope until the end of the current block, at which point the storage is reclaimed (unless the object was declared using the **static** storage class). Multiple declarations for dynamically allocated data objects are not allowed.[12] Declarations inside a block need not precede executable statements; their order may be intermixed.

[10] On parallel architectures, statically allocated symbols are accessed in critical sections of code. See Subsection 1.4.9.

[11] This can be accomplished by calling the routine **ae_remove_symbol()** as described in Subsection 1.4.7.

[12] Unless the user has manually removed the symbol by calling the routine **ae_remove_symbol()**. See Subsection 1.4.7.

Static Local Variables When the keyword **static** is used to declare a data object local to a block, then the semantics depend upon whether we are interpreting a named routine or are within an unnamed block of code. In either case, given a declaration

static <type> object;

two symbols are created: one by the name **object**, and another by the name **<routine>‘<block>‘object**, where **<routine>** is the name of the routine being interpreted[13] and **<block>** is the current block number. **<block>** is omitted if it is zero, i.e. if we are in the outermost block of the routine.

If we are interpreting a named routine, storage for the object and the symbol **<routine>‘<block>‘object** is statically allocated, and the symbol may be referenced by that name as if it were a global variable. The object may be referred to by the name **object** until the end of the current block. If a subsequent invocation of the interpreter encounters the same routine and block,[14] **object** will refer to the same storage location; the semantics specified by the C standard are preserved.

If we are within an unnamed block of code, the symbol **‘<block>‘object** only remains in scope until we exit the outermost block.[15] **object** remains in scope until the end of the current block. This was done for the following reason: if looping constructs are added, the interpreter may look up the symbol **‘<block>‘object** to find the appropriate storage location when an inner block is reentered,[16] but since unnamed blocks are considered distinct, the object cannot be referenced from within another unnamed block.

Static Global Variables When a declaration for a previously undeclared data object that includes the **static** keyword appears outside the scope of any block, then the semantics depend upon whether the filename of the input stream has been set. This may be done using the **#line** compiler directive in the standard fashion. If it has not been set or has been reset to the empty string, the declaration is considered private to the current invocation of the interpreter. The symbol is dynamically allocated and remains in scope until the interpreter exits.

[13] **<routine>** is the empty string in the case of an unnamed block.

[14] The code for the routine must be identical to what was previously encountered.

[15] We are in an unnamed block, so <routine> is not specified.

[16] This is impossible in the absence of looping constructs.

If the filename of the input stream has been set to non-empty string, and a **static** declaration such as

static <type> object;

is encountered, then the symbol **<ntrp_unit>'object** is created, in addition to **object**. **<ntrp_unit>** is identical to the filename of the input stream, where all characters not allowed in C identifiers have been replaced by underscores. An underscore is prepended to the entire string should it being with a digit. **object** remains in scope for the current invocation of the interpreter, whereas **<ntrp_unit>'object** is treated as a global variable. Should a subsequent invocation of the interpreter have the filename set to its previous value and encounter a **static** global declaration for an object by the same name, then the new declaration refers to the same object. Of course, the specified data types must agree.

1.4.2 The return Statement

If a **return** statement is executed, the interpreter exits and control returns to the caller. If an expression follows the **return**, it is evaluated the result copied to the space pointed to by the second argument of **fae()** or **sae()**. In this case, it is an error for the second argument to be null; an error message is displayed, and the **return** aborted.

Control will also return to caller when the end of a named block of code is encountered or when the end of the input text stream is reached.

1.4.3 The typeof Operator

The **typeof()** operator returns a pointer to the internal type descriptor associated with a data object. **typeof** may also take a type name as an argument, and therefore cannot be implemented as an intrinsic function without using macro substitution on the input stream to insert a data object which is casted to the specified type. The type descriptors used by ae have data type **union ae_TYPE**, which has the equivalent typedef name **ae_type**. Consequently, the result of a **typeof** expression has type **ae_type ***. Like the **sizeof** operator, parentheses are only required around the operand if it is a type name:

typeof <expression>

or

typeof (<type_name>)

<type_name> is defined in Kernighan & Ritchie [KR88], Section A8.8. In the current implementation **<expression>** is fully evaluated. See Subsection 1.6.

The typeof operator is commonly used in conjunction with the library routine **ae_fprintf()** (described in Subsection 1.4.7) to examine the type of a data object:

```
float f;
{ ae_fprintf(&_iob[2], "%T", typeof f); }
0xc21118
type: ae_D_FLOAT
size: 4
```

stderr is usually defined as a macro for (**&_iob[2]**) in **stdio.h**. No symbolic information exists for macros, so we supply the expanded test directly. Alternatively, the input stream may be preprocessed with the C preprocessor, *cpp*. cpp buffers its input stream in large blocks; this must be changed if expressions are to be evaluated as they are entered.

1.4.4 The typedec Declarator

The **typedec** keyword may be used in a declaration to define a new data object using a type descriptor supplied by the user:

```
typedec (<expression>) <dec_list>;
```

declares all data objects in **<dec_list>** to have a base type[17] specified by **<expression>**, which must point to a valid type descriptor. The argument is checked to be sure it is of type **ae_type *** and has a non-null value. An invalid type descriptor that passes these checks may cause a *fatal error* to be subsequently encountered (See Subsection 1.4.8). The **typeof** keyword

[17] The identifiers in <dec_list> may be declared as a pointer to, an array of, or a function returning the base type, or a combination thereof.

Designing CDS : an On-line Debugging System for the C_NET Programming Environment.

Pierre MOUKELI

Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 LYON cedex 07
FRANCE
e-mail: moukeli@frensl61.bitnet
moukeli@lip.ens-lyon.fr

I. Introduction.

One of the most challenging problems in designing a distributed debugging system (DDS) is the implementation of debugging functionalities in a given programming environment.

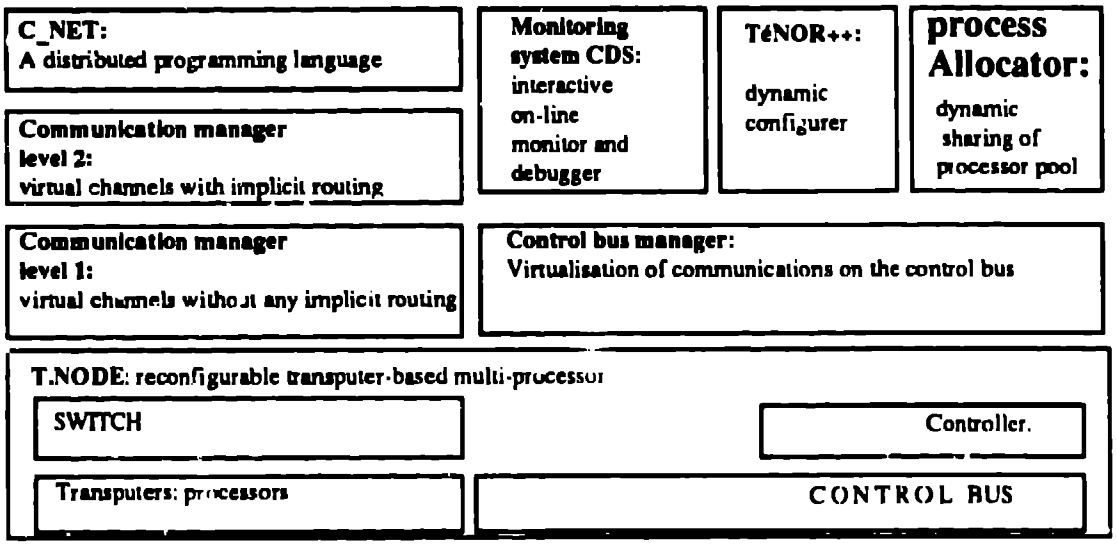
A possible solution consists on decomposing the debugging environment into domains, and analyzing the interactions between the DDS and each domain. Cheng, Plack and Manning [CBM] suggested to decompose the debugging environment into three domains: specification, execution, and observation. The first domain is the source code in which the programmer specifies the expected process behaviour and data to be collected. The second domain concerns the relations between the DDS, the operating system and the user processes under debugging. The third domain deals with the possible interactions that can take place between the user and the debugger. DDS is an agent which manages the interactions among these three domains [CBM].

We have considerated an additional domain dealing with the hardware on which the DDS can be implemented. The later induces a set of constraints which should be met by the DDS implementation. Designing a DDS consists to find out suitable methods which solve or minimize the problems related to DDS interactions with its environment, and the hardware constraints.

When designing the C_NET Debugging System (CDS) according to the above idea, we proceed in five steps described in this paper. Section II presents the environment according to which CDS was designed. Section III describes the five steps proceeded to design CDS. Section IV is an overview of CDS; and then, we conclude by showing problems we faced when designing CDS.

II. C_NET programming environment.

For about two years, the "Parallel Languages and Systems" group of the LIP laboratory has been carrying out the development of a high level programming environment for reconfigurable transputer-based multi-processors. The environment suggested aims to virtualise the use of such machines. It has been developed on a SuperNode and is composed of a set of software components presented below (fig. 1) :



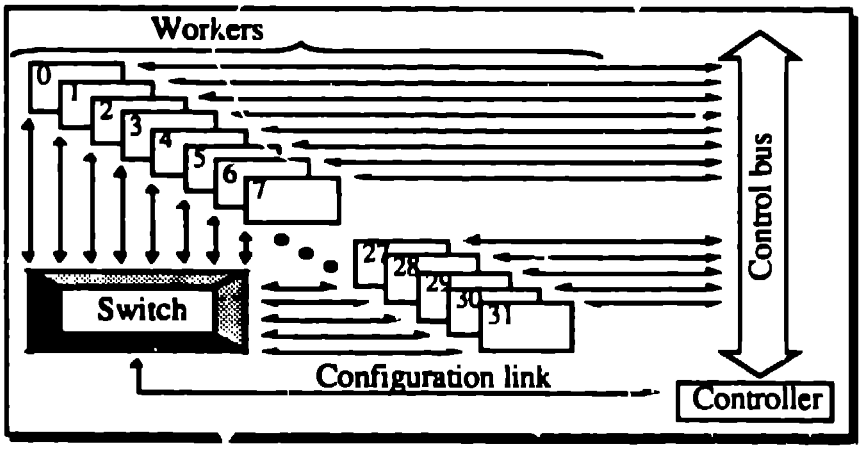
-Fig. 1- Organization of the C_NET programming environment

T800 transputer processor. The T800 developed by INMOS. is now a well-known single chip processor [J]. It has four asynchronous communication links and an internal on-chip memory (4 k bytes, 50 ns). The links are 20 MHz circuits. An additional component, the 64-bit floating-point unit lies across the top of the chip. The transputer has two priority levels. In particular, processes running in the high priority level cannot be descheduled unless they reach a scheduling point (e.g. timer halt, communication point).

SuperNode. SuperNode is a transputer based multi-processor machine (up to 1024 transputers). It provides a crossbar switch for dynamic reconfiguration. All the transputers in the SuperNode are connected to a controller transputer via a master-slave bus (fig. 2). We use this bus to transport debugging observations in order to avoid overloading transputer communication links.

Control-bus manager. The control-bus manager virtualises communication on the control bus. It offers a high level interface that allows the processes running on the control transputer and the worker transputers to share the control-bus. More details can be found in [M2]. Within the C_NET programming environment, the bus manager is essentially used by CDS [M1]. On-line observations carried by the bus do not interfere with the messages flowing across the switch in the main communication network. The bus manager is also used by the dynamic configurer (phase synchronisation and communication of permanent variable values).

Dynamic configurer TéNOR++. The dynamic configurer TéNOR++ allows programs with variable topologies to be developed. TéNOR++ can be accessed through the use of two C extension languages. More details can be found in [AB].



-Figure 2- The SuperNode.

Debugging system CDS. This system is described in section IV. It allows process behavior observations to be carried out in real-time when a program is executed. Observations are drawn across the control bus. They are displayed through a multi-window interface which allows inter-active debugging to be completed.

Process allocator. The process allocator is intended to allow the machine to be shared between several independent applications (processors, switch and control bus).

Communication manager. The C_NET programming environment permits the development of statically-defined phase chaining programs. Each phase is assumed to be run on a particular topology, generally specified by the programmer, which is intended to fit as best as possible the communication needs of the phase. Detailed information on TéNOR++ can be found in [ABB].

A distributed programming language: C_NET. The C_NET language issued from our motivation to design a language supporting both object-oriented (C++) and concurrent programming (CSP-OCCAM). C_NET tries to solve appropriately the difficult problem of interfacing class and process notions. The solution suggested in [A] fits the data encapsulation principle and is fully compatible with the inheritance mechanism.

III. CDS designing steps.

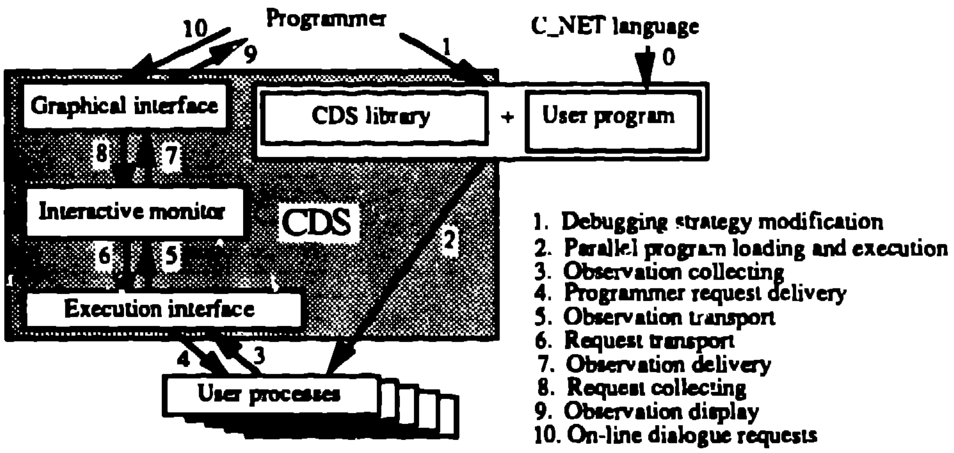
When designing CDS according to the above environment, we proceed in five steps.

Step 1. Defining the basic requirements for CDS; these requirements were defined according to the features of the C_NET environment : process halting and interactive context modification (process variables, channels, system variables), trace collecting and event time-stamping with a global software clock.

Step 2. Analyzing the interactions of CDS with its environment; starting from the requirements, we outlined the objects exchanged between CDS and each of the domains above presented. Then, we defined suitable interfaces to handle the object exchanges. This gave us an interaction graph, and then, the general structure of CDS. Figure 3 shows the issue of that step : the interaction graph, where numbers represent the type of interactions for object exchange between CDS and each domain.

In short, in the specification domain (C_NET language), interactions consist to insert debugging code in the source program. The suitable interface is a library of functions. In the execution domain (user processes), objects are process events and programmer requests to running processes. Our interface is a set of communicating processes and functions. The observation domain is the programmer controlling process execution by means of a graphical interface, and modifying source code according to the observed behaviours of these processes. Finally, the hardware (shown in grey in figure 3) is the support of these interactions.

Step 3. Identifying the problems incurred by these interactions; these problems depend on both the type of the objects exchanged between CDS and each domain, and the interface in which these interactions take place. So, we considered all the exchanged objects systematically.



-Figure 3- Interaction graph.

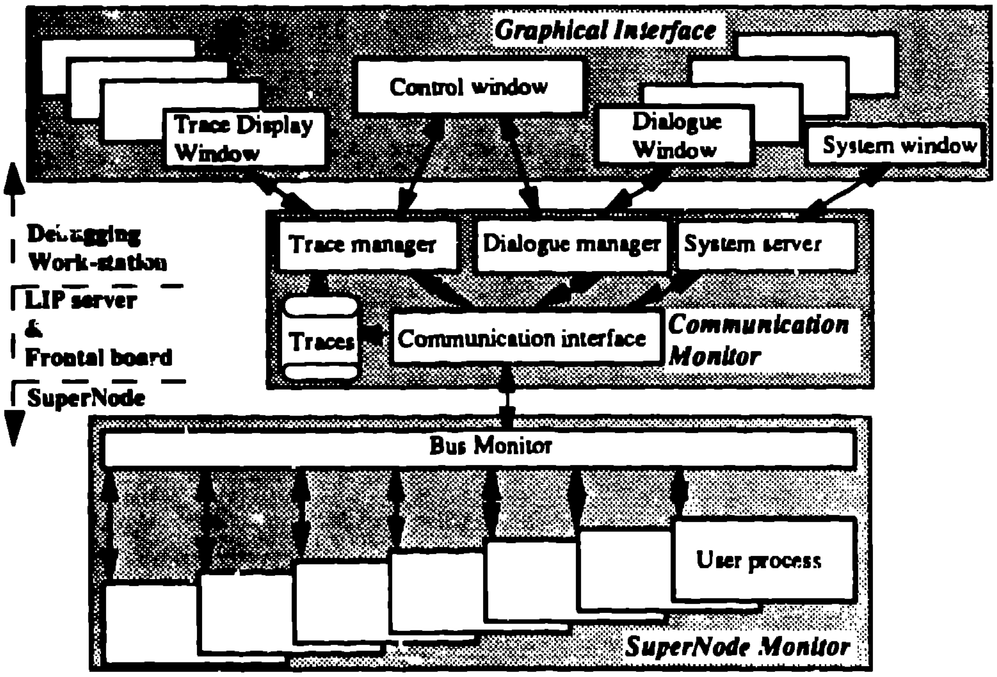
For instance, in the specification domain (C_NET language, fig. 3), objects are process behaviours, and our interface consists of a library of functions. Examples of related problems are readability and redundancy of debugging code. In the execution domain, objects are process events; interface is a set of monitoring functions whose utilization can incur problems such as interference with user process scheduling, and probe effect.

Finally, new problems, such as congestion, transparency, and overflow, arise. They are related to the structure of CDS (e.g. interactive software monitor, fig. 3) and the hardware supporting it (e.g. SuperNode control bus which is used to convey debugging observations).

Step 4. Analyzing implementation choices. They are related to the hardware. Therefore, we first analyzed the constraints generated by the hardware (e.g. measure of bus performance as well as the access time to standard input-output and shared resources). Then, we found means to solve or minimize the problems identified in step 3, meeting the hardware constraints. For example, as building a transparent software monitor is impossible, we managed to make this monitor fair. We also designed a control bus manager (fig. 1) in order to share that bus between CDS, TéNOR++ and user processes. In the same way, we reduced the congestion incurred by the interactive monitor when conveying debugging observations.

Step 5. Implementing CDS. This step is the application of solutions found in the fourth step. It consists in analysing in more detail, the different components of CDS and specifying the cooperation between them. For example, communication between the interactive monitor and the graphical interface takes place in Unix sockets, because they run on different computers. Another example is the bus sharing between trace conveying and clock synchronization. This step also consists in solving programming problems incurred by the implementation of choices made in the fourth step.

Figure 4 shows the currently implemented architecture of CDS. Arrows represent the interaction between CDS processes.



-Figure 4- CDS implemented architecture.

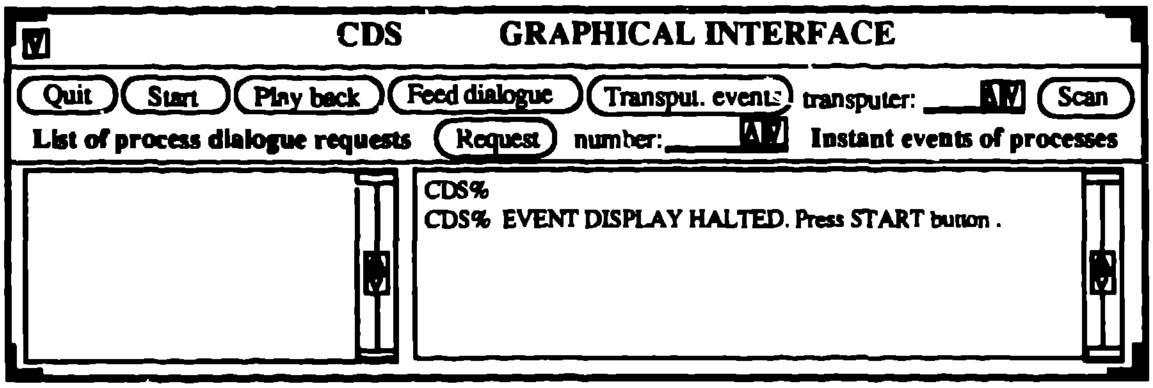
IV. Presentation of CDS.

CDS is an on-line multi-function debugger, implemented on the SuperNode at the LIP laboratory. It has been operational since March 1991. Currently, CDS consists of three components (fig. 4) :

(a). An *interactive software monitor* to collect, convey and manage debugging observations. It consists of a set of concurrent processes distributed on each SuperNode transputer, the frontal board, the SuperNode server, and the debugging work-station. The interactive monitor uses the control-bus manager to convey debugging observations.

(b). A Xwindow based *multi-window graphical interface* to handle dialogues, to display run-time events and to re-examine traces. It runs on the debugging work-station. We currently use Sun Spark work-stations at the LIP laboratory. Figure 5 shows the main window which displays real-time events and enables the programmer to open dialogue windows (one for each process in breakpoint) and trace windows (one for each transputer).

(c). A *library of functions* devoted to set breakpoints and to specify trace collecting points at source code level.

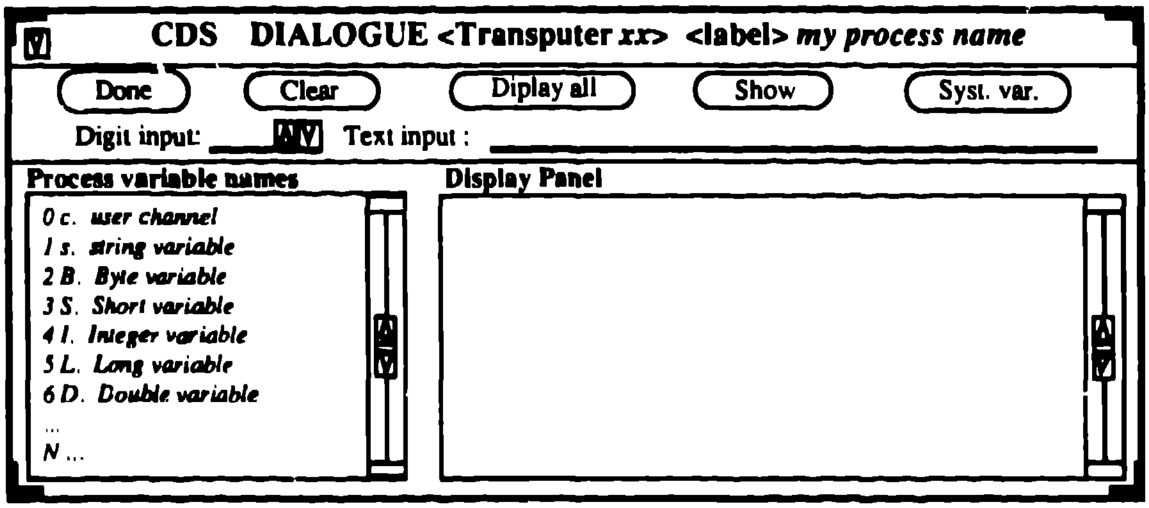


```
CDS%
CDS% EVENT DISPLAY HALTED. Press START button .
```

-Figure 5- CDS control window.

CDS currently provides :

(a). Dialogue handling between the user and processes at breakpoints. A dialogue is a communication protocol which enables the programmer to observe and modify the process context (process local variables, channels and system variables). When a process reaches such a breakpoint, the dialogue protocol sends a request to the programmer, and the later can modify the context of that process by means of a dialogue window (fig. 6).

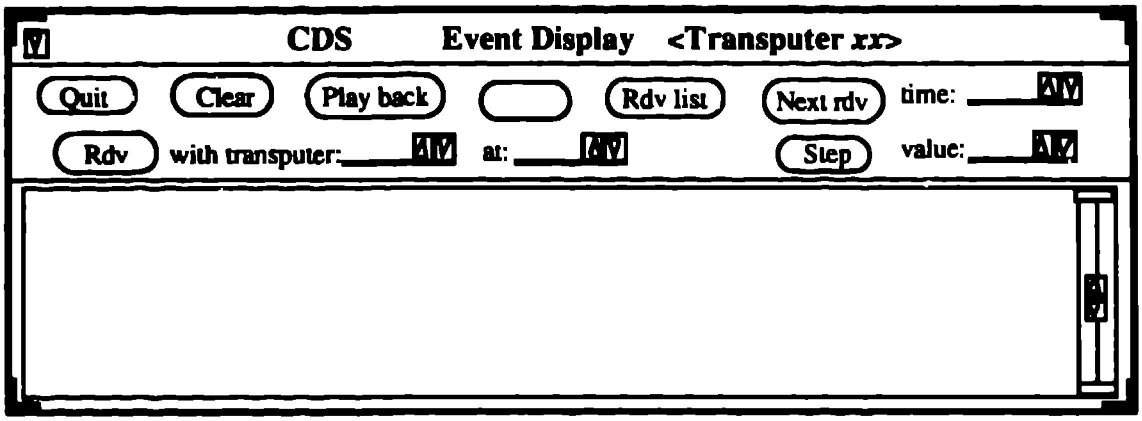


```
0 c.  user channel
1 s.  string variable
2 B.  Byte variable
3 S.  Short variable
4 I.  Integer variable
5 L.  Long variable
6 D.  Double variable
...
N ...
```

-Figure 6- CDS dialogue window.

(b). A spy process (scanner) which can collect meaningful information on running processes. There is one scanner on each transputer. The programmer can dialogue with them at any time to collect information specified for each spied process in special tables.

(c). Traces collecting. Run-time events can be observed in real time in the control window (fig. 5). The programmer can also open a trace window (fig. 7) for each transputer. CDS also provides an event filtering mechanism, which consists on the synchronization at given dates, of event display in several trace windows. This mechanism, currently simulated, is intended to be used with the global clock. Traces are also stored in files in which they could be re-examined using the CDS graphical interface.



-Figure 7- CDS trace display window.

Other functions are under development, among which, event time-stamping and ordering using a global software clock, and general breakpoint setting.

V. Conclusion.

The current version of CDS enables the programmer to debug his program at a low level. Now, we are looking for technics to specify high level process behaviours, such as general breakpoints. The most important difficulty we encountered when designing CDS was how to identify the problems incurred by the interactions between CDS and its environment (step 3). We did not find an efficient strategy for that purpose. In fact, these problems are not the same from one programming environment to another. However, there is a set of problems faced by most of debugger implementors, such as the interference with user processes, fairness, congestion, readability , and cooperation between the components of a DDS. Another non-trivial problem is how to determine the basic requirements according to the programming environment and most of the bugs everyone wants to finds out.

Acknowledgement. Special thanks to Nora Boukari, Luis Trejo, Jean-Marc Adamo and Léa-Flore Kanga for their help.

References.

[A] J.M. Adamo,
Extending C++ with Communicating Sequential Processes,
Research Report LIP RR 90-25, Transputing Conf. Santa Clara, Calif., Ap. 91.

[AB] J.M. Adamo, C. Bonello
TeNOR++: A Dynamic Configurer for SuperNode Architecture,
Proc. of COMPAR'90 conf., Sept. 10-13 1990, Zurich, Switzerland,
Springer Verlag.

[ABB] J.M. Adamo, J. Bonneville, C. Bonello
Virtualising Communication in the C-NET Programing Environment,
13th OUG Technical meeting, Univ. of York, Sept. 18-20 1990.

[CBM] W.H. Cheng, P.B. Black, E. Manning
A framework for Distributed Debugging
IEEE Software Jan. 1990, PP. 106-115

[J] C. Jesshope
Transputer and Switches as Objects in OCCAM
Parallel Computing 8 North-Holland, 1988, pp 19-30

[M1] P. Moukeli,
Etude et Mise en Œuvre du Système de Mise au Point de Programmes C_NET
Langages Paralleles,
Research report, LIP RR 91-22. Jul. 1991.

[M2] P. Moukeli
Présentation d'un Moniteur de Communication sur le Bus de Contrôle du
SuperNode
La Lettre du Transputer, Sept. 1991, PP. 33-50.

DBL : An Interactive Debugging System[*]

Mukkai S. Krishnamoorthy
Anastasios D. Anastasiou

Rensselaer Polytechnic Institute
Troy, NY 12180

November 11, 1991

Abstract

The objective of this paper is to discuss the design and implementation of an interactive debugging system for a functional language which serves as the host programming language for a software package suitable for graph theory applications. The design and implementation of the programming language is beyond the scope of this paper[2]. But in designing and implementing the command language for the debugger, we have tried to make its format as similar as possible to the format of the host language while keeping its syntax clear, logical, simple and flexible. Additionally, a brief description of graphical interface, that is introduced in the system for visualization of graphs, is given. The design and implementation of the debugging system explained in this paper could be applied in the development of a debugging system for any language.

1 Introduction

The debugging system implemented supports a range of commands wide enough to fulfill the requirements of a good interactive debugging system. The user may define breakpoints at lines or function calls within the program which cause execution to be suspended at these points. After execution is suspended, other debugging commands can be used to analyze the progress of the program and to diagnose errors detected. Then execution of the suspended program may resume. The user can step through the program executing one line at a time or follow the execution of the program even inside other functions called within the program being debugged. The user can also trace variables so that when a variable changes its value, the new variable value will be displayed. System commands and instructions of the source language can be executed from within the debugger environment. It is also possible to display the source code of the program being debugged, complete with statement numbers, information about the program itself (name, size in number of lines, current line of execution), and information about

[*] Supported in part by the NSF under grant CDA-8805910.

the status of the debugger (breakpoints, variables traced). A function is provided for displaying the values of variables and it can also execute function calls of the source language and display their results. Additional capabilities include removing breakpoints defined previously, removing variables so that they will no longer be traced and setting aliases for all the debugger commands to make the command language syntax simpler [3]. Lastly, an on-line help facility is provided to assist the inexperienced or occasional user.

To improve the friendliness and power of the debugger further, graphic capabilities were included. By introducing graphics in the debugging system, it can be transformed from a linear, command-line debugger to a visual debugger. Most of the data structures used in graph theory, especially graphs, were always best represented graphically; thus a visual debugger helps the users to visualize the representation of the data structures defined in their programs. Additionally, they will be able to trace variables visually, and as a result they will improve their ability to notice and understand the changes on the data structures used.

2 Significance of the Problem

It is obvious among all programmers that there is a great need for a useful interactive debugging system when writing and executing programs. The main purpose of this debugger is to provide programmers with facilities that will help them in testing and debugging their programs and, as a result, make the process of executing a program less painful and less time consuming. In this particular area of application (graph and set theory), having to deal with large graphs and sets makes it hard to follow the execution of a program and trace those structures. In addition, as a program grows in size and complexity, the importance of executing it in a small number of instructions each time, becomes more apparent.

The debugger can also be used as a learning tool. Many times, programs provide only the final result of the application of an algorithm to a problem, omitting important intermediate results. A user who is not familiar with the algorithm can follow the execution of the program step by step with the use of the debugger, discover how and why data structures change their values to produce the final result, and thus understand the logic of the algorithm much better.

3 Design Considerations

In designing and implementing the debugger, we considered the following two parameters : the user and the source programming language.

The first parameter suggested that the command language should have a clear, logical and simple syntax and be as flexible as possible. The commands were chosen to have a meaningful, easy-to-remember names (HELP, LIST, PRINT etc) . They are simple rather than complex with as few parameters as possible. Commands are automatically checked for syntax and logical errors and if any are found, the debugger provides a meaningful error message. However, in order to simplify the command language, if the error is not crucial and the debugger is able to figure what was the intending command, it will execute it. Default values are also provided for most of the parameters. The flexibility of the command language lies on the fact that the use of punctuation characters such as parentheses, quotation marks, slashes and semicolons was minimized. Also, through the use of aliases, the user can abbreviate commands and make the command language even simpler.

The debugger is also related to the source programming language and to the interpreter that executes it. The debugger command language was kept similar to the source programming language as long as it was not increasing the complexity of its syntax. As a result, the debugging environment is familiar and friendly to the user. Additionally, as programs of the source language were executed by an interpreter, the debugger itself works as an interpreter. In fact, the program being debugged is executed by the source language interpreter and most of the functions written for the interpreter are used by the debugger[1].

4 The Graphical Interface

The graphical interface provides the user with the ability to display directed and undirected graphs as well as the weights of arcs and vertices. Graphs can be displayed in different ways such as with vertices placed symmetrically, horizontally, vertically and circularly. In addition, other special graphs such as outer planar and bipartite can be displayed in an appropriate manner[2]. The form that the graph will be drawn is selected from a pull-down menu or entered as a command. The graphical interface allows the user to move vertices and arcs and to change their color and weight. It also provides the user with the ability to add and delete vertices and edges. All these operations to manipulate a graph can either be selected from the menu or entered as a command.

The graphical interface is very important for the debugging facility. At any point of the program execution, the user is able to display a graph and understand how the program execution affects a graph. The changes that occured in the graph (edges and vertices deleted or added) become more noticeable. We believe that a user can understand the structure of a graph much easier by visualization rather than by textual representation and that is why we consider the graphical interface a very important feature of the debugging facility. With the use of the graphical interface, the user is able to draw a graph, notice the changes that occured in the graph and understand the changes much better. In addition, it makes the whole system more user-friendly and easier to use.

5 Interpreter Modifications

Since the debugger executes the language commands by invoking the interpreter, the interpreter has to be modified so that it performs the interpretation and at the same time, it complies with the debugging commands issued. In a way, the interpretation must be synchronized with the execution of debugging commands. The only debugging commands which affect the interpretation of the source code are those which suspend and resume program execution, namely **stop** for suspension of execution and **next**, **step** and **continue** to resume execution. The interpreter, after being invoked by the debugger, interprets the source code until a breakpoint defined by a previous command suspends its execution. At this point, the user can enter any other debugging command which is directly interpreted by the debugger until the interpreter is invoked again by a **next**, **step** or **continue** command and execution is resumed.

To implement this switching of the interpreter between suspension and invocation, we used two more flags to control the invocation: the *step_flag* is set to true if the interpreter executes in step mode and the *next_flag* is set to true if the interpreter executes in next mode. The suspension is controlled by the use of breakpoints. The only place in the execution where the interpreter is suspended is during the read of a new source line. At this point, if there is a breakpoint defined at this line, execution is suspended and control passes to the debugger. Also if the debugger operates in step or next mode, again execution is suspended and both flags are set to zero. If a **continue** command is issued from the debugger environment, the debugger invokes the interpreter and passes control to it. This process continues until execution terminates or the user exits the debugger.

With these features added, the interpreter has the additional capability to suspend and resume its execution depending on the debugging commands issued by the user. In other words, the control of execution of the program switches between the interpreter and the debugger and the whole process is controlled by the debugger.

6 The Command Language

The current version of the debugger supports the following command language:

run

starts execution of the program to be debugged, for the first time.

rerun

repeats execution of the program. All debugging specifications(breakpoints, traces) used in the previous session are saved and used in the new session.

help*[command]*

invokes the on-line help facility. If a debugger command is specified as a parameter, the help file for this command is displayed. It defaults to a brief list of all the debugger commands with explanations.

list *[line1,line2]*

displays lines of the source code of the program being debugged. The default lists the current execution line plus 10 more lines after; **list***line1* displays the line specified by the parameter; **list** *line1,line2* displays the lines in the range specified.

continue

resumes execution of the program after a breakpoint.

exit, quit, bye
returns control to the system from where the debugger was invoked.

system *(system_command)*
executes a system command from within the debugger environment.

next
executes the next line of the source program without entering any functions called in that line. It suspends execution after that line is executed.

step
executes the next line of the source program, stepping into functions called in that line. It suspends execution after that line.

print {*variable* | *function_call*}
the current value of the variable is displayed if the parameter specified is a variable. If the parameter is a function_call, the function is executed and the result is displayed.

stop {**at** *line* | **in** *function_name*}
stop at *line* defines a breakpoint at the specified line; **stop in** *function_name* defines a breakpoint at a function. The execution is suspended immediately before the specified line or a line containing a call to a function specified by *function_name*.

trace *variable*
traces the changes of the values of the specified variable. Whenever there is a change to the value of *variable*, the debugger will display the variable, its new value and the line number at which the change had occared.

remove { **stop at** *line* | **stop in** *function_ name* | **trace** *variable*}
remove stop at *line* removes a breakpoint defined previously at the specified *line*.
remove stop in *function_call* removes a breakpoint previously defined in the specified *function_name*.
remove trace *variable* suspends the trace of the specified *variable*.

alias *identifier command*
sets *identifier* as an alias for the specified debugger *command*.

show /**stop** | **trace** /
when the parameter is **stop**, the debugger will display all current breakpoints (for lines and functions) whereas when the parameter is **trace**, it will display all variables specified for trace. The default displays all information mentioned above.

lila ***lila_function_call***
executes a function call of the source language from within the debugger environment.

display ***[*****line | file | size*****]***
displays some information about the program being debugged. The **line** option displays the number and the source code for the line currently being debugged. The **file** option displays the filename where the program resides and the **size** option displays the size of the program in number of lines. The default displays all information above. This command can be easily extended to display additional information if it is required.

7 Data Structures

This section provides an outline of the most fundamental data structures used in the design and implementation of the debugger.

The most important data structure used is the **symbol table**. The symbol table is a closed hash table in which each bucket points to a list of table entries. Each entry contains a pointer to a structure, which itself contains a value, and a list of variable names pointing to that value. It also contains a pointer to the next entry in the hash table.

An **activation record** structure is used to hold the following information about each activation:

- a pointer to the entry of the function within the function table
- a pointer to the first line of code of the function
- a pointer to the next line of code to be executed by an activation
- a pointer to a structure which identifies the data type returned by the function
- a pointer to the local symbol table of the function
- the position within the next line of code to be executed
- a flag that represents whether the next command will come from a file or the terminal
- the size of the function code in lines
- the name of the file that contains the source code for the function
- a pointer to the next activation record

To implement breakpoints at lines more effectively, a structure named **PROGRAM** was defined with the following fields:

- a string which holds the source code for the line currently debugged

- an integer specifying the line number of the line currently debugged
- an integer taking value 0 or 1 to denote whether there is a breakpoint defined at the line currently debugged
- a pointer to the next line to be debugged

Two hash tables were designed to implement breakpoints in function calls and to trace variables. For the function breakpoints, each bucket of the hash table holds the name of the function, and for tracing variables each bucket of the hash table holds the name of the variable to be traced. To avoid collisions, each bucket has a pointer to a link list of function names or variable names which happen to collide in that bucket.

8 Further Extensions

The debugger at its present version has enough capabilities to aid users in testing and debugging their programs in a very good degree. However, there are more capabilities that can be included to improve this debugging system and make it more powerful, simpler and easier to use. This section suggests some of these capabilities that can be added.

As a first step, the debugger can be improved by including some functions which will allow users to define conditional breakpoints and breakpoints after a fixed number of instructions has been executed. With conditional breakpoints, the user will define conditional expressions that will be continually evaluated during the debugging session. The program execution will be suspended when any of these conditions become true. Similarly, the user may define a fixed number of instructions to be executed, after which the program execution will halt. A menu-driven debugger will reduce the amount of information a user has to enter and remember. Menus will have titles to identify the tasks they help perform and they must have an equivalent action to the linear debugging language. In other words, there should be complete functional equivalence between commands and menus.

9 Algorithm

In this section, we will provide a general algorithm used to solve the problem. Each individual command of the interpreter is executed by one or more separate function. Refer to the Appendix A for a description of each function.

```
    create the token table
    create the command table
    read program to be debugged into memory
loop:
    prompt the user for command line
    get command, parse it and analyze token
    if the command is
```

```
        bye, exit, quit : exit the loop and stop
        stop : set the breakpoints
        run : start interpretation
        rerun : execute again
        list : display the specified lines of the source code
        help : invoke the help facility
        system : execute the system command
        remove : remove the specified breakpoints or traces
        print : display the result of its parameter evaluation
        next : execute the next line. Ignore function calls.
        step : execute the next line. Step into function calls.
        cont : resume execution until a breakpoint is encountered
        trace : mark the specified variables for tracing
        show : display breakpoints and traces
        lila: execute a language command from within the debugger
        display: display the required information about the program
        alias : set the specified alias
    goto loop
```

10 Conclusion

This paper describes the design and implementation of an interactive debugging system for a programming language to manipulate graphs and directed graphs. We have successfully used this implementation in debugging the planarity algorithm. In addition, it provides a brief description of the graphical interface which allows users to visualize graphs. It also explains how the design considerations of this system can be applied for the design of any debugging system.

A Appendix

This appendix provides a brief explanation for the most important functions written for the debugger. For each function, we provide its name, its parameters, and a brief description of the action that it performs.

Name: create_token_table
Parameters: none
Purpose: inserts each of the reserved words of the interpreter and its mnemonic constant into a table.

Name: create_command_table
Parameters: none
Purpose: inserts each of the commands of the debugger and its mnemonic constant into a

table.

Name: read_prog
Parameters: program name
Purpose: reads the program to be debugged into the memory. For each line, it stores the source code, the line number and a breakpoint flag.

Name: cget_token
Parameters: none
Purpose: gets the next token from the command line and saves it in the global variable *ctoken*

Name: cget_char
Parameters: none
Purpose: returns the next character from the command line.

Name: clexan
Parameters: none
Purpose: performs lexican analysis of a token. The global variable *ctok-type* is an integer constant and it can be a mnemonic constant denoting either a string constant, a reserved word, an integer constant, a constant denoting a function call or an identifier.

Name: hash
Parameters: name to be hashed
Purpose: applies the hash function to its argument and returns the position in the table to be inserted.

Name: dbl_stop
Parameters: none
Purpose: defines a breakpoint at a line or in a function. When the breakpoint is at a line, it finds the appropriate line and sets its breakpoint flag to 1. When the breakpoint is in a function, it inserts the function name into a hash table which stores all functions where breakpoints were set.

Name: stop_line
Parameters: none
Purpose: inserts a breakpoint at a specific line. Starting from the first line of the program, it finds the line where the breakpoint should be set and sets a breakpoint.

Name: stop_function
Parameters: none
Purpose: after checking the syntax of the command line, it calls the appropriate function which will insert the function name specified for a breakpoint into the *funcstops_table*

Name: insert_in_funcstops
Parameters: a character string representing the function name where a breakpoint was defined
Purpose: performs the insertion of the function name where the breakpoint was set into the *funcstops_table*

Name: is_stop_function
Parameters: a function name
Purpose: returns 1 if the function name specified by its argument was set for a breakpoint and 0 otherwise.

Name: dbl_trace
Parameters: none
Purpose: interprets the DBL command **trace**. It checks the syntax of the command line and calls the appropriate function which will insert the structure name specified for trace into the *trace_table*

Name: make_trace
Parameters: the name of the variable to be inserted for trace
Purpose: inserts the structure name specified by its argument into the *trace_table*

Name: is_trace
Parameters: name of the variable to check if is for trace
Purpose: returns 1 if the structure name specified by its argument was specified for trace and 0 otherwise

Name: dbl_remove
Parameters: none
Purpose: removes breakpoints and traces. When a breakpoint at a line is to be removed, its corresponding breakpoint flag is set to zero. When a breakpoint in a function or a trace is to be removed, the function name or variable name is deleted from the corresponding hash table

Name: rmv_line
Parameters: none
Purpose: It removes a breakpoint at a line. After specifying the line whose breakpoint will be removed, it sets its *stop* field to zero and essentially removing the breakpoint

Name: rmv_fctn
Parameters: none
Purpose: It removes a breakpoint in a function call. After locating this function in the *funcstops_table*, it removes its entry and thus removing the breakpoint in this function

Name: rmv_from_funcstops
Parameters: function name to be removed from the breakpoints
Purpose: it removes the function name specified by its argument from the *funcstops_table*

Name: rmv_trace
Parameters: none
Purpose: removes a variable that was declared for trace from the trace table (if this variable was really for trace) or it prints an error message otherwise

Name: bye
Parameters: none
Purpose: interprets the DBL commands **bye** , **quit** and **exit**. After assuring the syntactic correctness of the command line, it terminates the current session of the interpreter

Name: dbl_help
Parameters: none
Purpose: interprets the DBL command **help**. After assuring the syntactic correctness of the command line, it opens the appropriate help file and displays its contents on the screen.

Name: syst
Parameters: none
Purpose: interprets the DBL command **system**. After checking the syntax of the command line, it extracts the portion that will be executed and pass it to the system command 'system' which executes it

Name: dbl_list
Parameters: none
Purpose: interprets the DBL command **list**. After checking the syntax of the command line, it lists the range of lines specified in the command line.

Name: dbl_print
Parameters: none
Purpose: interprets the DBL command **print**. The argument for the print command could be an integer constant, a string constant, a variable structure, or a function call. After checking the syntax, it determines which one of these cases exists and prints the result

Name: dbl_lila
Parameters: none
Purpose: interprets the DBL command **lila**. It interprets any *lila* command from within the debugger environment.

Name: dbl_next
Parameters: none
Purpose: interprets the DBL command **next**. It steps thought the program without entering any function calls. At this point, only the *next_flag* is set to indicate that debugging will be in next mode

Name: prompt
Parameters: none
Purpose: it prompts the user for a command, parses the command, and calls the appropriate function to execute the command. If the command is either **step** or **next** or **cont** it breaks out of the loop, to continue the program interpretation

Name: dbl_run
Parameters: none
Purpose: interprets the DBL command **run**. It checks the command line, it initializes variables and flags and starts the interpretation of the program

Name: dbl_alias
Parameters: none
Purpose: interprets the DBL command **alias** to set aliases for the DBL commands. After checking the syntax of the command_line, it inserts the alias with its mnemonic constant into the hash table of function token

Name: dbl_show
Parameters: none
Purpose: interprets the DBL command **show** to display all current breakpoints and traced variables. First it displays all line breakpoints by examining the stop field of each line, then it shows all function breakpoints by searching the *funcstops_table* and finally it displays all variables under trace by searching the *trace_table*

Name: dbl_display
Parameters: none
Purpose: interprets the DBL command **display** which displays some useful information about the file being debugged

References

[1] L. L. Beck, *System Software*, 2nd Edition, Addison-Wesley, 1989.

[2] M. S. Krishnamoorthy, T. Spencer, E. McCaughrin, *GraphPack: A software system to manipulate graphs and directed graphs*, R. P. I. Computer Science Technical Report, 1989.

[3] *Unix Programming Manual - dbx system*, 1987.

X Window System Interface for CDBX

Peter A. Rigsbee

Cray Research, Inc., Eagan, MN

Abstract

CDBX is a dbx-based debugger for CRAY UNICOS systems. CRI initially implemented the basic line-oriented version of CDBX, and in 1989 added an optional X Window System interface. This paper describes the manner in which this interface was added, which is unlike the approach taken by developers of `xdbx` or `xgdb`. The paper also describes how this approach simplified subsequent work to allow the X interface to run native on a workstation.

Background of CDBX

From the user's point of view, CDBX appears to be nothing more that an implementation of the BSD dbx debugger. As with most dbx implementations, Cray Research has added a number of extensions. Key improvements include better Fortran support (including array syntax), support for multitasking and segmentation, improved links between symbolic and absolute debugging (new commands that translate between machine addresses and symbols), and improved on-line help and error messages.

Internally, however, CDBX is quite unusual. Part of CDBX consists of code from the BSD 4.3 version of dbx, and part consists of code from an earlier Cray Research debugger known as DRD. This unusual combination of software is complicated by the fact that dbx is coded in C and DRD was coded in Pascal. But by combining code from existing projects, it was possible to quickly develop and deliver a product to customers. In fact, development of CDBX began in March 1988, and a beta test was held at several customer sites only five months later, in August 1988. The first release of CDBX was included in the UNICOS 5.0 release in March 1989. It would not have been possible to have carried out a traditional "port" of dbx to the CRAY in the same timeframe.

Development of the X Window System interface for CDBX began in January 1989, and was first released in October 1989. Development started with prototype software provided by one of the Cray developers working on the X Window System itself. Using the prototype as an experimental vehicle, a design was developed which remains in use today and will be described in this paper.

User Level

Attached to this paper are two screen dumps from CDBX sessions, showing the key components of the X Window System interface. The first shows the basic window displayed by CDBX. When a user starts up the X interface, he gets a window containing several sections. These include two groups of buttons and three display windows.

The top group of buttons (`quit`, `help`, and `interrupt`) control CDBX itself and cannot be removed or changed. The remaining all generate CDBX commands, and are completely under user control. The user can add buttons, remove buttons, or even remove the entire group and build his own set. Certain buttons operate on text selection, some on line selections, and some bring up pop-up menus containing additional choices. User-defined buttons can be CDBX commands or user-defined aliases, and can make use of text or line selections.

There are three text sub-windows within the CDBX window. The top window is called the "information window" and shows status information about the debugging session. This information is updated as needed and shows information obtainable with CDBX status commands. The middle window is a read-only "source window", which displays the current file. The user can set breakpoints or select variables or other expressions by using the mouse; these selections can then be referenced by command buttons. The bottom window is a read-write "session window", which is nothing more than a line-oriented CDBX session. When a command button is pushed, for example, the associated debugger command is echoed in this window followed by any output for that command. Any keyboard input to the CDBX window is treated as input to the session window and is passed to the debugger.

There are also several "pop-up" windows that can be displayed as a result of certain actions. Some of these are shown in the other screen dump. The `help` button brings up an on-line help window, containing functionally-oriented usage information. A CDBX `display` command allows a user to identify certain variables whose contents should be updated each time the debugger reaches a breakpoint or otherwise "stops". These variables are displayed in a separate window. And several other CDBX commands (such as `sh` and `gripe`) result in pop-up windows being presented to the user.

We have found that people use the X interface in a number of ways. It is particularly useful for new or casual users, who appreciate the default set of command buttons highlighting the important commands. Many experienced CDBX users use the X interface primarily for the source display, and find themselves typing commands more often than using the buttons. Other experienced users have developed their own set of buttons that perform key or repeated functions specific to their applications. The interface is well-designed from the sense that it lets people easily use it the way they want, rather than forcing them into a particular mode of operation.

Design

Now let us look at the internal design. Unlike some public domain X interfaces, we had the advantage of being able to control the entire product. We felt there were two different approaches we could take. We could integrate the X interface with the traditional debugger, producing a windowed debugger. In this approach, the X interface is part and parcel of the debugger, producing a very powerful product, but limited to those users who are using the X Window System. Or we could produce a separate X interface, but have it work closely with the traditional line-oriented dbx-style debugger. We chose the latter approach, primarily because we had and still have a significant need for a line-oriented debugger.

The next decision was whether we should introduce a new user command for the X interface. Unlike many other vendors, we chose to use the same command to invoke either the X interface or the line-oriented debugger. This command would figure out what the user wanted and do it. It would not be necessary for the user to remember and use two different commands.

Internally, though, we kept the two parts of the product separate. When CDBX is started, a "driver" program (stored as `/usr/bin/cdbx`) starts up and examines the environment and command line to determine whether it should start the X interface or the line-oriented debugger. If the X interface is being run, the driver `execs` an executable called `cdbx.x`. This process then performs a `fork` and an `exec` with an executable called `cdbx.l`. As a result, the user has two processes running "the debugger". This split has a major advantage for the user of the line-oriented debugger; he does not have to pay for the additional memory overhead required for the X Window System code that he will not be using. There are other advantages in terms of operating system scheduling and throughput.

How do these processes communicate? There are three connections established when `cdbx.x` does its `fork` and `exec` of `cdbx.l`. A `pty/tty` pair is opened by `cdbx`, passed as a command-line option to `cdbx.x` and `cdbx.l`, and is associated with the `stdin` and `stdout` of `cdbx.l`. These connections allow commands to be sent to and output received from `cdbx.l`.

In addition, a one-way pipe is opened for data from `cdbx.l` to `cdbx.x`. This pipe is used to send packets of information for the X interface to use in updating the windows being presented to the user. These packets use a simple message format with a number of different packet types. The following list shows the packet types currently in use:

```
typedef enum {
 P_ALERT,
 P_DELETE,      /* breakpoints */
 P_DISPLAY,
 P_EDIT,
 P_EXECUTING,  /* user program */
               /* executing */
 P_FILE,
 P_GRIPE,
 P_INFO,        /* information */
 P_MENU,
 P_SH,        /* shell xterm */
 P_STARTX,  /* start: -r fails */
 P_UNDISPLAY,
 P_UNMENU,
 P_QUIT
} Packet_type;
```

P_INFO and P_EXECUTING are used to pass information to the X interface, to update the information windov/ or the cursor, and the rest are used to direct the X interface to perform some particular action (such as creating or destroying pop-up windows, adding items to pop-up menus, and so on).

Tradeoffs

The use of the pipe and the message packets has some very significant advantages. The primary one is that it helps isolate the two programs from one another, and provides a clear definition for their interface. It is not necessary for cdbx.x to execute "secret" commands to get information, nor is it necessary for cdbx.x to either parse commands entered by the user nor to parse the ASCII output from cdbx.1. Internally, it avoids the unmanageable use of global variables to silently pass information between the two programs.

The main disadvantage is it makes certain operations more difficult to implement cleanly. For example, cdbx.x is constrained to performing actions that can translate to commands. For example, with many windowed debuggers, you can double-click on a variable to see its value. This would be awkward to implement in CDBX. Another example of this is the CDBX display command, which has a quite complicated implementation (although this is not evident to the user.)

Distributed CDBX Experiment

In the spring of 1991, an experiment was carried out with the X in..rface. In this experiment, cdbx.x was divided into two executables:

- workstation component
- CRAY component

The workstation component consisted of the X interface code that handles the window events, keyboard input, and so on. The CRAY component, which was very small, consisted only of the code that did the fork/exec and set up the communications with cdbx.1.

In the workstation component, calls were then added to use two sets of fairly simple library routines. The original code that handled reading and writing from pipes and standard files was replaced by calls to library routines that handled simple message passing between two processes across sockets. And code that opened and read from source files was replaced by calls to a package providing user-configurable access to remote files. No changes at all were made to cdbx.1.

When the coding was complete, it was possible to run CDBX in a distributed manner, in which the X interface ran on the workstation (specifically, a Sun SPARCstation) and the "debugger" ran on the CRAY. This produced some interesting and encouraging results. The results were for simple [illegible] cases, consisting of a short scenario of about [illegible] CDBX commands.

First, we looked at user CPU time on the CRAY. For most users, workstation CPU time is "free", once they have acquired the workstation. A user who heavily uses his workstation CPU pays no more than one who lets it run a screen saver all day. But CRAY time is usually paid for, with either real or "funny" money, and user CPU time is often the major component of the cost. With the distributed version, user CPU time decreased significantly. With the distributed version, user CPU time was .0981 seconds. With the same session for an equivalent CRAY version, user CPU time was .5685 seconds. The difference is largely due to X Window System initialization, but there was a small time savings for each command executed as well. Now at most sites this wouldn't be much money (maybe a few cents), but over the course of many, long sessions, it would add up.

Second, we looked at user response time. These timings were all made on a fairly large memory (32 MW), two-CPU CRAY Y-MP, which was not heavily loaded. These parameters all favor the non-distributed version, since swapping of processes is not likely. As a result, the distributed version had poorer response time (an average of .320 seconds versus .146 seconds for the CRAY version). The difference was noticeable as I went back and forth between the two versions, but the response time for the distributed version was quite acceptable by itself. On a smaller CRAY system or one more heavily loaded, we would expect the response time for the distributed version to improve relative to the CRAY version, perhaps even running faster.

Third, we measured data transfers and messages. When a message is sent to the CRAY, it interrupts the system and requires the operating system to handle it. Some people feel that X Window System clients should not run on CRAY systems because of the high level of traffic caused by the many events triggered by mouse movement or keyboard entry. With our experiment, we measured the traffic with both the CRAY version and the distributed version. As expected, the distributed version showed significantly less traffic.

For example, simply starting up the debugger showed the following data. With the CRAY version, there were 71 messages containing a total of 26372 bytes of data sent from the SUN to the CRAY, and 68 messages containing 22316 bytes of data sent from the CRAY to the SUN. This traffic was primarily X Window System traffic passing a large resource file to the CRAY and initializing the complex windows in CDBX. By contrast, the distributed version showed 14 messages with only 203 bytes sent from the CRAY to the SUN, and **ZERO** messages sent from the SUN to the CRAY.

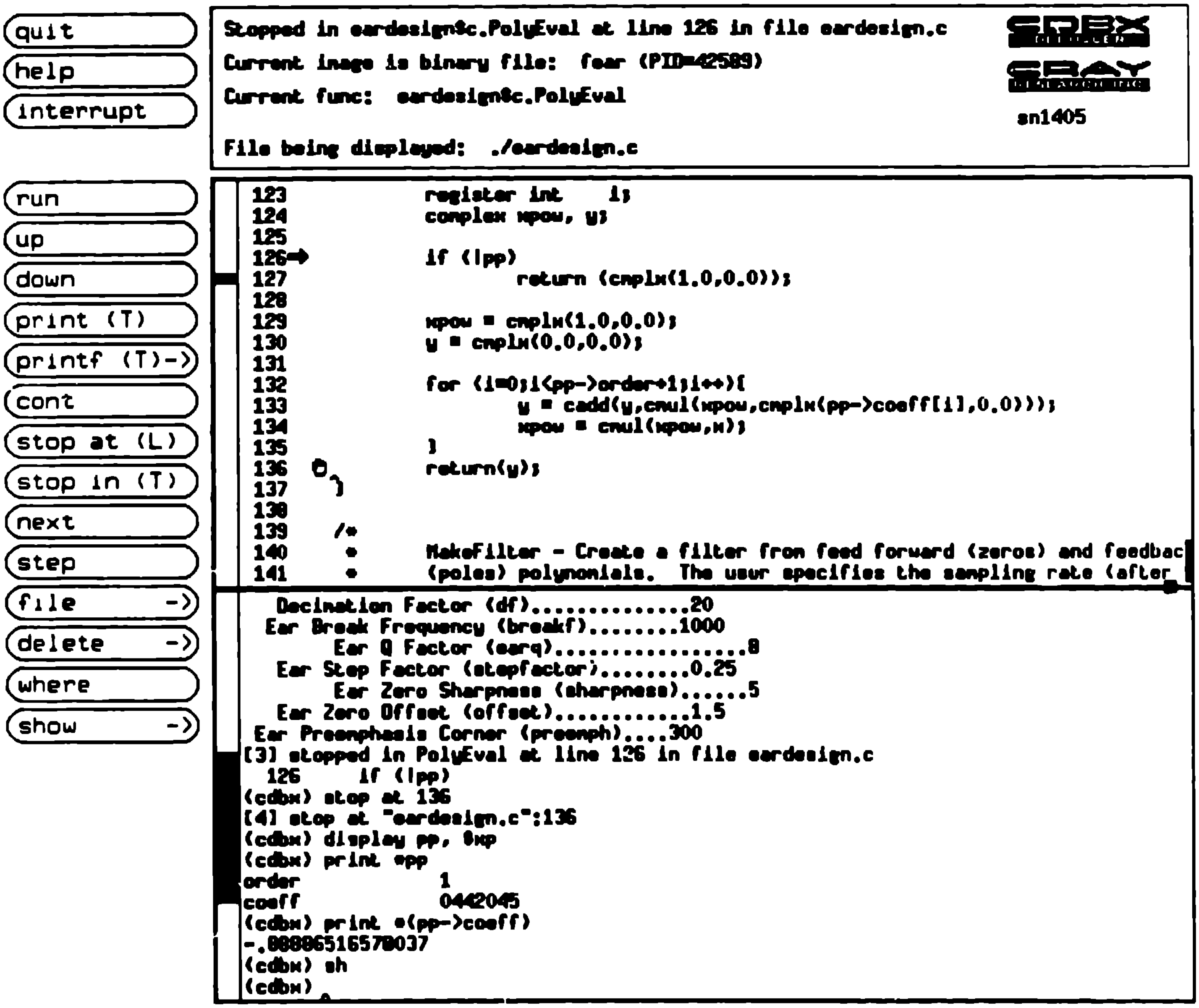
Once the session was underway, there was more traffic with the distributed version, but still much less than the CRAY version. For the simple, ten command scenario, the CRAY version saw an additional 87 messages containing a total of 12496 bytes sent from the SUN to the CRAY, and 93 more messages containing a total of 24196 bytes sent from the CRAY to the SUN. Many of these were X events resulting from keyboard entry and mouse movement. By contrast the distributed version had 10 messages (one per command) with 48 bytes sent from the SUN to the CRAY, and 56 messages with 773 bytes sent from the CRAY to the SUN.

Summary and Conclusions

The X Window System interface for CDBX has been very successful. It is heavily used and provides a simple introduction to a product with an otherwise complex interface. The design has worked well; there have been numerous enhancements to the interface since the original vesion, and they have fit in well with the original design. And the design allowed for a distributed version of the debugger to be developed with a minimum of work. While it is not clear at this time if such a product will ever be released, results show that it has the potential of reducing load on the CRAY and cost to the end user.

Credits

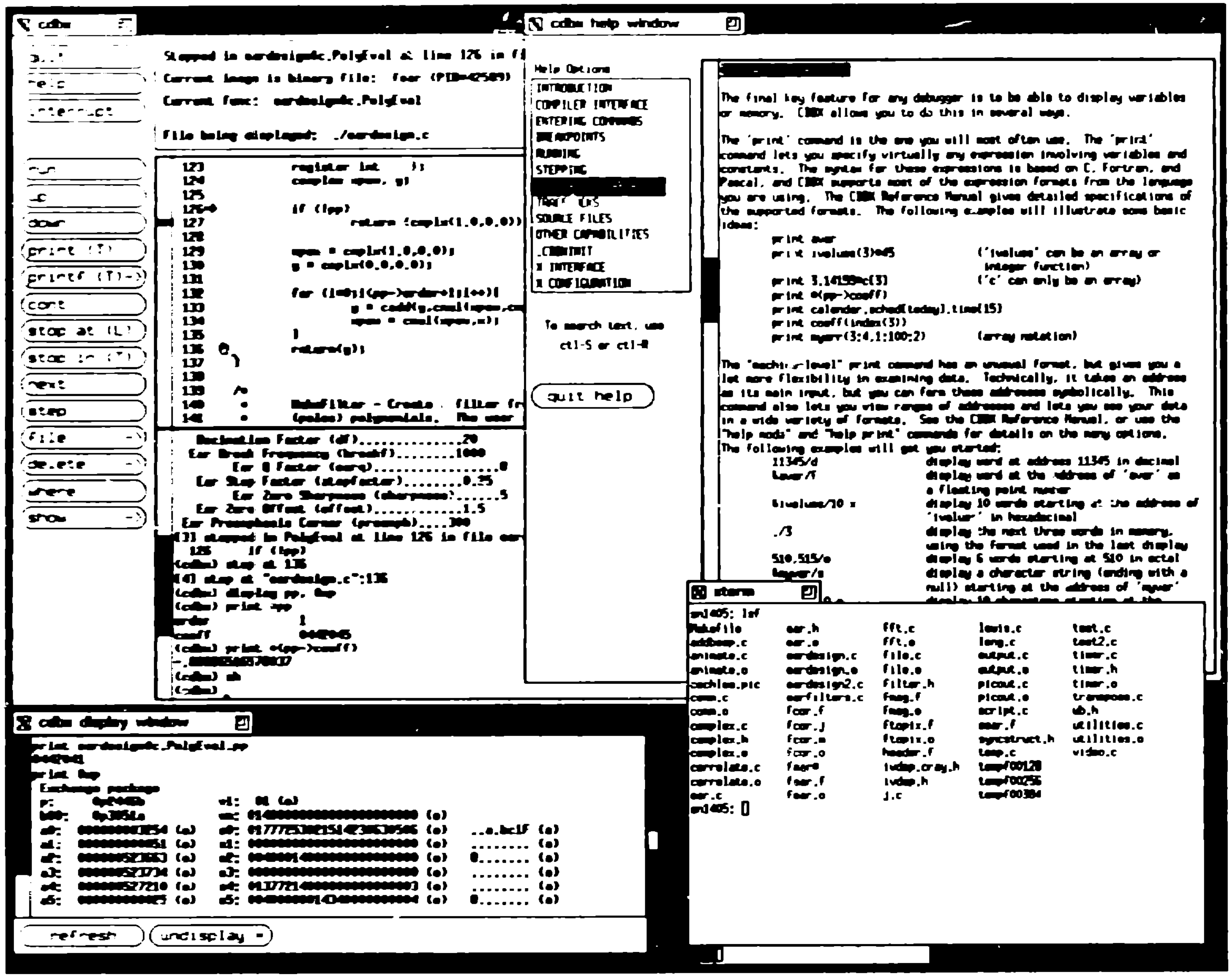
Most of the development work on the CDBX X interface was performed by Barbara Smith. She devloped the design and implemented the initial versions of the interface and several subsequent major enhancements. The prototype version of the X interface (mentioned above) was developed by John Freeman.



```
Stopped in eardesign$c.PolyEval at line 126 in file eardesign.c
Current image is binary file:  fear (PID=42589)
Current func:  eardesign$c.PolyEval

File being displayed:  ./eardesign.c
```

```
123             register int    i;
124             complex xpow, y;
125
126->           if (!pp)
127                     return (cmplx(1.0,0.0));
128
129             xpow = cmplx(1.0,0.0);
130             y = cmplx(0.0,0.0);
131
132             for (i=0;i<pp->order+1;i++){
133                     y = cadd(y,cmul(xpow,cmplx(pp->coeff[i],0.0)));
134                     xpow = cmul(xpow,x);
135             }
136  ◎          return(y);
137     }
138
139   /*
140    *        MakeFilter - Create a filter from feed forward (zeros) and feedbac
141    *        (poles) polynomials.  The user specifies the sampling rate (after
```

```
  Decimation Factor (df)..............20
 Ear Break Frequency (breakf)........1000
      Ear Q Factor (earq)..................8
  Ear Step Factor (stepfactor)........0.25
      Ear Zero Sharpness (sharpness)......5
  Ear Zero Offset (offset)............1.5
 Ear Preemphasis Corner (preemph)....300
[3] stopped in PolyEval at line 126 in file eardesign.c
  126     if (!pp)
(cdbx) stop at 136
[4] stop at "eardesign.c":136
(cdbx) display pp, &xp
(cdbx) print *pp
order              1
coeff              0442045
(cdbx) print *(pp->coeff)
-.88886516578037
(cdbx) sh
(cdbx)
```



```
Stopped in eardesign&c.PolyEval at line 126 in fi
Current image is binary file:  fear (PID=42509)
Current func:  eardesign&c.PolyEval

File being displayed:  ./eardesign.c
```

```
 123            register int    ;
 124            complex xpow, g;
 125
 126->          if (!pp)
 127                    return (cmplx(1.0,0.0));
 128
 129            xpow = cmplx(1.0,0.0);
 130            g = cmplx(0.0,0.0);
 131
 132            for (i=0;i(pp->order+1;i++){
 133                    g = cadd(g,cmul(xpow,co
 134                    xpow = cmul(xpow,x);
 135            }
 136  @         return(g);
 137     }
 138
 139     /*
 140      *      MakeFilter - Create   filter fr
 141      *      (poles) polynomials.  The user
```

```
   Decimation Factor (df)................20
  Ear Break Frequency (breakf).........1000
       Ear Q Factor (earq)......................0
   Ear Step Factor (stepfactor).........0.25
       Ear Zero Sharpness (sharpness)......5
   Ear Zero Offset (offset)..............1.5
 Ear Preemphasis Corner (preemph)....300
[3] stopped in PolyEval at line 126 in file ear
  126      if (!pp)
(cdbx) stop at 136
[4] stop at "eardesign.c":136
(cdbx) display pp, *pp
(cdbx) print *pp
order              1
coeff              0442045
(cdbx) print *(pp->coeff)
-.00000256570037
(cdbx) sh
(cdbx)
```

The final key feature for any debugger is to be able to display variables or memory. CDBX allows you to do this in several ways.

The 'print' command is the one you will most often use. The 'print' command lets you specify virtually any expression involving variables and constants. The syntax for these expressions is based on C, Fortran, and Pascal, and CDBX supports most of the expression formats from the language you are using. The CDBX Reference Manual gives detailed specifications of the supported formats. The following examples will illustrate some basic ideas:

```
        print aver
        print ivalues(3)*45            ('ivalues' can be an array or
                                        integer function)
        print 3.14159*c(3)             ('c' can only be an array)
        print *(pp->coeff)
        print calendar.sched[today].time[15]
        print coeff(index(3))
        print myarr(3:4,1:100:2)       (array notation)
```

The "machine-level" print command has an unusual format, but gives you a lot more flexibility in examining data. Technically, it takes an address as its main input, but you can form these addresses symbolically. This command also lets you view ranges of addresses and lets you see your data in a wide variety of formats. See the CDBX Reference Manual, or use the "help mode" and "help print" commands for details on the many options. The following examples will get you started:

```
        11345/d                 display word at address 11345 in decimal
        &aver/f                 display word at the address of 'aver' as
                                a floating point number
        &ivalues/10 x           display 10 words starting at the address of
                                'ivalues' in hexadecimal
        ./3                     display the next three words in memory,
                                using the format used in the last display
        510,515/o               display 6 words starting at 510 in octal
        &myvar/s                display a character string (ending with a
                                null) starting at the address of 'myvar'
```

```
print eardesign&c.PolyEval.pp
0442045
print *pp
 Exchange package
 p:        0p20460b          vl:  01 (a)
 b00:      0p30561a          vm:  0140000000000000000000 (o)
 a0:   0000000003854 (a)   s0:  0177725302151423053056 (o)   ..a.bcIf (a)
 a1:   0000000000051 (a)   s1:  0000000000000000000000 (o)   ........ (a)
 a2:   0000000523663 (a)   s2:  0000001400000000000000 (o)   @....... (a)
 a3:   0000000521734 (a)   s3:  0000000000000000000000 (o)   ........ (a)
 a4:   0000000527210 (a)   s4:  0137721400000000000003 (o)   ........ (a)
 a5:   0000000000025 (a)   s5:  0000000001430000000004 (o)   @....... (a)
```

```
ard405: lsf
Makefile        ear.h           fft.c           lewis.c         test.c
addbump.c       ear.o           fft.o           long.c          test2.c
animate.c       eardesign.c     file.c          output.c        timer.c
animate.o       eardesign.o     file.o          output.o        timer.h
cochlea.pic     eardesign2.c    filter.h        picout.c        timer.o
comm.c          earfilters.c    fmag.f          picout.o        transpose.c
comm.o          fcor.f          fmag.o          script.c        ub.h
complex.c       fcor.j          ftopix.f        sear.f          utilities.c
complex.h       fcor.m          ftopix.o        sysstruct.h     utilities.o
complex.o       fcor.o          header.f        temp.c          video.c
correlate.c     fear*           ivdep.cray.h    tempf00128
correlate.o     fear.f          ivdep.h         tempf00256
ear.c           fear.o          j.c             tempf00384
ard405: []
```

DWARF: A Debugging Standard

Janis Livingston
Software Engineer
Motorola, Inc.
6501 William Cannon Drive West
Austin, Tx 78735-8598
M/D OE112
512-891-2304
janisl@oakhill.sps.mot.com

Karen Spohrer
Software Engineer
Motorola, Inc.
6501 William Cannon Drive West
Austin, Tx 78735-8598
M/D OE112
512-891-2080
karens@oakhill.sps.mot.com

Abstract

A major limitation of current debugging techniques is the lack of a debugging language. Unix System V Release 4 (SVR4) introduced a debugging language, DWARF. A Unix International Programming Language Special Interest Group (PLSIG) is modifying and enhancing DWARF, addressing the debugging needs of the language tools community. The PLSIG hopes that interested parties will see the advantage of DWARF and conform to this defacto standard.

This paper discusses the current status of the DWARF language including:

- SVR3 and SVR4 file format comparisons,
- Communication between generators and consumers,
- Innovations which underscore DWARF's effectiveness, and
- Impact DWARF will have on compiler and debugging tools.

The new and improved DWARF debugging language will be useable on hardware platforms ranging from microprocessors to supercomputers as the debugging language of choice.

Introduction

DWARF is a debugging information format used in Unix[1] System V Release 4 (SVR4) for transmitting accurate source-level information between generators and consumers. Compilers, assemblers and linkers are defined as generators Debugging tools (debuggers, profilers, disassemblers, etc.) are defined as consumers.

Figure 1. DWARF Definition

> ***DWARF***
>
> **Debugging information format used in Unix System V Release 4 for transmitting accurate source-level information between generators and consumers.**

This paper investigates DWARF as a debugging language. However, to understand the advantage of DWARF, one must first look at object file formats and how they convey debugging information to the consumers. This paper explores the differences between object file formats and then discusses the DWARF language format. Finally the paper investigates the flexibility and extensibility of the DWARF language and how these features adds a new dimension to the debugging world.

History

AT&T Bell Labs introduced DWARF in April 1990. Shortly thereafter a Unix International Programming Language Special Interest Group (PLSIG) formed to foster the development of language tools. The goals of the PLSIG are:

1. Develop debugging standards so that generators and consumers could be developed independently and work together correctly.

2. Develop debugging standards which adequately handle the needs of most programming languages.

DWARF has been selected by the PLSIG as a debugging language through which their goals could be accomplished. The original DWARF introduced by AT&T has been refined and polished, by the PLSIG and is known as DWARF version 0. While it is possible to support any computer language with DWARF, version 0 focuses on C, C++, and Fortran. Version 1 is expected to be completed in 1993. It will add enhancements to the current definition, while retaining backward compatibility, and expanding its language focus.

Debugging Strategies

Debugging information is included in executable files. Prior to the introduction of DWARF as a debugging language, the inclusion of debugging information was not handled elegantly. Debugging information was intertwined in object files rarely providing an accurate source level picture. Strict contracts between the generator and consumers were the norm. Compilers generated special assembler directives to describe the source language. Compilers were dependent on the assemblers understanding of the special directives to pass information into the executable files. Assemblers were required to generated tags for unnamed structures in the source code, providing misleading source pictures to the consumers. Additionally this method typically supported a single high level source language. Thus, when debugging techniques changed or new language features were introduced, entire software packages were re-written. This proved to be a weak and poorly defined approach to debugging with the consumer and generator tightly coupled and totally dependent upon each other.This is very much the case in the Unix System V Release 3 (SVR3) Common Object File Format (COFF) files.

SVR4 and DWARF offer a new approach for conveying source information between generators and consumers. The new approach successfully addresses many of the problems identified with traditional methods of transmitting source information to debugging tools. The SVR4 Extensible Linkage Format (ELF) provides a flexible storage mechanism which allows generators to use the DWARF language to communicate a more detailed source level picture to its consumers. DWARF is flexible enough to handle any number of source languages, and is easily extensible to handle new debugging techniques and features. Equally important SVR4 and DWARF eliminate the number of contracts between generators, such as the compiler, assembler and linker.

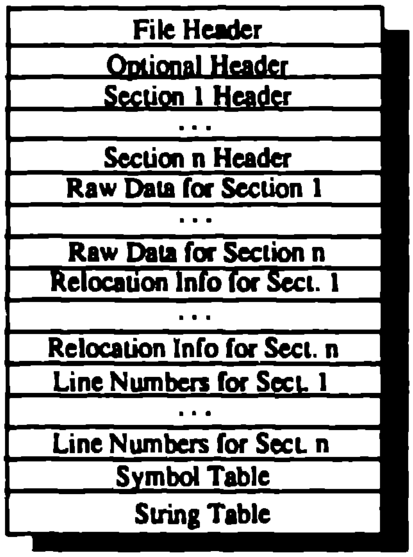
Object File Formats

COFF

As mentioned earlier, pre-SVR4 AT&T systems pass information to consumers using COFF file format. The COFF file format is based on a static design, and consists of 8 sections, 4 required sections and 4 optional sections. Figure 2 illustrates the layout of the sections in a COFF file. The sections are required to appear in the order shown. The required sections are: File Header; Optional Header; Section headers; and Raw section data. The remaining sections are optional.

[1] Unix is a registered trademark of UNIX System Laboratories, Inc. in the United States and other countries.

Figure 2. COFF File Format

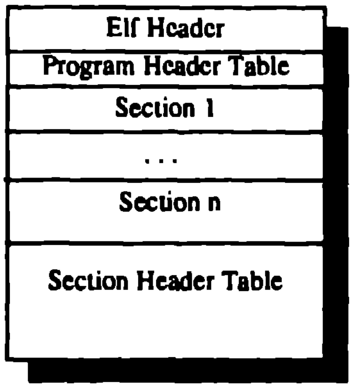


COFF file sections are traditionally used exclusively for text, data, and miscellaneous items such as comments. There is not a predefined section dedicated to debugging information. COFF based generators are limited to generating debug information that can be stored in the predefined COFF line number structure and the symbol table structure. In addition, since the symbol table and line number section holds all debugging information the generator must understand and produce a unique set of special pre-defined symbols to describe source structures such as inner blocks, and complex data structures. These unique pre-defined symbols must be recognized by the assembler so that correct information is passed to the consumers. This approach forms a highly dependent relationship between the generators (compiler and assembler) and limits the information which can be passed to the consumer. Lastly COFF was written with the C programming language in mind, it is very difficult to add features required to adequately support other languages.

ELF

Unlike COFF, ELF file sections are not limited in the same manner as COFF's predefined line number table and symbol table. ELF is similar to a giant container for information. The container is split into sections for specific information. These sections include but are not limited to the traditional text and data. Sections can also be defined by the operating system or can be defined by the user. Thus specific information, such as debugging information, is placed in a unique and separate section of the object file. ELF does not impose inter-section dependencies. Figure 3 depicts the ELF File Format.

Figure 3. ELF File Format



Debugging information is self contained in the ELF *.debug* and *.line* sections. The *.debug* section contains source information depicted by the DWARF language. The *.line* section contains source line information for associating source files with the machine instruction addresses in the executable. The self contained debug information, allows producers to use preexisting directives to create the debugging information in the object file, unique assembler directives to handle debugging information are not required.

DWARF Debugging Language

This section presents an introduction to the concepts and information available in DWARF. This does not exhaustively describe DWARF but is intended to give a broad understanding of the format. For further information the reader should refer to the DWARF specification[DWA91].

Language Structure

DWARF as a language does not favor a specific programming language, rather it is media for communicating accurate source information between generators and consumers. DWARF is an open ended language allowing for addition of new information as new languages or new debugger capabilities are introduced.

Source language information is passed to the consumer through a low level representation known as a Debugging Information Entry (DIE). Each DIE describes a single entity in the source program (variables, subroutines, etc.). A series of DIE are used to describe an entire source program. Each DIE may parent or own one or more DIE. The parenting concept provides a means of describing complex programming structures and source file inter-relationships. All DIE owned by the same parent form a sibling chain. Each sibling references other DIE in the chain. The chain is ended by a null sibling reference. This parenting concept forms a tree structure describing the source program which consumers can easily traverse when seeking source information. The DIE describing the compilation unit is the parent of all subsequent DIE.

Debugging Information Entry

Each DIE consists of a die length followed by an identifying tag name followed by one or more attributes. DIE format is pictured in Figure 4.

Figure 4. Debugging Information Entry.

```
hex_offset:<decimal_length>TAG_tagname
            AT_attrname(value)
            AT_attrname(value)
```

The DIE length is a 4 byte unsigned integer whose value is the total number of bytes in the DIE. A DIE length less than 8 bytes represents a null or padding DIE. The padding DIE is used to satisfy alignment constraints imposed by the architecture. A 2 byte tag name identifies the class of information in the DIE. The list of reserved tag names are listed in Figure 5. The DWARF specification has assigned a value to each reserved tag name. In addition to the reserved tag names, applications may define tag names and values in the range TAG_lo_user through TAG_hi_user without conflicting with current or future system-defined tags.

Figure 5. Tag Names.

TAG_array_type	TAG_class_type
TAG_compile unit	TAG_entry_point
TAG_enumeration_type	TAG_formal_parameter
TAG_format	TAG_global_subroutine
TAG_global_variable	TAG_hi_user
TAG_imported_declaration	TAG_inline_subroutine
TAG_label	TAG_lexical_block
TAG_lo_user	TAG_local_variable
TAG_member	TAG_member function
TAG_padding	TAG_pointer_type
TAG_reference_type	TAG_set_type
TAG_source_file	TAG_string_type
TAG_structure_type	TAG_subroutine
TAG_subroutine_type	TAG_typedef
TAG_union_type	TAG_unspecified_parameters
TAG_variant	TAG_with_stmt

Attributes are value/name pairs describing the characteristics of a single source entity being defined by a DIE. Attributes are the means by which DWARF:

- provides consumers a method to find the location of program varibles,
- determines array subscript bounds,
- calculates the return address of a subroutine,
- finds the base address of the stack, *or*
- determines the name of variables.

Attributes are represented as a 2 byte name field followed by the appropriate value. A list of reserved attribute names is included in Figure 6. Similar to tags, the attribute list has a user defined range, AT_lo_user through AT_hi_user. Applications may define the use of attributes in this range without conflicting with the system-defined values.

Figure 6. Attribute Types

AT_bit_offset	AT_bit_size
AT_byte_size	AT_comp_dir
AT_deriv_list	AT_discr
AT_discr_value	AT_element_list
AT_frame_base	AT_fund_type
AT_hi_user	AT_high_pc
AT_import	AT_incomplete
AT_language	AT_lo_user
AT_location	AT_loclist
AT_low_pc	AT_member
AT_mod_fund_type	AT_mod_u_d_type
AT_name	AT_ordering
AT_producer	AT_sibling
AT_start_scope	AT_stride_size
AT_string_length	AT_stmt_list
AT_subscr_data	AT_user_def_type
AT_visibility	

Attribute values are one of the following forms:

- address - location in address space,
- reference - member of dwarf description,
- constant - uninterpreted data,
- block - arbitrary number of bytes, *or*
- string - null-terminated string.

Attributes may only assume the value form as specified in the DWARF specification. The form of the value is encoded in the attribute name. For example the AT_name attribute is always of form string. DWARF defines a set of form encodings which are appropriate for each attribute type. To maintain the flexibility and extensibility of DWARF, most encodings have a user definable range, for application specific information.

Example

It would be much to difficult to explain each attribute in detail. Thus, the following example is to give the reader the opportunity to look at DWARF language generated for a simple C language program. An effort has been made to annotate the DWARF language, pointing out details of interest. The example illustrates sibling chains and the DIE format.

Figure 7. DWARF Examples

C Language Source Code

```
main (argc, argv)
     int argc;
     char *argv[];
{
     int k=10;
     {
       int j=3;
       j=1;
       here:
       j=k;
     }
     k=4;
}
```

DWARF REPRESENTATION

```
0000:<43>TAG_compile_unit
        AT_sibling(0x14c)
        AT_name("test.c")
        AT_language(Lang_C89)
        AT_low_pc(x2035C)            (1)
        AT_high_pc(x20360)
        AT_stmt_list (0x0000)        (2)

 002b:<47>TAG_global_subroutine
        AT_sibling(0x145)
        AT_name("main")
        AT_fund_type(FT_integer)
        AT_low_pc(0x2035c)
        AT_high_pc(-x203b0)
        AT_comp_dir("/home/mine")
        AT_producer("name/version")

 005a:<38>TAG_formal_parameter
        AT_sibling(0x080)            (3)
        AT_name("argc")
        AT_fund_type(FT_integer)
        AT_location(<11>OP_basereg(FP
          OP_const(0x10)   OP_add)(4)
```

```
0080:<42>TAG_formal_parameter
        AT_sibling(xaa)
        AT_name("argv")
        AT_mod_fund_type(<4>MOD_pointer_
          MOD_pointer_to  FT_char) (5)
        AT_location(<11> OP_basereg(FP)
          OP_cont(14)   OP_add)

00aa:<24>  TAG_lexical_block
       AT_sibling(0141)
       AT_low_pc(020374)
       AT_high_pc(0x2039c)

00c2:<25>TAG_label
       AT_sibling(0xdb)
       AT_name("here")
       AT_low_pc(0x20380)

00db:<35>TAG_local_variable
      AT_sibling(0xfe)
      AT_name("k")
      AT_fundamental_type(FT_integer)
      AT_location(<11>OP_basereg(30)
         OP_const(0x8)   OP_add)

00fe:<24> TAG_lexical_block
       AT_sibling(0x13d)
       AT_low_pc(0x2037c)
       AT_high_pc(020394)

0116:<35>TAG_local_variable
       AT_sibling(0x139)
       AT_name("j")
       AT_fundamental_type(FT_integer)
       AT_location(<11 > OP_basereg(FP)
          OP_const(0x12)   OP_add)

139:<4>                                (6)

13d:<4>

141:<4>

145:<7>                                (7)

14c:                                   (8)
```

Example Annotation

(1) AT_low_pc and AT_high_pc values are the relocated addresses of the first machine instruction generated for the compilation unit and the first machine instruction past the last exectuable instruction, respectively.

(2) AT_stmt_list attribute value is an offset in the .line section referencing the first byte of information for this compilation unit. The .line section is used to correlate source-level statements and machine executable instructions. This information is useful for displaying source statements and single stepping through a source program.

(3) The AT_sibling attribute is used to order lexical blocks, subprograms, etc. in DWARF. In this instance the next sibling is at location 0x080, which is the description of variable *argv*. *Argc* and *argv* are both siblings of parent subroutine main.

(4) The AT_location attribute is used to build complex addressing expressions. The AT_location value is of type block, which is a count followed by a contiguous set of bytes. DWARF defines a set of basic building blocks by which the address of the object is determined. In the example 16 is added to the value in the Frame Pointer (FP) register. That is, the location of *argc* is stored at the address pointed to by the Frame Pointer plus 16.

(5)The AT_mod_fund_type attribute defines variables which are described by applying one or more modifiers to a fundamental type. In this case *argv* is described as a pointer to a pointer of fundamental type character.

(6) A null record, the end of sibling chain.

(7) A padding record, to align debugging information to a 4 byte boundary. Some architectures require alignment of sections.

(8) Beginning address of the next compilation unit. Many executable pieces of software are comprised of software modules. Dwarf has the concept of compilation units, to describe the modules making up the executable.

Compatibility

DWARF imposes constraints on additions to the standard and user definitions so that all versions and implementations will remain compatible. Figure 8 enumerates the compatibility requirements.

Figure 8. Compatibility Requirements.

1. New attributes must be added in such way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.
2. New attributes may be included in any DIE as long as the semantics of any new attributes do not alter the semantics of previously existing attributes
3. New tags may be created as long as the semantics of the newly created tags do not conflict with the semantics of previously existing tags.
4. New values may be created for the visibility attribute. Any new values so created would have language dependent meanings.

In addition, DWARF consumers encountering application specific information or specifications from latter versions are expected to ignore information which they are unable to interpret. Finally, as previously discussed, users defining application specific tags, attributes, or attribute values are expected to define only those values specified in the user ranges so as not to conflict with the DWARF specification. These few compatibility constraints ensure that all generators and consumers of DWARF will successfully interoperate, regardless of who and where they were developed.

DWARF's Impact

Dwarf impacts source-level debugging.

1. DWARF is a foundation for building a flexible, extendable, and effective debugging environment.
2. DWARF is a debugging language providing interoperability between generators and consumers.

The DWARF language is a solid basis for building a debugging environment. DWARF is easily expanded to include new language features and application specific features, by adding new tags, attributes and attribute value encodings. Additionally DWARF serves as a solid building block, supporting state of the art debugging techniques. As a result, there is nc need for a single vendor to support a multitude of debugging methods and languages. DWARF's flexibility and extensibility encourages reuse and commonality.

DWARF is a debugging language providing interoperability between generators and consumers. DWARF is a self contained language using generic containers and architecture provided directives and instructions to convey information between generators and consumers. As a stand alone language, DWARF is not dependent on a single operating system, or object file format eliminating the typical generators and consumers contracts. As a result, generators and consumers do not need to be built and maintained at a single site, allowing software developers to specialize and modularize software packages, potentially decreasing costs while increasing software quality.

The PLSiG has put much time and effort into developing the DWARF language guaranteeing that the features allowed by the DWARF language are implemented in the most efficient and compatible way. It is easy to adapt and conform to DWARF making it a debugging language of choice. Thus it can be easily incorporated and used in most software environments from the microprocessor to the supercomputers.

The UI PLSIG

The PLSIG is a group of companies and individuals who are interested in enhancing programming languages tools. Currently the group is addressing debugging issues. In particular the group is expanding DWARF, including recommendations and techinques for greater efficiency, additional functionality and new programming languages. The group meets every 6-8 weeks at locations around the country, with each meeting typically sponsored by one of the members. If you are interested in more information about the PLSIG or DWARF contact Dan Oldman, chairman, email oldman@dg-rtp.dg.com.

References

[SVABI90] *System V Application Binary Interface* Unix Software Operation AT&T 1990.

[DWA91] *DWARF Debugging Information Format*, UNIX@ International Programming Languages Special Interest Group 1991.

[OCS90] *Object Compatibility Standard*, 88Open Consortium Ltd. Release 1.1, April 1990

Watson: A Graphical User Interface Environment for Debugger Development

Randy Murrish, Cray Computer Corporation

Abstract

In today's graphically oriented user interface (GUI) environment, the need for high-quality, user-friendly interfaces to debuggers is almost as important as the underlying capabilities of the debugger program itself. To support even the simplest of user interfaces, developers must become experts in GUI design and development. In response to this problem, Cray Computer Corporation has developed the Watson[1] GUI environment for application development.

Initially developed for use with our enhanced debugging environment, Watson has evolved into a general purpose GUI development system which allows novice window-system developers access to complex GUI capabilities. Watson provides a consistent, reliable, and tested object-oriented interface to a number of GUI environments including Athena Widgets, OSF/Motif[2], and OPEN LOOK[3]. This interface allows the Cray Computer Corporation debugger (bdb) to run in any one of these three environments, without change to the application. Watson provides for the changing interface to the underlying GUI selected, while maintaining a consistent interface for the application developer.

[1] Watson and bdb are trademarks of Cray Computer Corporation.

[2] OSF/Motif is a trademark of the Open Software Foundation, Inc.

[3] OPEN LOOK is a registered trademark of USL in the United States and other countries.

Prototyping debugger user interfaces with Watson is expedited with an interface to the Tool Command Language (Tcl) developed by Prof. John Ousterhout of the University of California at Berkeley. This interface allows quick and easy prototype development with the additional benefit of reusing most of the prototype code in the developed product. All of the user interface in bdb is controlled through a set of Tcl procedures which are interpreted by Tcl into calls to Watson. This setup allows easy customization of the interface by the developer and the end user.

Overview

Watson is a set of libraries which supports an object-oriented approach to user interface development. The object-oriented flavor of Watson closely matches that which is found in the X Window System Toolkit. Watson uses commercial toolkits to provide various look-and-feel standards such as OSF/Motif and OPEN LOOK, along with public domain products such as Athena Widgets. The application programmer is presented with a consistent programming interface which supports the major look-and-feel flavors.

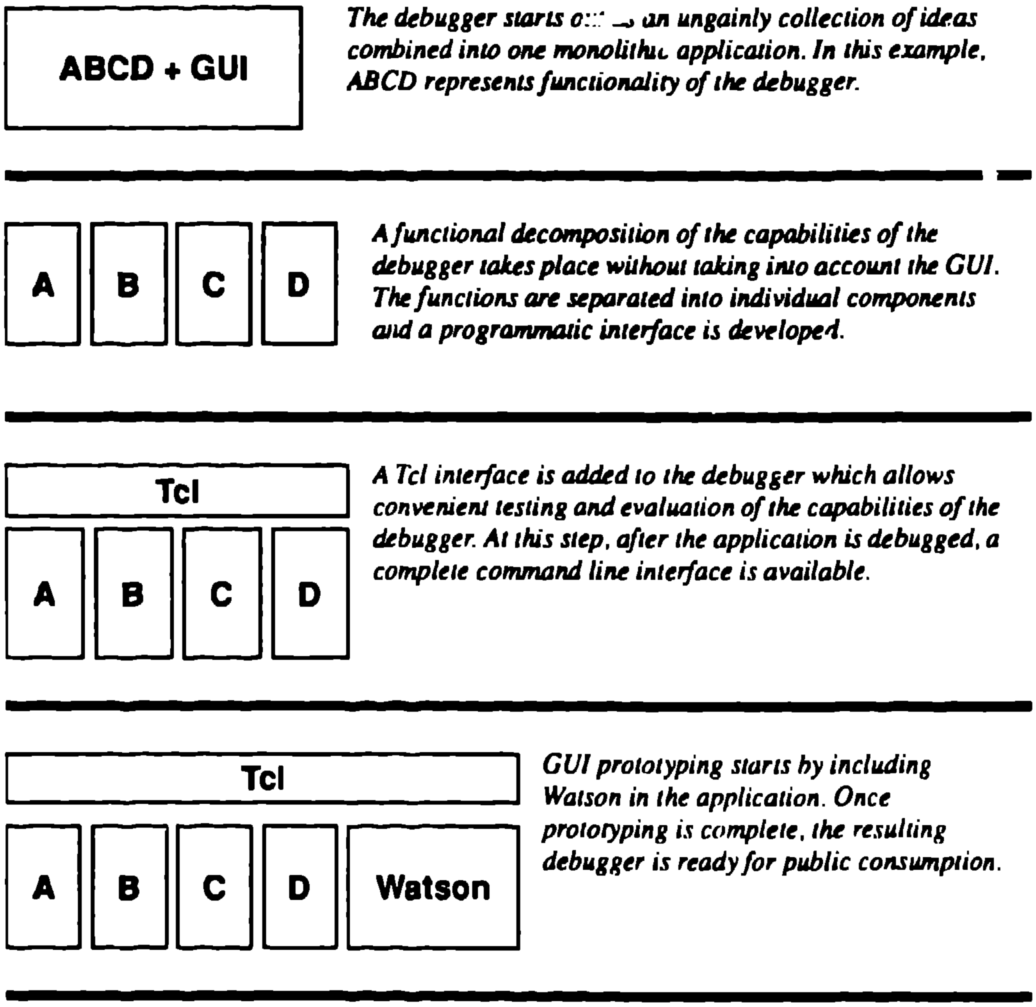
Basically, Watson is a standard set of user interface objects which, when combined in some logical, coherent manner, provides a graphical user interface to the end user. In general, Watson objects may be combined in any number of ways to create a unique, tailored interface for any application. This paper looks at the development of bdb, the Cray Computer Corporation enhanced debugger, with specific emphasis on the user interface.

Developing with Watson

The first, and most important step, in developing bdb with Watson was to completely separate the GUI from the debugger. This separation allowed easy testing of individual components, and will allow for easy distribution of the debugger across different platforms and architectures.

In addition to easy GUI development, bdb (through Watson) supports three different look-and-feel flavors (OSF/Motif, OPEN LOOK, and Athena Widgets) without change to the bdb user interface code. This advantage allows the debugger developer to concentrate on developing a consistent user interface without regard to the low-level details of various toolkits.

Figure 1 bdb Development Steps



The debugger starts out as an ungainly collection of ideas combined into one monolithic application. In this example, ABCD represents functionality of the debugger.

A functional decomposition of the capabilities of the debugger takes place without taking into account the GUI. The functions are separated into individual components and a programmatic interface is developed.

A Tcl interface is added to the debugger which allows convenient testing and evaluation of the capabilities of the debugger. At this step, after the application is debugged, a complete command line interface is available.

GUI prototyping starts by including Watson in the application. Once prototyping is complete, the resulting debugger is ready for public consumption.

Separate the Debugger from the User Interface

The first step involved in developing the user interface is to completely separate the user interface from the actual debugger. The developer concentrates on the capabilities of the debugger and designs the application as a set of components that, when combined together, provides all the capabilities required of the debugger. In several ways, this first step is similar to a functional decomposition of the requirements for the application.

During development, bdb used a functional decomposition to build the core capabilities of the debugger. This core capability was contained in a library that forms the basis of all debugger products at Cray Computer. The library has programmatic interfaces which allow complete access to the debugging environment such as:

- opening and closing a running process
- starting new processes under the environment
- setting and deleting breakpoints

Build the Tcl Interface to the Debugger

After developing the actual programmatic interface to the debugger, the next step is to build the application binding to Tcl. Although this step is not required, the prototyping and testing advantages of Tcl make these bindings very productive. The bindings are simply C routines which export the individual components of the application to the Tcl environment.

At this point, Tcl becomes an excellent tool for testing the various components of the debugger. By developing special internal interfaces to Tcl, the developer has a unique capability to test and debug the debugger or application. Tcl provides an interactive interface which allows complete control over parameters used during the debugging process.

Prototype and Build the Graphical User Interface

At this point, the debugger developer can begin to use the capabilities of Watson to prototype a GUI to complement the line mode interface provided by Tcl. The interactive nature of Tcl allows for fast turnaround times between changes in the user interface, allowing the developer to incrementally build the interface one component at a time and test the individual components at each step of the process.

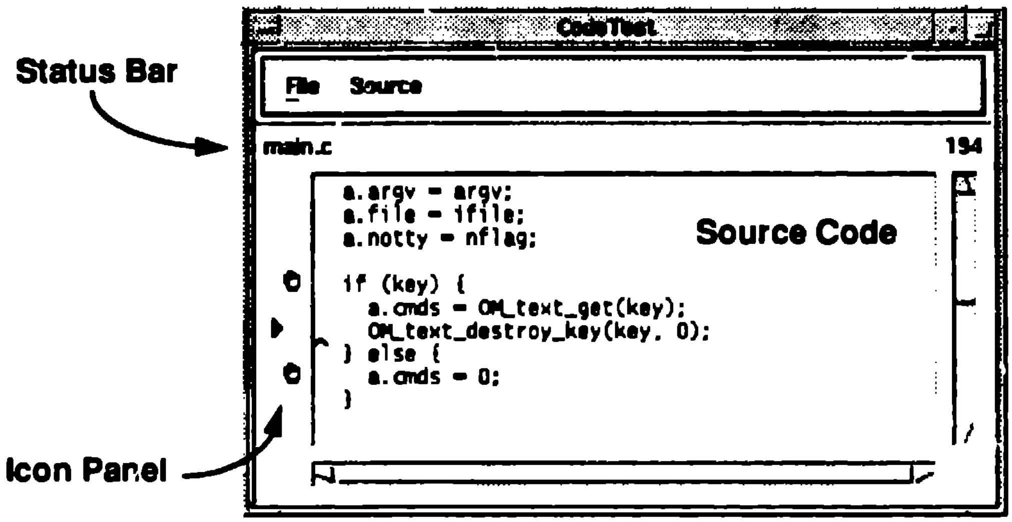
Watson and bdb

Several special capabilities were added to Watson to support the unique requirements of the bdb debugger. The most notable addition was a code display class which displays program source code with an optional status bar and icon panel. The icons are currently used to indicate breakpoints and the current program line. A status bar at the top of the display shows the current program name along with the current line number. An example of the code display class may be found in Figure 2.

The code display class can display up to eight columns of icons in the icon panel. The icons may be defined by the application programmer or by the user, and are resized by Watson based on the font selected by the user.

The code display class is not limited to use in debuggers; it was designed to be flexible and has several other uses in other application environments.

Figure 2 Sample Source Code Display (OSF/Motif)



```
a.argv = argv;
a.file = ifile;
a.notty = nflag;

if (key) {
  a.cmds = OM_text_get(key);
  OM_text_destroy_key(key, 0);
} else {
  a.cmds = 0;
}
```

The displays for the OPEN LOOK and Athena Widgets versions are basically the same as the OSF/Motif version shown in Figure 2.

Future versions of the source code display class will allow the user to add the display of individual line numbers in the Icon Panel to the left of the source code. Several users have indicated that, although this option will require more screen real estate, the benefit of having the line numbers visible will enhance readability. This feature is being developed as an option to the source code display and individual users may configure the class to toggle this capability on or off at any time.

Other enhancements may include making the icon panel sensitive to user mouse clicks, allowing the user to set or clear a breakpoint by selecting the breakpoint icon on a particular line.

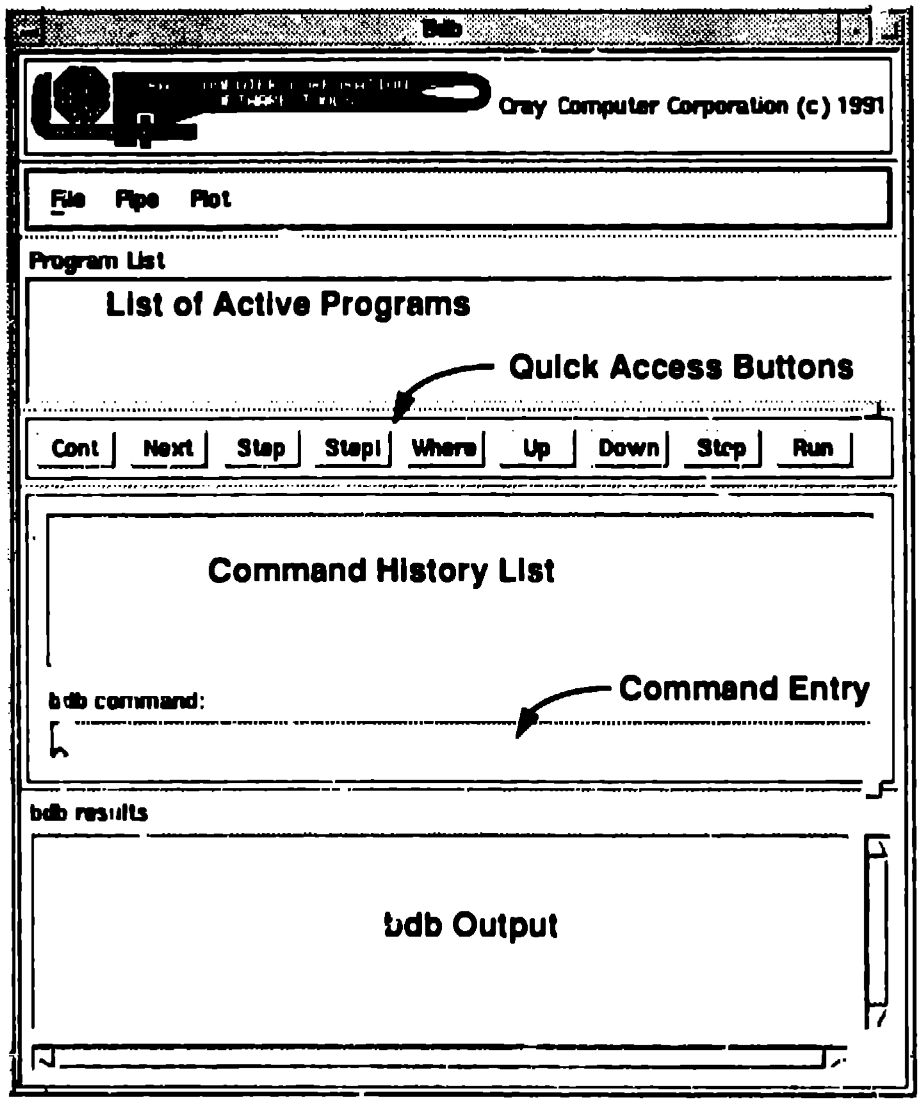
bdb User Interface

The code display class is just one unique part of the complete bdb GUI. bdb has been separated into three different windows.

bdb Main Display

The main display (shown in Figure 3) is the first window to greet users when they invoke the bdb command. This main window contains a list of the current programs being debugged by bdb, a quick access button area, a command entry and command history area, and a joint bdb and program output area.

Figure 3 bdb Main Display (OSF/Motif)

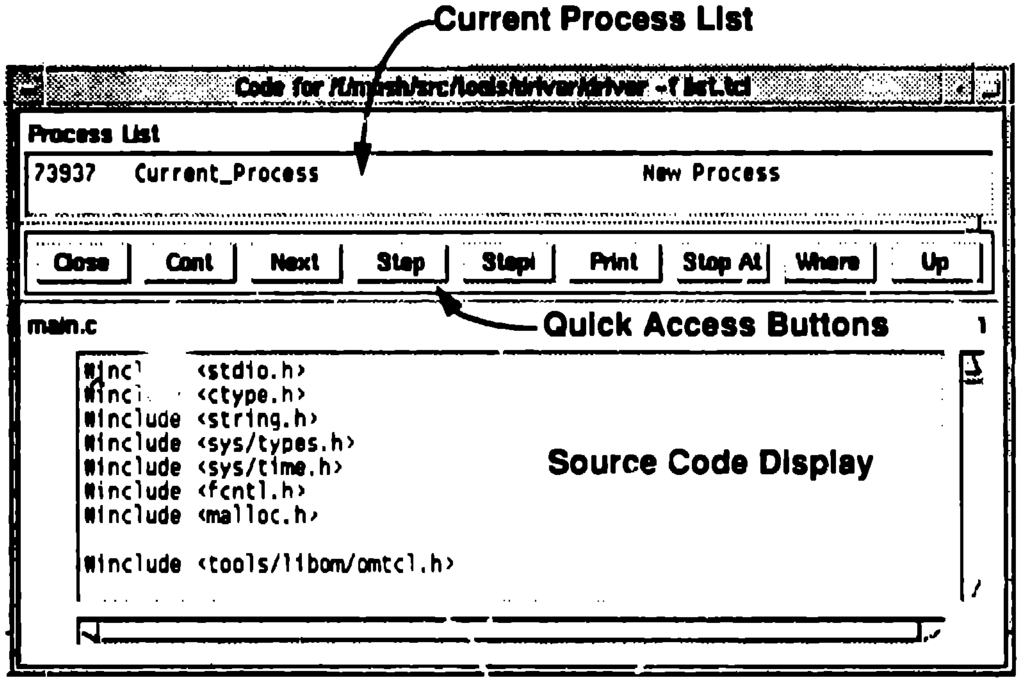


bdb Code Display

bdb creates one code display window for every process that is being debugged. An example of the separate code display is shown in Figure 4. The code display consists of a set of quick access buttons which acts only upon the program displayed in the code window, a list of individual processes connected to the program, and the now familiar code display object which displays all source code for the program

Users may select which process is active by selecting one of the entries in the current process list. The list also contains the process id, the current state, and an indication of which process is currently being displayed in the code window.

Figure 4 bdb Code Display (OSF/Motif)



```
73937   Current_Process                      New Process
```

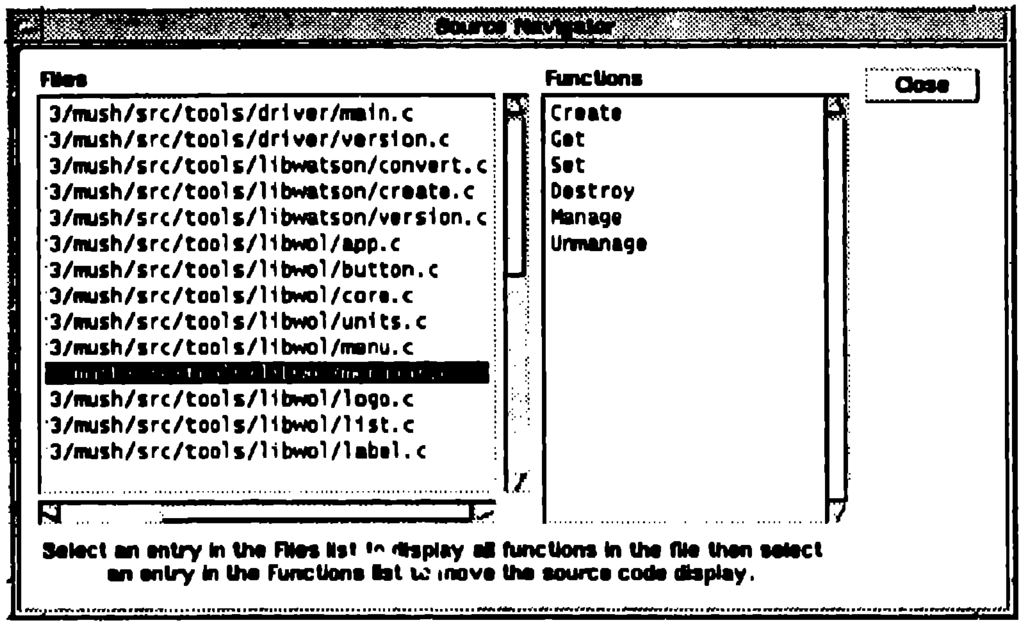
```
#inc'     <stdio.h>
#inci     <ctype.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>
#include <fcntl.h>
#include <malloc.h>

#include <tools/libom/omtcl.h>
```

bdb Auxiliary Displays

The third type of bdb display is a collection of auxiliary windows which supports the main and code displays. Currently, bdb has a complete on-line help facility that displays man page formatted documentation in a scrollable text window. Future enhancements to the Watson standard will allow full hypertext-like help capabilities to be built into every Watson based application.

In addition to the help display, bdb contains a unique Source Code Navigator display that allows the programmer to browse through all files associated with the program being debugged. The Navigator (shown in Figure 5) is simply two lists, the left containing the current source files from the program, and the right containing the functions found in one of the selected source files.

Figure 5 bdb Source Code Navigator Display (OSF/Motif)



```
3/mush/src/tools/driver/main.c
3/mush/src/tools/driver/version.c
3/mush/src/tools/libwatson/convert.c
3/mush/src/tools/libwatson/create.c
3/mush/src/tools/libwatson/version.c
3/mush/src/tools/libwol/app.c
3/mush/src/tools/libwol/button.c
3/mush/src/tools/libwol/core.c
3/mush/src/tools/libwol/units.c
3/mush/src/tools/libwol/menu.c
[illegible]
3/mush/src/tools/libwol/logo.c
3/mush/src/tools/libwol/list.c
3/mush/src/tools/libwol/label.c
```

```
Create
Get
Set
Destroy
Manage
Unmanage
```

Select an entry in the Files list to display all functions in the file then select an entry in the Functions list to move the source code display.

When a source file is selected in the left list, the right list is changed to show all of the functions in that source file. When a function is selected in the right list, the current code display window is changed to display the source code of that function. The Navigator significantly decreases the amount of time a programmer takes to find a function in a particular file.

Watson and Other Applications

Although Watson began as the user interface product for our debugger, it has grown to encompass the GUI needs of almost all applications developed by Cray Computer. Watson supports the needs of several classe , of applications from a Visual System Monitor that monitors various parameters and values of running processes, to a Visual System Administrator that provides an easy-to-use, graphical interface to all types of system administration tasks.

The benefit of using one product for all GUI needs is obvious to the user, all applications developed with Watson will have a consistent look-and-feel that reduces learning time, increases productivity, and allows easy end user customization of the GUI.

Acknowledgments

The author wishes to acknowledge the help of a number of individuals at Cray Computer Corporation who participated in defining the various capabilities of Watson and participated in testing activities: Scott Bolte, Ben Young, Darragh Nagle, and Tom Engel (the whip-wielding manager).

References

Information about Tool Command Language, along with the latest source code, may be obtained from John Ousterhout, University of California at Berkeley. A mailing list exists which is devoted to Tcl questions. To join, mail a request to *tcl-request@sprite.berkeley.edu* and ask to be included on the distribution.

Author Information

The author can be contacted by mail at Cray Computer Corporation, 1110 Bayfield Drive, Colorado Springs, CO, 80906, or by e-mail at *mush@craycos.com*.

Watson: A Graphical User Interface Environment for Debugger Development

Randy Murrish

Cray Computer Corporation

Graphical User Interface Design Goals

- We wanted to rapidly build displays that are:
  - Concise
  - Unambiguous
  - Visual
  - Hierarchical
- We also wanted to design displays that are:
  - Easy to learn using built-in help facilities
  - Easy to use
  - Easy to customize
  - Consistent across applications
  - Easy to develop and maintain by the developer
  - Easy to port to new platforms

Watson Overview

- Watson[1] is a Graphical User Interface (GUI) environment for application development.
- Object-oriented nature of Watson provides simple, standard building blocks for the developer
  - similar programming paradigm as found in the X Windows Toolkit
- Transparent support for distributed applications without changing the application
- Tool Command Language interface is available through the Object Manager support library
  - Tcl is an interpretive command language that may be imbedded in applications
  - supports rapid prototyping of new user interface techniques
  - easy for the user to customize the user interface to meet their unique needs
  - new capabilities may be added by the end user
- Built using standard products and interfaces
  - based on X11R4/R5
  - OSF/Motif[2] Toolkit v1.1
  - OPEN LOOK[3] Intrinsics Toolkit v4i
  - Athena Widget

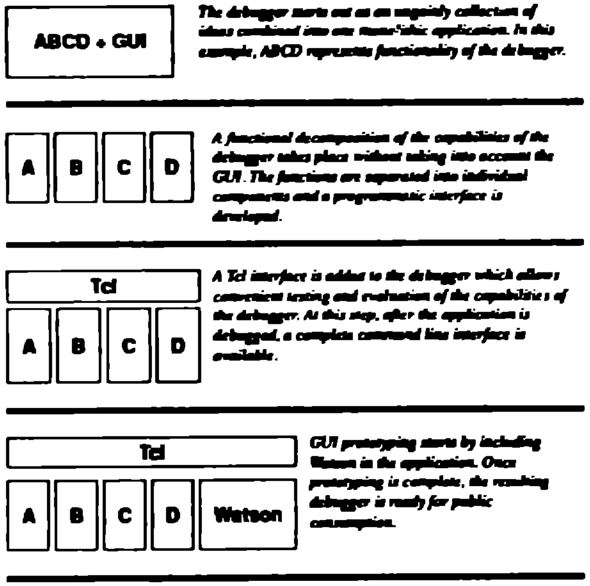
[1] Watson and bdb are trademarks of Cray Computer Corporation

[2] OSF/Motif is a trademark of the Open Software Foundation, Inc

[3] OPEN LOOK is a registered trademark of USL

Building a Debugger with Watson

- bdb progressed through several stages during the development phase:
  - bdb core capabilities identified by a functional decomposition of the requirements
  - development of the capabilities
  - Tcl interface added
  - GUI added



The debugger starts out as an ungainly collection of ideas combined into one monolithic application. In this example, ABCD represents functionality of the debugger.

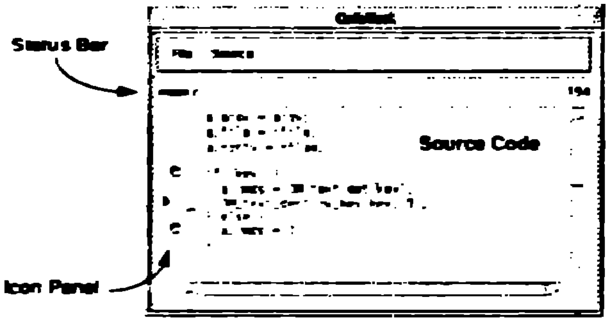
A functional decomposition of the capabilities of the debugger takes place without taking into account the GUI. The functions are separated into individual components and a programmatic interface is developed.

A Tcl interface is added to the debugger which allows convenient testing and evaluation of the capabilities of the debugger. At this step, after the application is debugged, a complete command line interface is available.

GUI prototyping starts by including Watson in the application. Once prototyping is complete, the resulting debugger is ready for public consumption.

Watson and bdb

- Special capabilities were added to Watson to support the unique requirements of bdb
- Source code display class (OSF/Motif version):



- Future enhancements to the source code display include:
  - allow the user to toggle the display of line numbers in the icon panel
  - make the icon panel sensitive to user mouse clicks allowing the user to set or clear a breakpoint by selecting a breakpoint icon

bdb User Interface

- Comprised of three different types of displays
  - Main Display

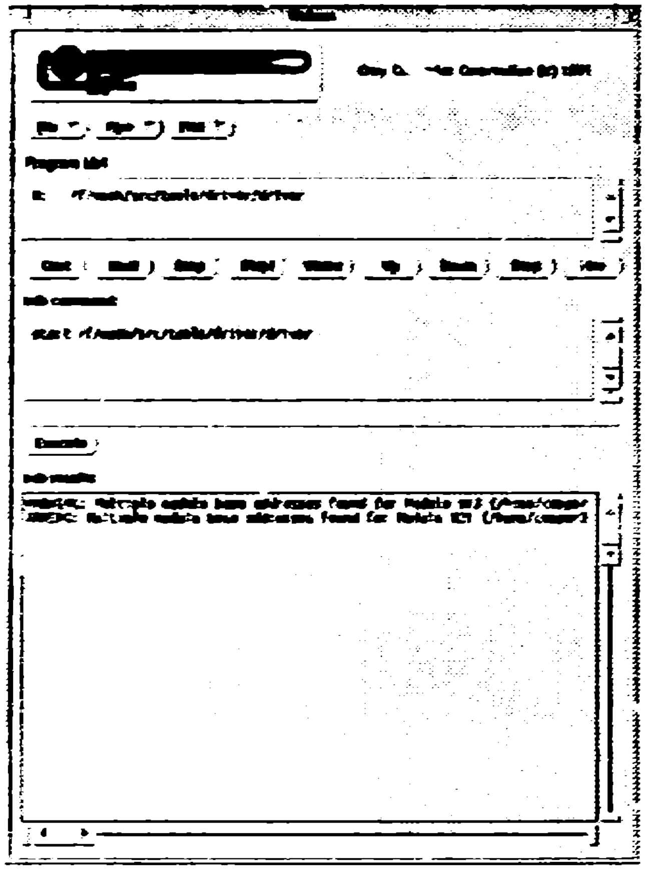
    A single window which controls all programs currently being debugged by this instance of bdb.
  - Code Display

    One code display is created for every program currently being debugged by bdb. This contains the source code display class and a list of all current processes associated with the program.
  - Auxiliary Displays

    A complete, menu-oriented help display that is available at any time.

    The Source Navigator Display which allows the user to quickly navigate through all available source code in a program.

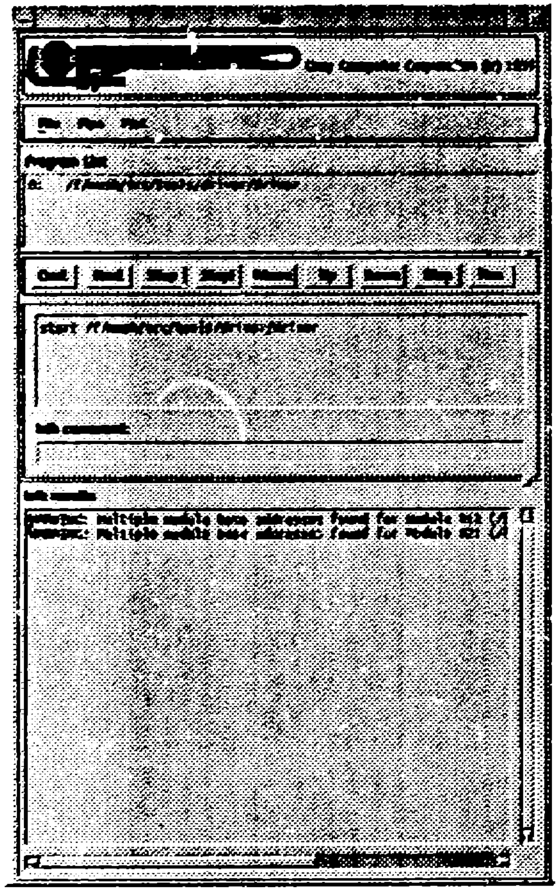
bdb Main Display (OPEN LOOK)



```
start //[illegible]
```

```
[illegible]: Multiple module base addresses found for Module [illegible]
[illegible]: Multiple module base addresses found for Module [illegible]
```

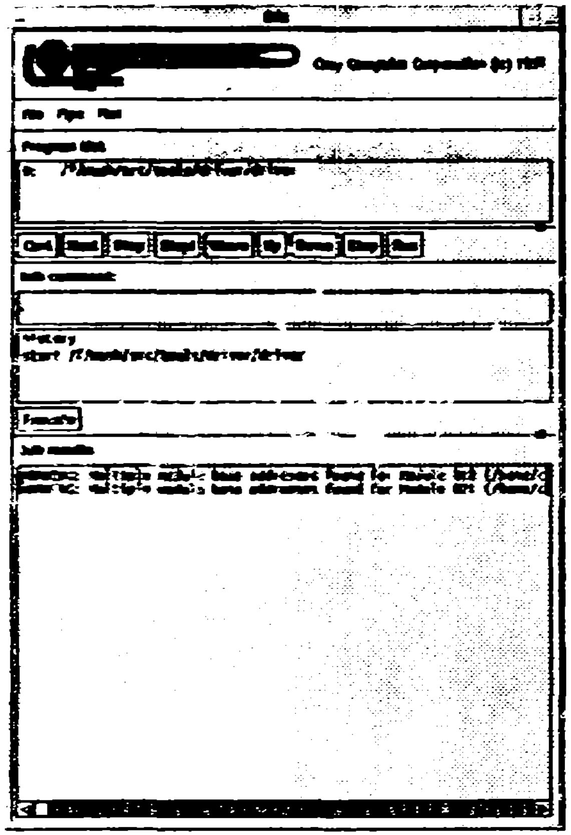
bdb Main Display (OSF/Motif)



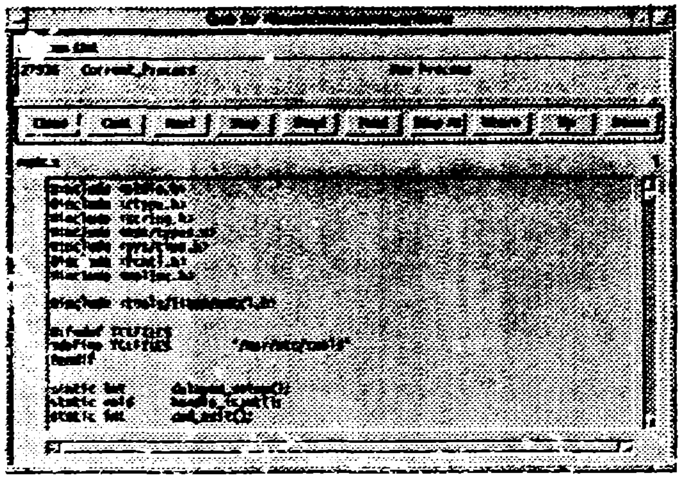
```
start //[illegible]
```

```
[illegible]: Multiple module base addresses found for module [illegible]
[illegible]: Multiple module base addresses found for module [illegible]
```

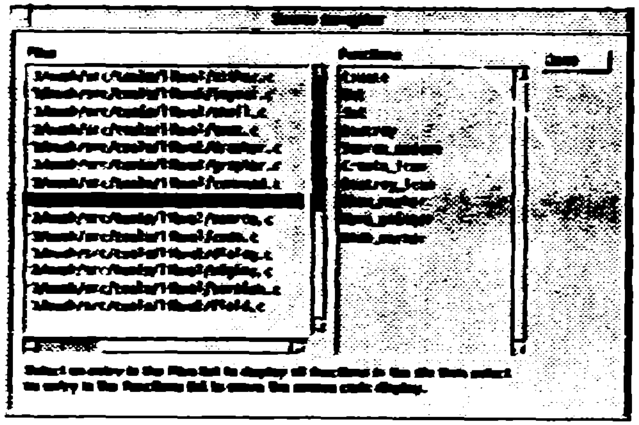
bdb Main Display (Athena)



bdb Code Display (OSF/Motif)



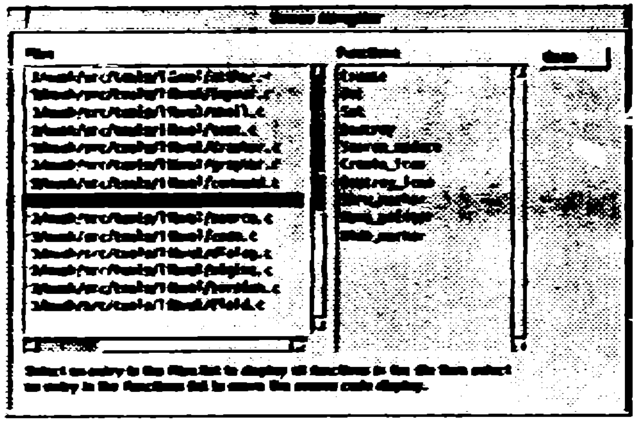
bdb Source Code Navigator (OSF/Motif)



Select an entry in the Files list to display all functions in the file then select an entry in the Functions list to move the source code display.

CRAY COMPUTER CORPORATION
Software Tools

bdb Source Code Navigator (OSF/Motif)



Select an entry in the Files list to display all functions in the file then select an entry in the functions list to move the source code display.

CRAY COMPUTER CORPORATION
Software Tools

Debugging Optimized Code Without Surprises (Extended Abstract)

Max Copperman
Charles E. McDowell

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Optimizing compilers produce code that impedes source-level debugging. A source-level debugger of optimized programs should be able to warn the user when optimization has caused the value of a variable at a breakpoint to be misleading. We describe the information an optimizing compiler must make available to a debugger, and how the debugger can use the information, to determine when optimization has caused the value of a variable to differ from the value that would be predicted by a close reading of the source code.

keywords: debugging, compilers, optimization

1 Introduction

A major cost in the construction of production-quality software is debugging. In the past decade, interactive source-level debuggers have become commonly available, and such tools increase the ease of debugging, providing a tremendous increase in the speed of locating bugs. Most of the production quality interactive source-level debuggers function as expected only on unoptimized code. When such a debugger is used to debug optimized code, it may mislead the user, causing the time spent debugging to be increased. Since optimization is desirable (sometimes necessary) for production software, interactive source-level debuggers that can be used on optimized code without misleading the user are needed.

A common misconception is that a program's behavior will not change due to optimization unless the optimizer is incorrect. This is the case if the program contains no errors and no dependencies on evaluation order. However, optimization can uncover program errors that are benign in the unoptimized case. Examples are given in Section 1.1. It is not surprising, therefore, that sometimes when a program is recompiled without optimization in order to use a source-level debugger, the bug goes away.

Previous work in this area has tended to target specific optimizations, largely local optimizations (within a basic block). This paper describes an approach that applies equally to local and global optimizations, and the debugger algorithms are independent of the optimizations that have been performed. The compiler modifications to provide the input to the debugger algorithms are not identical for all optimizations.

1.1 Why Debug Optimized Code?

Why debug optimized code? Why not simply turn off optimizations when debugging? It would then be unnecessary to solve problems related to source-level debugging of optimized code.

Disabling Optimization May Be Undesirable

In a production software engineering environment, it may be expensive to disable optimization. It may require two compilations of each compilation unit and storage of two copies of each compiled object. Furthermore, it may be impossible to avoid optimization. The compiler of choice may not allow optimization to be disabled. There is at least one highly optimizing compiler [Pic90] that, when compiling with optimizations turned off, still performs live/dead analysis, constant propagation, copy propagation, and global register allocation, any of which can confuse a source-level debugger. In principle it may be possible to get a different compiler, but as a practical matter, it may be impossible or undesirable. In addition, optimization of functions to which the user does not have source code (such as library functions) can cause debuggers to give misleading information. For example, if a library function's stack frame has been optimized away, the debugger may show an inaccurate stack trace.

Changes in Program Behavior

The most compelling reason for this research is that a program compiled with optimizations enabled may behave differently from the same program compiled with optimizations disabled. Optimization can change the program behavior for one of the following reasons.

- Loose semantics: A language may contain constructs whose semantics allow multiple correct translations with distinct behaviors. Most common general purpose programming languages do contain such constructs. The most commonly known area of "loose semantics" is evaluation order, but there are others. A correct optimized translation of a program containing code with loose semantics may have different behavior from a correct unoptimized translation of that program.

```
int i;
char a, b[10], c;

void overwrite_c() {
    a = getchar();
    c = a;
    for (i=0; i<=10; i++) {
        b[i] = '\0';
        }
    c = a;
    if (c == '\0') {

          program misbehaves

        }
    }
```

Figure 1.1: Optimization Changes Program Behavior: Example 2

- Buggy programs: A correct optimized translation of a program containing a bug may have different behavior from a correct unoptimized translation of that program. This is an important and commonly overlooked case, two examples of which are given below. We are concerned mainly with programs that have bugs — otherwise they would not undergo debugging. This can cause frustration: the programmer recompiles without optimization in order to use a source-level debugger, and the bug "goes away" (the behavior that is evidence of the bug changes).
- Buggy compiler: An optimized translation of a program may have different behavior from an unoptimized translation of that program if the translator contains errors. If a buggy compiler causes a particular (part of a) program to act differently optimized from unoptimized, getting a corrected compiler means getting a compiler whose bugs are not exhibited by translating that (part of that) program. It is often easier for the programmer to find the code that causes the compiler bug to show up and replace it with semantically equivalent code on which the compiler functions correctly. The point is that the programmer still must debug the (incorrectly) optimized code.[1]

It is common for even experienced software engineers to be surprised at the fact that a program can behave one way when optimized and a different way when unoptimized *when it has been compiled with a correct compiler*. Here are two simple examples of programs with bugs and how optimization can affect them

In the code in figure 1.1, the bug is benign when the program is unoptimized and malignant when unnecessary stores are eliminated. The bug is to write past the end of an array, overwriting the character c. In the absence of optimization c is subsequently overwritten with the correct value and the bug goes unnoticed. In the presence of optimization, the bug affects the behavior of the program: the optimizer eliminates the second assignment into c, because it can determine that in a correct program c would already contain the to-be-assigned value

Note that this situation can occur with statements that are arbitrarily far apart in the source code, so long as the optimizer can determine that a has not been modified between the first and second assignment into c

[1] Even if the choice is to get a fixed compiler, the programmer typically has to debug the optimized code enough to report the problem to the compiler vendor

```
int i;
char b[10], c;

void walk_on_c() {
    c = getchar();
    for (i=0; i<=10; i++) {
        b[i] = '\0';
        }
    if (c == '\0') {

           program misbehaves

        }
    }
```

Figure 1.2: Optimization Changes Program Behavior: Example 1

In the code in figure 1.2, the bug shows up when the program is not optimized. It has no visible effect when data fetches are optimized by aligning data structures on 4 byte boundaries.[2]

If each data object is aligned to a four byte boundary, there will be two bytes of padding between the end of array b and character c, and the bug (writing one byte past the end of b) has no effect on program behavior. If data objects are not aligned, there will be no padding between b and c; c will be overwritten.

We have seen that optimization can change the behavior of a program. It is therefore necessary, upon occasion, to either debug optimized code or never optimize the code. It is not always possible to debug unoptimized code and have it run correctly when recompiled with optimizations enabled, even when the compiler is correct.

1.2 The Data Value Problem

A source-level debugger has the capability of setting a breakpoint in a program at the executable code location corresponding to a source statement. When a breakpoint at some point P is reached, presumably the user wishes to examine the state of the program, often by querying the value of a variable V. Commonly available debuggers, upon receiving such a query, will display the value in V's location. Unfortunately, this value may be misleading due to optimization. The user will be misled if the value is different from the value that would be predicted by looking at the source code (and knowing the relevant context, such as within which iteration of a loop execution is suspended). In unoptimized code, at each point the value in V's location matches the value that would be predicted by the source code, so the *expected* value of V at P can be defined as the value that V would have at the corresponding point in an unoptimized version of the program. To provide *expected behavior*, a debugger must provide the expected value of V at P. Sometimes it may not be practical to provide the expected value of a variable. In this case the debugger can exhibit *truthful* behavior if it provides the current value in V's location, but in addition reports that the value of V might be different from V's expected value, and why.

General approaches to the problem have been

- to restrict the optimizations performed by the compiler to those that do not provoke the problem ([WS78], [ZJ90]),

[2] Note that program misbehavior, which is the external evidence of the bug, could occur when the program is not optimized and go away when the program is optimized by changing the sense of the conditional — that is, by adding another bug. What is important is that the behavior changes depending on optimization.

- to recompile, without optimization, during an interactive debugging session, the region of code that is to be debugged ([FM80], [ZJ90]), and
- to have the compiler provide information about the optimizations that it has performed and to have the debugger use that information to provide appropriate behavior ([WS78], [H82], [CMR88], [C90], [ZJ90]).

The larger problem we are concerned with is lowering the cost of debugging production quality software. Much if not most production quality software produced in this country is heavily optimized, and the first approach would result in compilers that would not get used; their use would degrade the quality of the software. The second approach requires a software engineering environment that provides incremental compilation. Such environments are not in general use and even should they become commonplace, the approach is unacceptable because optimization may change the behavior of the program (cf. section 1.1). We take the third approach.

If the value in a variable's storage location is suitable to be displayed to the user it is *current*. The remainder of this paper outlines how to determine whether a variable is current at a breakpoint. The fundamental idea behind our solution to the currentness determination problem is the following: if the definitions of a variable V that "actually" reach a point P are not the ones that "ought" to reach P, V is not current at P. The definitions of V that actually reach P are those that reach P in the compiled version of the program. The definitions of V that ought to reach P are those that reach P in a strictly unoptimized version of the program.[3] The set of definitions of V that reach at any point in a program (optimized or unoptimized) can be computed using existing algorithms [AU77]. If the set of definitions of V that reach P differs in the optimized and unoptimized version of the program, then V is not current. The debugger can use the sets of definitions to describe, in source-level terms, why V is not current. Unfortunately, if the two sets of definitions are equal it is still possible that V is not current. This is discussed further in Section 3.2.

In order to determine a variable's currentness:

1. The compiler must generate a set of debug records relating statements to code addresses; these debug records are ordered in two flow graphs, one representing the program before optimization and the other representing the program after optimization.
2. The flow graphs are used to compute reaching definitions, which are in turn used to create reaching sets (sets of definitions that reach a breakpoint location).
3. The reaching sets are compared to compute the currentness of variables.

Section 2 describes the data structures that must be produced by the compiler. Section 3 describes how these data structures are used to provide expected behavior most of the time and truthful behavior the rest of the time.[4] Section 4 discusses the accuracy of the results.

[3] One compilation of the program is sufficient to provide the information with which to compute both the definitions that ought to reach P and those that actually reach P.

[4] It is trivial to provide (useless) truthful behavior. Simply always give a warning that the code has been optimized. This is of course unacceptable.

2 Compiler Support

2.1 Debug Records

The compiler provides the debugger with information about every declaration and statement in the program. We call the collection of information about a statement (declaration) a debug record. A distinct debug record is produced for each modification to each program variable, so more than one debug record is produced for a statement that has side effects. For example, the following code causes 6 debug records to be produced:

`int a, b, c;`	(Produces three declaration debug records.)
`a = 0;`	(Produces one statement debug record.)
`b = c++;`	(Produces two statement debug records: one for the assignment into b, and one for the side effect on c.)

A debug record R for a statement S has the following fields:

- *Var(R)* — a variable name,
- *Sref(R)* — a source reference,
- *Cref(R)* — a code reference, and
- *Moved(R)* — a boolean (one bit).

The Var field identifies the variable updated by S. We say R defines V if Var(R) = V, that is, if V is the variable updated by S. If S does not update a variable, the Var field is null. The Sref field contains the source reference for S (file name and line number, perhaps which statement on the line, if the debugger is to handle lines with multiple statements). The Cref field contains the address of the instruction that represents S. If no instruction is generated for S, the Cref field is null, unless the debug record describes a declaration, in which case the Cref field contains the address at which storage for the declared variable is allocated. The Moved field encodes whether the code for S has been moved out of the basic block in which it originated.

2.2 Representative Instructions

The Cref for a statement is the address of the code for the statement. Because the code for a statement may be moved by optimization and may be discontiguous, one instruction is chosen as the representative of the statement The breakpoint location for a statement is the address of its representati e instruction (its Cref) [5] Given that the currentness of a variable V at a statement S is computed relative to the breakpoint location for S, the choice of a representative instruction for S has an impact on the quality of source level debugging provided [6] In unoptimized code, it is adequate to use the first instruction generated from S as its breakpoint location. However, this can be a poor choice in optimized code For each construct in a programming language, the breakpoint location should be chosen appropriately For assignment (and side effects that modify variables), the representative instruction must be the store into the variable for our method to be correct See [C'90], pages 60 62 for further details

[5] A single instruction represents a statement because (among other things) it would be inappropriate to break at every instruction generated from a statement.

[6] Choosing the statement as the granularity of debug records means a variable's currentness can be determined at statement boundaries, but not at arbitrary machine instructions If a breakpoint is reached, the currentness of a variable V can be accurately determined but if the program traps (halts at a non representative instruction, thus at a non statement boundary) accuracy is not guaranteed

2.3 Flow Graphs

The compiler also provides the debugger with two representations of the control flow of the program.

A flow graph representing the basic block structure before optimization is called the *source graph*. Each node in the source graph corresponds to a basic block and contains a sequence of (pointers to) debug records[7], one for each statement within the block in the order in which the statements appear in the source code.

A flow graph representing the basic block structure after optimization is called the *object graph*. Each node in the object graph corresponds to a basic block and contains a sequence of (pointers to) debug records that corresponds to the sequence of statements that have ended up in that block.

2.4 Reaching Definitions

The flow graphs are used to compute reaching definitions. We are interested in determining, for each statement that defines a variable V and reaches a breakpoint B in the unoptimized code, whether its corresponding object code reaches B. Both statement and breakpoint locations are represented with debug records, so the desired determination can be made by computing which debug records representing definitions of V reach the debug record representing the breakpoint B.

These reaching definitions are computed across, as well as within, basic blocks, so those records that must reach B (such as definitions occurring prior to B in the same block) can be distinguished from records that may reach B (definitions occurring on some but not all paths to B). For a breakpoint B, the set of source definitions that may reach B is computed based on the Sref field of the debug records in the source graph, and is called the *set of definitions* of V that reach B. The set of object code definitions that may reach B is computed based on the Cref field of the debug records in the object graph, and is called the *set of stores* into V that reach B.

We assume a null definition and a null store at the beginning of the program or subroutine, that is, at the start node of a connected component of a flow graph. This ensures that if a single definition or store for a variable reaches a breakpoint, it reaches along all paths to the breakpoint. This also ensures that at least one definition or store for each variable reaches the breakpoint.

In the absence of pointers and array references, reaching definitions could be computed using a standard iterative algorithm [AU77]. This would produce at most one definition of a given variable at the exit of a block. Using such an algorithm, an assignment through a pointer or array reference would kill all pending definitions. This would destroy information required by the currentness determination algorithm.

In this paper, reaching definitions are used to determine if a variable V has *definitely* received its value from *one particular* definition or *may* have received its value from *one of several* definitions. In the presence of pointers and array references, if a definition D through a pointer or array reference (call it $*P$) reaches B, $*P$ may be an alias for V, thus D may be a reaching definition of V.

If $*P$ is an alias for V, V receives its value from the computation associated with D. If $*P$ is not an alias for V in some particular execution, V receives its value from whatever definition D' would have reached if D were not present. Therefore, both D and D' must be considered to reach B. This is treated more formally in [C90] pp. 110-112. In the presence of pointers and array references, reaching definitions must be computed using a modified algorithm in which an assignment through a pointer or array reference does not kill previous definitions, thus more than one defintion of a given variable may reach any point, including the exit of a block.

[7] There is a single set of debug records that is shared between the two flow graphs, however, for all intents and purposes the nodes are treated as if they contain debug records as opposed to pointers to records.

2.5 Equivalent Definitions

An optimizing compiler may be able to determine that two definitions are equivalent and generate a single store, or it may generate multiple stores from a single definition. To accomodate this ability, we redefine the terms definition and store.

Definition 1: *A* definition of V *is any member of an equivalence class of modifications of* V *that occur in an unoptimized version of a program and can be determined by a compiler to represent the same computation.*

Definition 2: *A* store into V *is any member of an equivalence class of modifications of* V *that occur in an optimized version of a program and that have been generated from one definition.*

We extend debug records with a field *Equiv(R)* for the compiler to record the equivalence class that the definitions and stores fall into.[8] The reaching-definitions computation then computes the set of equivalence classes (definitions) that reaches a breakpoint in the source graph and the set of equivalence classes (stores) that reaches a breakpoint in the object graph.

[8] If the compiler has determined that a set of definitions represents the same computation, all of the stores generated from those definitions represent the same computation, thus the debug record, which represents both a definition and a store, needs only a single field to represent the equivalence class that the definition falls into and the equivalence class that the store falls into.

3 Currentness Determination

This section describes how to determine which state of currentness a variable is in at a breakpoint – the problem of *currentness determination*.

The debugger has available to it the flow graphs and debug records described in section 2. When a breakpoint B in a program is reached, and the user asks for the value of a variable V, two sets of reaching definitions are needed:

- the set of stores into V that reach B in the object graph, that is, the modifications to V that actually reach the point at which execution is suspended, and
- the set of definitions of V that reach the point in the source graph specified by the user (in source terms) that corresponds to B in the object graph, that is, the definitions of V that the user expects to have reached the point at which the user believes execution is suspended.

A number of variations on how to compute these sets of definitions, trading storage space and one-time computation costs for speed at the point of the (interactive) query are possible, the most straightforward being that they are computed by the debugger at the point of the query about V.

To determine a variable's currentness we compare these sets. There can be one or many definitions of a variable that reach a breakpoint and there can be one or many stores into that variable that reach that breakpoint, inducing the matrix of four cases shown in Table 3.1. In the most complex case, in which many definitions of and many stores into a variable reach a breakpoint, comparison of the reaching sets alone is not sufficient to determine a variable's currentness. The additional work that is required to make the determination is described in sections 3.2 and 3.3. Table 3.1 summarizes this additional work.

The table yields one of five possible responses:

Current The value of the variable will be the expected value.

Endangered The value of the variable will be the expected value for some execution paths and will not be the expected value for some execution paths.

Noncurrent The value of the variable will not be the expected value (or at least not derived from the expected computation).

Not Current The variable is either endangered or noncurrent. Not current is similar to endangered but indicates that there may in fact be no execution paths where the variable gets its value from the expected computation.[9]

Graph Traversal Required Comparison of the reaching sets, and an inexpensive further test, have failed to determine the currentness of the variable. It is quite likely that the variable is at least endangered, but a precise answer (one of the above four) requires the (more expensive) graph traversals described in section 3.3.

Except for the Many-Many case, this yields a simple algorithm for determining the currentness of a variable.

3.1 When a Variable is Not Current

When the debugger is asked to display a variable, it determines whether the variable is current. If the variable is current, the debugger displays its value without comment. If the variable is not current, in addition to displaying its value, the debugger uses the sets of stores and definitions that reach the breakpoint to describe the effects of optimization. The general flavor of what the debugger can do is given by the following two sample messages that might accompany the value of a variable V. Assume a breakpoint is reached at line 339.

[9] Endangered means there is at least one execution path along which the variable has its expected value. Noncurrent means there is no path along which the variable has its expected value. Not current means there may be a path along which the variable has its expected value.

	One definition, d, reaches	Many definitions reach
One store, s, reaches	Was s generated from d? Yes: current No: noncurrent	Was s generated from one of the definitions that reach? Yes: endangered No: noncurrent
Many stores reach	Was one of the stores generated from d? Yes: endangered No: noncurrent	Were any of the stores generated from any of the definitions? No: noncurrent Yes: Were the stores exactly those generated from the definitions, and did every definition generate a store? No: not current Yes: Was there any relevant code motion? No: current Yes: graph traversal required

Table 3.1: The Various Cases

> "V should have been set at line 336. However, optimization has moved the assignment to V at line 342 to near line 327. V was actually set at one of lines 336 or 327."
>
> "V should have been set at line 336. However, optimization has moved that assignment to near line 348. V was actually set at one of lines 312, 327, or 323."

The description of the effects of optimization will vary in specificity as the effects of optimization vary in complexity. The descriptions in the Many-Many case will be less specific than the descriptions in the One-One case.[10]

3.2 Multiple Stores and Multiple Definitions

Consider the case in which there are multiple definitions of V and stores into V that reach B, and the stores that reach are exactly those generated from the definitions that reach. V may be current, no current, or endangered. Figure 3.1 shows all three possibilities. It is unacceptable to be overly conservative and claim that V is not current in this case, because the stores that reach are always exactly those generated from the definitions that reach, in unoptimized code. A debugger using such an algorithm on unoptimized code would claim that any variable that has definitions on more than one path to B is endangered, when in fact no variables are endangered.

One way to determine V's currentness is to test whether for each path p to B the store that reaches B along p was generated from the definition that reaches B along p. Depending on the characteristics of programs, it may be preferable to use an approximation to V's currentness at B that sacrifices accuracy to avoid potentially expensive graph traversals. Such an approximation should be conservative — it may occasionally incorrectly tell you V is not current, but it should never tell you that V is current when V in fact is not

There is such an approximation, which, if the compiler saves the appropriate information, is simple to compute. The approximation is: *If no relevant code motion has crossed block boundaries, V is current at*

[10] Such messages can be produced because line number information is stored in the debug records and post optimization statement ordering information is present in the object graph

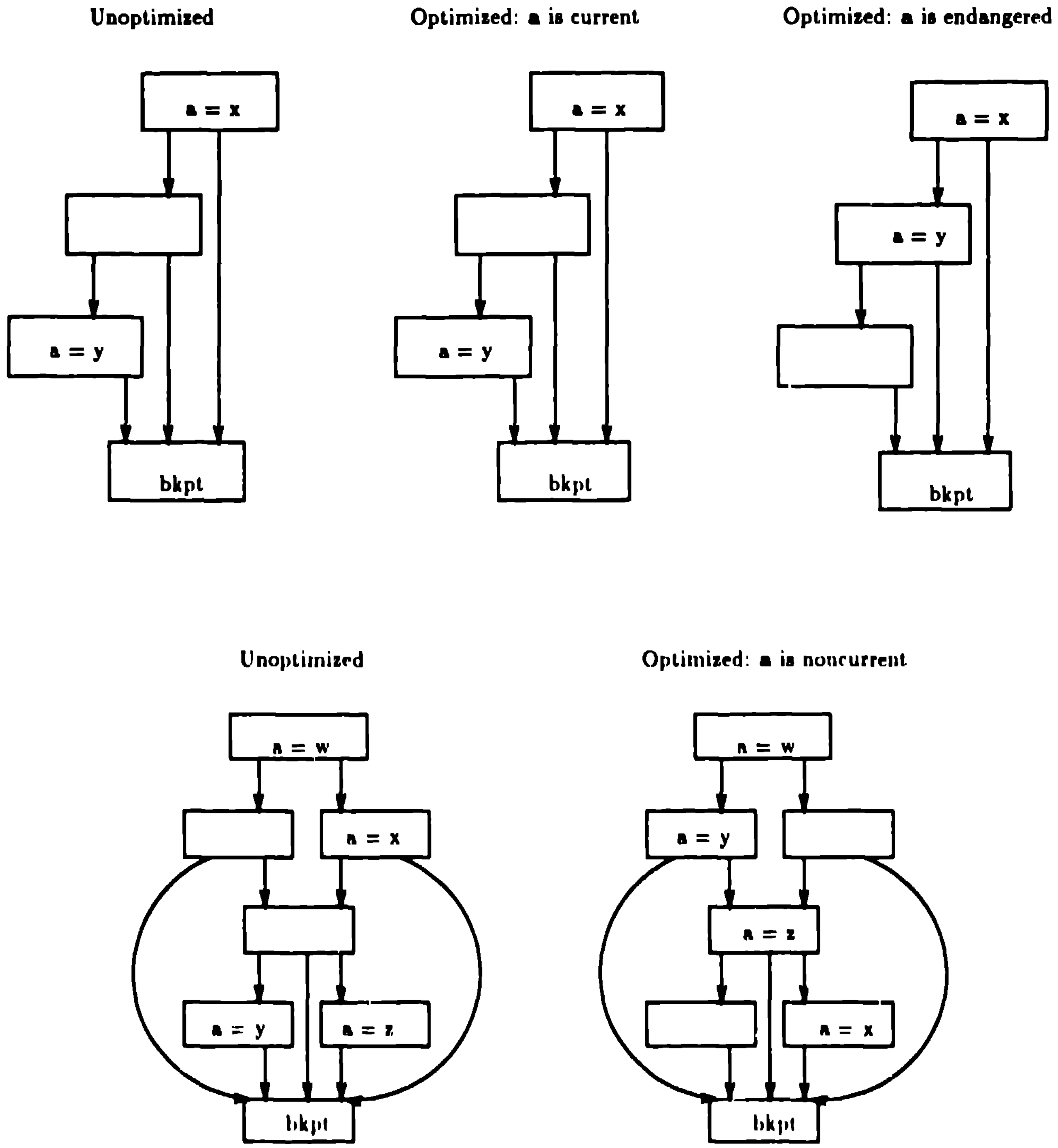


Figure 3.1: Stores that Reach bkpt are Exactly Those Generated from Definitions that Reach bkpt

B. *If such motion is found, V may be conservatively claimed to be not current at B*. Informally, relevant code motion is any motion across block boundaries of stores generated from definitions that reach *B* or movement of *B* across a block boundary (this includes code elimination as a special case). It is not known how good this approximation is. However, because no code motion occurs in the absence of optimization, this approximation works perfectly on unoptimized code. Furthermore, to get to the inaccurate (conservative) case there must be

- optimization involving relevant code motion,
- more than one definition of *V* reaching the breakpoint,
- more than one store into *V* reaching the breakpoint,
- and the stores that reach must be precisely the stores generated from the definitions that reach.

3.3 When All Else Fails

Let us examine the case in which comparing reaching sets does not give us an answer and relevant code motion has occurred. We are now assuming the conditions enumerated above.

In general, *V* is current at *B* if every path to *B* that goes through a definition of *V* also goes through the store into *V* generated from that definition, and neither the definition nor store are subsequently killed. This would be expensive to determine in general. It is somewhat cheaper in this special case (although still considerably more expensive than comparing reaching sets) because we know that the definitions and stores that reach *B* match exactly, and stores that have not moved can not endanger *V*.

Given the assumptions necessary to reach "graph traversal required" in Table 3.1, V is current at B if for all definition/store pairs *d*,*s* such that *s* was generated from *d* the following hold:

1. If *s* has been moved DOWN out of the block containing *d* then
   - (a) for all paths from *d* to B along which *d* reaches B, *s* reaches B and
   - (b) there is no path to *s* that did not go through *d*.
2. If *s* has been moved UP out of the block containing *d* then
   - (a) for all paths from *s* to B along which *s* reaches B, *d* reaches B and
   - (b) there is no path to *d* that did not go through *s*.

Notice that case 2 above is identical to case 1 with the roles of *d* and *s* reversed.

Figure 3.2 attempts to capture the restrictions pictorially on an example in which the store has moved down. Not captured in the figure is that if one of *d* or *s* is killed along any path, both must be.

Whether these restrictions hold can be computed by a pair of recursive graph traversal algorithms bolstered by an additional reaching definitions computation. Algorithm Through-top (given in Figure 3.3) can be used to test whether all paths to one block pass through another (conditions 1b and 2b above). Algorithm Through-middle (given in Figure 3.4) can be used to test whether all paths from one block to another pass through a particular middle block (conditions 1a and 2a above). Both algorithms rely on being able to determine if one block is an ancestor of another, so the transitivity of the flow graphs, ignoring back edges, must be computed.

Through-middle may uncover a block *A* such that there is a nonempty set of paths from the block Top containing a store[11] to the block Bottom containing the breakpoint that pass through *A*, but do not pass through the block Middle containing the definition[11].

[11] This assumes the store has moved up. If the store has moved down, replace "store" with "definition" and "definition" with "store".

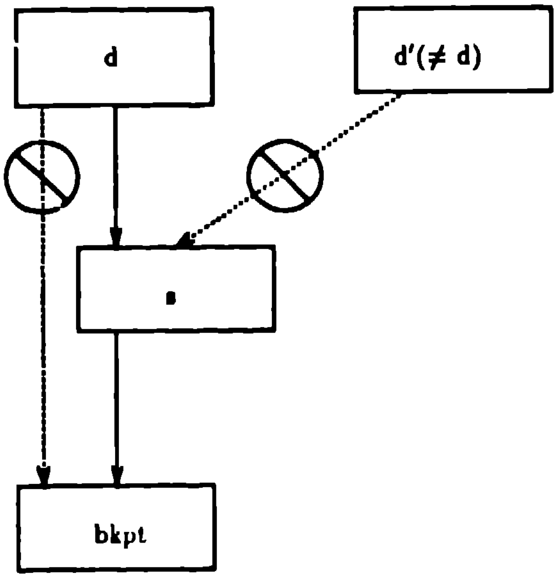


Figure 3.2: Paths if *V* is Current

```
Through-top(Top, Bottom)
   for each predecessor S of Bottom
      if S != Top
         if Top is not an ancestor of S
            return False
         else if Through-top(Top, S) = False
            return False
   return True
```

Figure 3.3: Graph Traversal Algorithm Through-top

```
Through-middle(Top, Middle, Bottom, DefOrStore)
   for each successor S of Top
      if S != Middle
         if S is an ancestor of Bottom
            if S is not an ancestor of Middle
               if not Top-killed(Top, Bottom, DefOrStore)
                   return False
            else if Through-middle(S, Middle, Bottom, DefOrStore) = False
               return False
   return True
```

Figure 3.4: Graph Traversal Algorithm Through-Middle

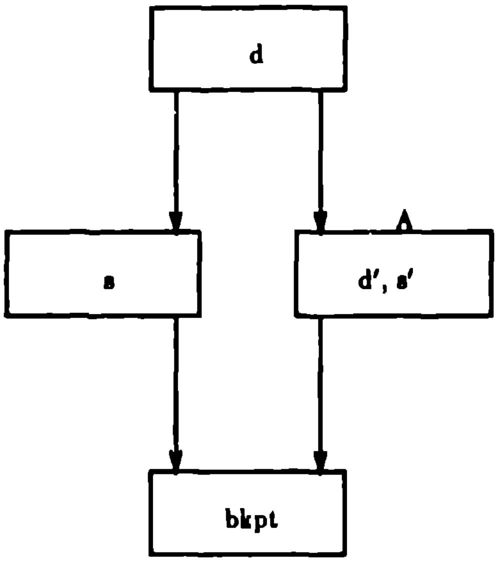


Figure 3.5: *V* is Current

If the store[11] in Top reaches the breakpoint on any of these paths, *V* is endangered. If the store[11] in Top is killed on all such paths, *V* is not endangered by the motion of this particular store[11]. Figure 3.5 is an example of such a situation in which *V* is current. If Through-middle does find such a block *A*, Top-killed() is called. Top-killed(Top,Bottom,DefOrStore) returns True if DefOrStore is killed on all paths from Top to Bottom and False otherwise. DefOrStore is a definition of V or a store into V in block Top. Top-killed can be computed using a standard reaching definitions algorithm on the graph containing Top, Bottom, and all blocks on paths between the two.

4 Summary

We have presented a solution to the problem of optimization causing a debugger to provide an unexpected and potentially misleading value when asked to display a variable. The solution works for both local and global optimizations. The algorithms for the debugger are independent of which optimizations have been performed, however, the algorithms used by the compiler to generate the necessary flow graphs are not.

For most optimizations, under most situations, our results are precise (i.e., a variable claimed to be current is current, a variable claimed to be endangered is endangered, etc.). When the results are not precise, they are conservative: a variable claimed to be endangered, noncurrent, or not current may in fact be current. A variable that is not current is never claimed to be current.

The situations in which the results may be conservative are:

- when the breakpoint has moved in such a manner that the set of definitions that reach its new location differs from the set that reach its original location, and
- when a variable is current along all feasible paths but noncurrent along some infeasible path.[11]

[11] An infeasible path is one that cannot be taken in any execution.

Optimization	Algorithm Accuracy
common subexpression elimination	Generally Precise[12]
cross-jumping	Generally Precise
instruction scheduling	Generally Precise
other code motion	Generally Precise
partial redundancy elimination	Generally Precise
loop reordering	Generally Precise
induction-variable elimination	Generally Precise
loop fusion	Generally Precise
loop unrolling	Conservative
inlining (procedure integration)	Conservative

Table 4.1: Characteristics of Representative Optimizations

For some optimizations, our results may be conservative in any situation. These optimizations are those that duplicate code where the duplicates are not in the same equivalence class (one duplicate does not represent the same computation as another, as in loop unrolling). Table 4.1 lists representative optimizations and shows how precise our results are on them.

The method to precisely determine a variable's currentness in the most difficult case, described in section 3.3, may be expensive. Section 3.2 describes an inexpensive conservative approximation to the precise result in this case.

[12] See bullets in the conclusion for exceptions.

References

[AU77] Aho, A.V. and Ullman, J.D. "Principles of Compiler Design," Addison Wesley, Menlo Park, CA, 1977.

[CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[C90] Copperman, M. "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values," *University of California, Santa Cruz technical report UCSC-CRL-90-23*, May 1990.

[EK77] Eve, J. and Kurki-Suono, R. "On computing the transitive closure of a relation," *Acta Informatica* Vol. 8, pp. 303-314, 1977.

[H82] Hennessy, J. "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982.

[FM80] P. H. Feiler, R. Medina-Mora, "An Incremental Programming Environment," *Carnegie Mellon University Computer Science Department Report*, April 1980.

[ZJ90] L. W. Zurawski, R. E. Johnson, "Debugging Optimized Code With Expected Behavior," Unpublished draft from University of Illinois at Urbana-Champaign Department of Computer Science, August 1990.

[WS78] H. S. Warren, Jr., H. P. Schlaeppi, "Design of the FDS interactive debugging system," *IBM Research Report RC7214*, (IBM Yorktown Heights), July 1978.

[Z84] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," *Research Report CSL-84-5* , Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

[Pic90] D. Pickens, MetaWare Incorporated, Santa Cruz, CA, personal communication regarding the MetaWare High C compiler, January 1990.

The Symbolic Debugging of Code Transformed for Parallel Execution

Patricia Prather Pineo
Allegheny College
Meadville, Pa. 16335

Mary Lou Soffa
University of Pittsburgh
Pittsburgh, Pa. 15260

Abstract

A technique is presented that enables the debugging of transformed and parallelized code from the point of view of the sequential code. The method frees the programmer from the necessity of viewing the substantiall altered code produced by the parallelizing transformations[1].

A critical problem that arises in the debugging of such code is that values requested by the programmer at a breakpoint placed in the transformed code may be different when reported than the values expected by the programmer. In other cases, the requested variable may have changed in type or dimension. Such variables and their values are termed *non-current*. These discrepancies between the user's view and the runtime representation are due to the action of the transformations on the code, where declaration statements can be modified and executable statements can be added, deleted, moved or replicated. The accurate tracking of non-current variables for the purpose of reporting expected values to the debugger is the subject of this work. A subset of this problem has been studied previously in the context of code transformed for optimization.

The technique for debugging transformed code is realized in a three stage system, where the code is first transformed by global renaming into single assignment code. The code can then be transformed and parallelized by any desired software package. A drawback of the global renaming is that many new names are created, most of which may not produce a useful benefit. Therefore a second stage analysis is applied to the program, after parallelization has occurred but before compilation, where new names which have not produced any benefit are reclaimed. The names not reclaimed have either been useful in exposing additional parallelism, or are required by the tracking function for the debugger. The third stage is a runtime interface that retrieves and reports the values.

This design results in a solution that is largely architecturally independent, is independent of transformations applied, correctly reports values in a high percentage of cases, and enhances the parallelizing process. The combined effects of the global renaming and name reclamation stages are to free the code of undesirable data dependencies exposing all available parallelism. For this reason the technique is termed Code Liberation.

A prototype tool designed for structured FORTRAN 77 codes has been implemented. Results showing the technique to be effective and efficient are presented

[1] P. P. Pineo and M. L. Soffa, "Debugging Parallelized Code Using Code Liberation Techniques", Sigplan Notices (Proceedings of the Workshop on Parallel and Distributed Debugging), December 1991

Intermediate Languages for Debuggers*

Benjamin B. Cha

Department of Computer Science
P. O. Box 1892
Rice University
Houston, Texas
77251-1892

Abstract

Existing source-level debuggers are heavily dependent on both the source language of the program being debugged and the architecture of the machine on which the program runs. These dependencies place a burden on a programmer who wants to extend the debugger to work on a different language or machine. Such debuggers also typically offer poor support for sophisticated features such as debugging code that has been radically transformed during compilation.

We propose a new design for debuggers that uses a structure analogous to that of modular compilers. The debugger is separated into stages corresponding to the parser, optimizer, and code generator of a compiler. The various stages of the debugger communicate using an intermediate language which can be derived by augmenting the intermediate language of the compiler with primitives to support debugging.

We expect this design will promote the ability to debug optimized code, ease in porting the debugger to different architectures, and reusability of each portion of a debugger. This design is intended to increase the cooperation between implementers of compilers and debuggers, and to allow efficient implementations of sophisticated debugging operations, while supporting the debugging of optimized programs.

1 Introduction

Source-level debuggers are an integral part of the process of writing and maintaining programs. They permit the user to observe, control, and modify the dynamic behavior of a machine language program in terms of more familiar source symbols and constructs. A debugger maps between the source text of a program, the machine code derived from the source text, and the portion of the user's commands that are expressed with source constructs. This mapping is typically accomplished by having the debugger use a symbol table, produced by the compiler, that describes how source level constructs map to the machine level. References to the source are retained as the program passes through the various translation stages of the compiler. As the compiler determines the final translation, the machine code generator emits these annotations for later use by the debugger.

While this design has been reasonably successful, it does have significant drawbacks. The resulting debuggers depend heavily on both the language being debugged and the architecture of the machine on which the program is run. These dependencies complicate the process of modifying the debugger to accommodate a different source language or target machine. Tailoring such debuggers to follow the transformations of

*This work was supported in part by the National Science Foundation under Grants CCR [illegible] and ASC [illegible] and by IBM Corporation through its research contracts with the Department of Computer Science at Rice University.

an optimizing compiler reduces the portability of the debugger. Finally, the traditional model does not accommodate program changes during debugging. These limitations are discussed in more detail below.

In an attempt to improve portability, traditional debugger designs abstract the machine dependent parts of the debugging process and isolate them in a collection of modules. To build a debugger for a new machine, one only needs to re-implement the machine-dependent modules. The same technique can be used to provide some measure of portability across source languages. When this approach to portability is used, the debugger implementer is essentially inventing an *ad hoc* intermediate level between the source language and the machine. We will see later that besides portability, other benefits can be realized if we formalize the intermediate level.

Attempting to provide portability across a family of compilers, the implementers of UNIX[1] designed a standard symbol table format that could be shared by the compilers and debuggers on the system. Unfortunately, although their design was feasible, there are ambiguities in the "standard" that result in subtle but significant differences in the symbol tables generated by different compilers. Also, while in theory this approach ensures portability across compilers, standardization introduces other problems, as detailed below.

Unfortunately, the benefits of standardized symbol tables begin to break down when the problem of optimizing compilers is considered. The notations for a family of compilers sharing a back end must be sufficiently expressive to accommodate the constructs of all of the different source languages recognized. As the compiler more radically changes a program, the debugger must become more tailored to that compiler and its target language, so that the compiler's code transformations may be decoded by the debugger. The communication between the compiler and debugger that describes the transformations, in the form of symbol table notations, must become more complicated. These symbol notations must be able to represent the composition of all of the various mappings induced by each part of the compilation process. This composition can change as different transformations are requested or inhibited by the programmer, and as parts of the compiler are modified.

Modifying the state of a running program is an important part of debugging. Unused or infrequently executed regions of code can be manually tested for bugs by forcing the flow of execution to enter those regions. This technique can be simpler than trying to create the program inputs that will exercise that region of code. Malfunctioning blocks of code can be simply skipped, and correct results can be substituted for the incorrect values that would have been generated by the skipped blocks. This ability allows the user to continue execution, perhaps finding other bugs, and is an important feature when at a point in the execution history that is difficult to reach.

If the debugger permits minor program changes during debugging, how does it accomplish this? The debugger can interpret the inserted source code, but this implementation makes the debugger more dependent on the source language. The debugger can compile the inserted source code and patch the result into the program text, but this makes the debugger more dependent on the machine on which the program is running. As the allowed forms of source level modification become more elaborate, the debugger has to become more dependent on the source and machine language and the machine state, to translate these modifications of the source level state into changes in the state of the running process.

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

2 Proposed Solution

We propose that debuggers be built similar to compilers, adopting and extending the abstract machine model provided by a compiler's intermediate language. If compiler implementers are responsible for supporting a number of languages on several different architectures, then they are strongly motivated to use a common intermediate language, and separate each compiler into a front end and a back end. A compiler front end, consisting primarily of a scanner and parser, recognizes a given source language and generates the intermediate language translation. Each back end of a compiler recognizes this intermediate language and generates instructions for a particular machine. This separation reduces the amount of work involved in building compilers, because instead of writing a separate compiler for each language-machine pair, only one front end is needed for each language and only one back end for each architecture.

By centering the debugging process on the intermediate language, it is also possible to split the debugging system into a small number of components. The debugger would consist of a front end that is capable of mapping the translation from source to intermediate language and a back end that is capable of mapping the translation from intermediate to machine language. In response to user actions, the front end generates intermediate code that performs these actions. The back end executes this intermediate code in the current context of the program being debugged. Creating a debugger for a particular language-machine pair should be easily achieved by coupling the appropriate front end and back end.

The similarity between existing compilation systems and the proposed debugging system is intentional. Compilers and debuggers need to cooperate extensively to support source-level debugging. During the translation process, a compiler must produce the symbol table necessary to relate the source program to its machine language equivalent. In our approach, the information required by the front end of the debugger, mapping between the source and intermediate representations of the program, would be provided by the front end of the compiler. Similarly, the mapping required by the debugger back end would be generated by the compiler back end. Figure 1 depicts the relationships between the compiler and debugger in our model.

The interface between a UNIX debugger and operating system can be viewed as a version of our proposed design. The extended intermediate language is actually the machine language. The extensions added to the language to support debugging are the operations supported by the system calls used by the UNIX debuggers

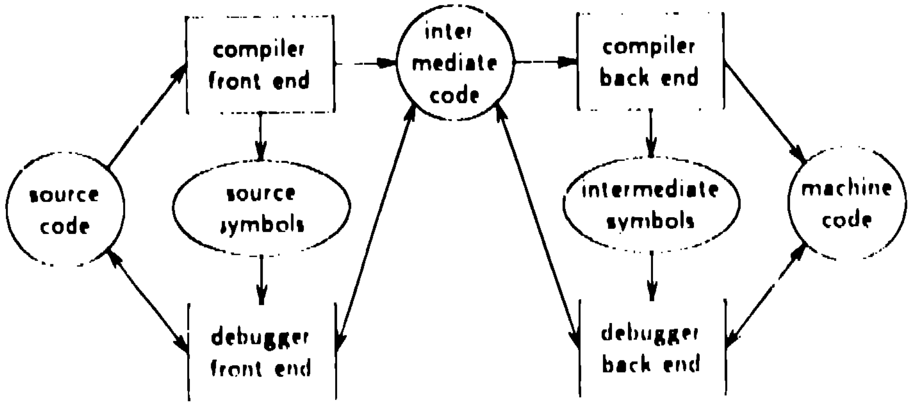


FIGURE 1. Proposed model of compiler debugger cooperation

The "back end" of the debugger is roughly the portion of the operating system that supports these calls, and the front end of the debugger we have designed is the UNIX debugger proper. The information needed by a symbolic debugger to map between source symbols and locations is found in a special part of the UNIX file containing a compiled program.

A more sophisticated compiler might have a number of middle phases that perform transformations on the intermediate language representation of the program before a final code generation phase translates the intermediate code to machine code. The debugging system for such a compiler would have corresponding interior phases, not pictured in Figure 1. Each of these *transformation* stages would compensate for the code-improving transformations performed by the corresponding middle phase of the compiler. The effects of a particular code transformation performed by a stage of the compiler would be handled by the corresponding transformation stage of the debugger. Each of these debugger stages would be both source and machine language independent, and would only need to be changed if there was a change to the corresponding stage of the compiler.

3 Benefits

The separation of the debugger into stages should simplify the task of designing, implementing, and maintaining a debugger that collaborates with an optimizing compiler. As new optimizations are added to the compiler, our design should accommodate these transformations more easily than the traditional monolithic design. The partitioning of the debugger should encourage modularity, portability, and reuse of code. Yet the debugger can still be efficient, because the abstractions in the intermediate language can be implemented in an efficient way for a particular architecture.

Changes to one part of the debugger can occur in isolation from work on the remainder of the debugger. Fast prototyping and implementation are more natural using this design, and new concepts can be tested without writing both a language-specific front end and a machine-specific back end. Such a partitioning of the debugger allows interesting research to be done on each of the independent parts of the debugger. If a team is implementing such a debugger, each part of the debugger can be written independently.

Portability is achieved by freezing the interface between the phases of the debugger, rather than by standardizing the symbol table format. This choice of interface permits the compiler and debugger to cooperate more effectively. In particular, the implementers of corresponding phases of the compiler and debugger can design the communication between those components that is most capable of representing the transformations performed.

Optimizing transformations made on the intermediate representation need only affect the transformation stages of the debugger. For a monolithic debugger, transformations such as loop unrolling, cross jumping, and procedure integration can produce a complicated mapping between source and machine code [Zel84], especially when combined with transformations performed by other stages of the compiler. By compensating for these transformations within a special transformation stage, we insulate the user interface and the machine dependent portions of a debugger from these problems.

Using this design, incremental changes to the program are relatively straightforward. Each stage of the debugger must support a mechanism for modifying its source program and its dynamic state. In all of the phases except for the last one, this modification of state is synonymous with the translation of the request to the next lower level. The stage that maps from intermediate code to machine code has two options: either translate the change into machine code and patch the executable, or arrange for the changed code to be interpreted. The other stages of the debugger are unaffected by which choice is made.

The primitives that are added to the intermediate language to support debugging provide a level of abstraction. These primitives may have a variety of implementations, differing in efficiency, ease of implementation, and intrusiveness. This abstraction allows the implementation of primitives to use special debugging support offered by a particular operating system and hardware, while retaining portability.

Since debugging operations are built from primitives in the intermediate language, the set of operations supported at the user level can be enhanced without changing the back end of the debugger. For example, given a primitive to watch a memory location, we should be able to extend the capabilities of the debugger, allowing it to continuously update a view of the contents of an abstract data structure, or dynamically monitor the performance of a running program, all by changing only the front end of the debugger.

Although the separation of the debugger should make it easier to design, implement, and maintain a debugger for optimized code, it will not necessarily be able to unravel the effects of optimization any better than a monolithic debugger. On the contrary, the separation should result in a lower level of debugging support than that provided by Navigator[Zel84]. This separation prevents various stages from cooperating with the whole compiler to preserve critical data and program text. The level of support for debugging optimized code provided by the proposed design should be comparable to that described by Hennessy[Hen82].

4 Design of an Intermediate Language for Debugging

Perhaps the most critical single element in the design of a particular compiler is the choice of the intermediate machine model. That choice is equally important to a debugger designed using our framework. Since the language chosen must also support the compilation process, intermediate languages used by existing compilers (*e.g.*, syntax trees, postfix notation, three address code) are the logical starting points for the search

Although there is considerable latitude in the selection of an intermediate language to be adapted for debugging, the choice is not arbitrary. If we use an interpreter for the intermediate language, we will prefer intermediate forms that are easily interpreted, such as three-address code. If the debugger is part of an incremental compiling system or an integrated programming environment, it should be easy to perform standard editing operations on the intermediate language, so program edits during debugging can be supported. If a program written in this language is slightly modified, (*e.g.*, an insertion or deletion of a few instructions) then the code in other parts of the program should not need to change to maintain program correctness. An example is a relative branch instruction, which only needs to be changed if there is a change in the size of the section of code being branched over. In contrast, absolute branch instructions must be updated whenever changes to the program alter the absolute addresses used in these instructions. This is usually the case for all absolute branch instructions appearing after the point of change. Also, if instructions in the intermediate language implicitly set condition codes, then program changes will require careful analyses to determine when those values are actually used

4.1 Extensions for Debugging

We want to express as many debugging functions as possible in terms of the compiler's intermediate language. While this language may be a good start, it will probably prove insufficient. Our intermediate language should be rich enough to represent most debugging operations, allowing them to be compiled. For example, we could add an instruction to the intermediate language specifically to support conditional breakpoints. However, a conditional breakpoint at the source level could be inserted by translating the evaluation of its boolean expression into intermediate code and following that by a "branch-on-false" around a "pause" instruction. For the sake of economy, we will introduce new constructs only when the compiler's language lacks needed operations, or when necessary to provide important debugging abstractions that may have many implementations of varying efficiency.

In our framework, every interaction with the program, including instrumentation requests, incremental text changes, and advancing the program counter, is expressed with a sequence of instructions in the intermediate language. We view program execution during debugging as two instruction streams, the program itself and an "immediate" command stream. Interaction with the program occurs in the form of instructions appearing on the immediate command stream.

To support interactive debugging, we add an operation that will temporarily install instructions at some point in the program. The need to insert and remove groups of instructions is common for debugging operations that monitor and control execution. In most debuggers, this operation is accomplished by overwriting the monitored point with a special trap instruction and saving the overwritten instruction for later reinstallation when the tracepoint is removed. When the trap instruction is encountered, the generated exception is fielded by the debugger, and the actions for a tracepoint are performed.

A more desirable operation is actually installing the instructions that implement the tracepoint, and subsequently removing them when the tracepoint is removed. Thus, we add two operations, called ***patch*** and ***unpatch***, to the intermediate language. When patching and unpatching in regions where the code has been nontrivially transformed, the same flavor of analysis performed by the compiler is performed for the inserted group of instructions with respect to the patch location to ensure that the patch is reasonable. In some cases, a ***patch*** operation may fail because the patch location cannot be reconstructed by the debugger, or because the patch refers to values that do not exist in the transformed version of the program. Instances where a debugger might fail to reconstruct information are described by Hennessy [Hen82].

We also need an operation to identify and name a sequence of instructions as belonging to a group that may be patched into a program to support some debugging task, and then later removed. We refer to these groups of instructions as ***fragments*** and call the operation for forming them *create*.

Many intermediate languages lack a means of accessing all of the memory locations that could be interesting during debugging. For example, it may be impossible to retrieve nonglobal values that are not in the current activation record. Examples include most intermediate languages used in compilers of C and FORTRAN. In the case, we will need to modify the intermediate language so that all values in all active procedures can be accessed.

The programmer may want to examine a single unchanging state of the process. The debugger's intermediate language must include a means of stopping a running program. After examining and perhaps modifying a single state of the process, the programmer may eventually want to proceed to a different process state

A means of resuming execution is also needed. We will call these two operations *pause* and *resume*.

In addition to starting and stopping a process, we will want other operations. These include the ability to destroy existing processes, start new processes, attach to existing processes to debug them, and detach processes after the user is done examining them using the debugger. Other examples of operations on processes are those that would support debugging in the presence of multiprocessing, using operations for naming and controlling multiple processes. Finally, a sophisticated debugger will want operations to save and restore the state of a process.

4.2 Abstractions for Efficiency

There are potential drawbacks to splitting the debugger into separate pieces. One problem is that the back end will not know the "intent" of the front end. Consequently, efficiency may suffer. For example, tracing changes to a memory location in a naive way can be very time-consuming. Existing debuggers [DEC88] have used operating system features to improve the efficiency of this operation. One scheme is to write-protect the memory page containing the monitored location. When the program attempts to write the page containing the monitored location, an exception is generated, which is fielded by the debugger. Thus the debugger only checks memory references in the vicinity of the monitored location, often making this monitoring operation much more efficient.

In our model, if the front end of the debugger specifies that a memory location should be monitored, but specifies this action in a low level way, it may be impossible for the back end of the debugger to recognize. The debugger will not realize that write-protecting the page of memory containing the monitored location might be advantageous. The front end will not be able to write-protect memory pages, because our abstraction hides the implementation of the back end. Instead, we need to express this low level operation in an abstract way.

5 Example of Stage Interaction and Fragment Insertion

The following example shows how the transformation stages of a debugger act to compensate for the transformations performed by the compiler on the very simple loop shown in Figure 2. The transformations considered in this example are strength reduction and dead code elimination. These two transformations acting together can replace a loop variable with other induction variables. Specifically, in this example, we will show how the debugger handles a request to print the value of the loop variable *i* at the beginning of the loop body. At that location, the variable *i* has been eliminated by the compiler.

```
do i = 0,n
   A[i] .
enddo
```

Figure 2: A simple loop

		Original Code	Strength Reduced	Dead Code Eliminated
1		$i \leftarrow 0$	$i \leftarrow 0$	
2			$iv1 \leftarrow A$	$iv1 \leftarrow A$
3		$(i \geq n)$?	$(iv1 \geq n \times 4 + A)$?	$(iv1 \geq n \times 4 + A)$?
4		goto *skip*	goto *skip*	goto *skip*
5	*body* :			
6		$t \leftarrow 4 \times i$	$t \leftarrow 4 \times i$	
7		$*[A + t] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$
8		$i \leftarrow i + 1$	$i \leftarrow i + 1$	
9			$iv1 \leftarrow iv1 + 4$	$iv1 \leftarrow iv1 + 4$
10		$(i < n)$?	$(iv1 < n \times 4 + A)$?	$(iv1 < n \times 4 + A)$?
11		goto *body*	goto *body*	goto *body*
12	*skip* :			

FIGURE 3: Intermediate code at three stages of compilation

The intermediate code is shown in Figure 3 as it is being transformed by the compiler. Some blank rows have been inserted in the first and third columns of code to visually preserve the correspondences between versions of the loop as it passes through the compiler. The first column of code is a straightforward translation of the source loop.

The second column shows the code after it has been subjected to strength reduction. Note the added statements at lines 2 and 9, and the changed code at lines 3, 7, and 10. The compiler has introduced a new induction variable $iv1$, to take over most of the uses of the loop variable i.

The third column is the code after it has been subjected to both strength reduction and dead code elimination. Dead code elimination has removed all the occurrences of i, which were superfluous after $iv1$ had been added. The two occurrences of $n \times 4 + A$ will be cleaned up by a later phase of the compiler that handles constant and shared expressions.

To print the value of i in the loop body, the front end of the debugger will try to insert the fragment **print** i after line 5 of the intermediate code. The correspondence between the body of the loop in the source program and line 5 of the intermediate code is determined by the front end of the debugger, using information recorded by the front end of the compiler. That stage of the mapping is not discussed here, and is assumed to exist. In the Original Code column of Figure 4, the fragment from the front end is shown as it would appear after insertion.

The stage of the debugger handling strength reduction can perform this insertion request by translating the fragment into its equivalent form, mapping the fragment insertion point before strength reduction to the equivalent insertion point after that transformation. (Because of the way I have numbered statements and left blank lines in this example, this location mapping is trivial.) Also, this stage of the debugger will translate any data locations named in the fragment, as needed. It appears that no translation of i is needed, because that variable exists before and after translation. The fragment, translated to compensate for strength reduction, is shown in the Strength Reduced column of Figure 4 as it would appear after insertion.

To actually accomplish the insertion request, the strength reduction stage of the debugger in turn asks the next stage of the debugger, the stage that compensates for dead code elimination, to insert the fragment **print** *i*. However, the compiler eliminated *i* during dead code elimination. This makes it impossible for the dead code elimination phase of the debugger to translate the fragment being inserted into something meaningful for the next stage of the debugger. Thus, the dead code elimination phase of the debugger reports to the strength reduction phase that the fragment **print** *i* cannot be inserted. This failure is depicted in the Dead Code Eliminated column of Figure 4.

A particular phase of the debugger attempting to insert a fragment could simply always report failure to its preceding phase, when faced with a report of failure from a subsequent phase. This is a valid action, but better courses of action may be available to that phase of the debugger. Better support of optimized code will result if phases try different translations of an insert fragment operation that has failed. The insert fragment operation requested by the preceding stage succeeds if any of the translations of this operation can be performed by the subsequent stage of the debugger. Thus, for each stage of the debugger, the success of a fragment insertion is the disjunction of the successes of all the translations of that operation.

For instance, when faced with the failure depicted in Figure 4, the strength reduction phase of the debugger could try a different translation for the original fragment. The strength reduction phase of the compiler created synonyms of the loop variable, new induction variables that are affine functions of the original loop variable. Using information recorded by the compiler, the strength reduction phase of the debugger can react to the failure of its first attempt at insertion by restating the fragment in terms of the new induction variable. In this case the fragment **print** *i* is translated into **print** $(ir1 - A)/4$ as it passes through the strength reduction phase of the debugger. Figure 5 shows the successful fragment insertion.

The success of a particular fragment insertion performed by a phase of the debugger can also depend on the conjunction of the successes of a collection of fragment insertions performed by the subsequent debugger

```
 1              |  i ← 0            |  i ← 0                |
 2              |                   |  ir1 ← A              |  ir1 ← A
 3              |  (i ≥ n)?         |  (ir1 ≥ n × 4 + A)?   |  (ir1 ≥ n × 4 + A)?
 4              |     goto skip     |     goto skip         |     goto skip
 5      body :  |                   |                       |
                   print i             print i                 print "??"
 6              |  t ← 4 × i        |  t ← 4 × i            |
 7              |  •[A + t] ← ...   |  •[ir1] ← ...         |  •[ir1] ←
 8              |  i ← i + 1        |  i ← i + 1            |
 9              |                   |  ir1 ← ir1 + 4        |  ir1 ← ir1 + 4
10              |  (i < n)?         |  (ir1 < n × 4 + A)?   |  (ir1 < n × 4 + A)?
11              |     goto body     |     goto body         |     goto body
12      skip    |                   |                       |

                     Original            Strength               Dead Code
                       Code               Reduced               Eliminated
```

Figure 4: Intermediate code, showing a fragment insertion

		Original Code	Strength Reduced	Dead Code Eliminated
1		$i \leftarrow 0$	$i \leftarrow 0$	
2			$iv1 \leftarrow A$	$iv1 \leftarrow A$
3		$(i \geq n)?$	$(iv1 \geq n \times 4 + A)?$	$(iv1 \geq n \times 4 + A)?$
4		goto *skip*	goto *skip*	goto *skip*
5	*body* :			
		print i	**print** $(iv1 - A)/4$	**print** $(iv1 - A)/4$
6		$t \leftarrow 4 \times i$	$t \leftarrow 4 \times i$	
7		$*[A + t] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$
8		$i \leftarrow i + 1$	$i \leftarrow i + 1$	
9			$iv1 \leftarrow iv1 + 4$	$iv1 \leftarrow iv1 + 4$
10		$(i < n)?$	$(iv1 < n \times 4 + A)?$	$(iv1 < n \times 4 + A)?$
11		goto *body*	goto *body*	goto *body*
12	*skip* :			

FIGURE 5: Intermediate code, showing a fragment insertion

phase. A translation of that particular fragment may involve several fragment insertions, all of which must be successful for the translation as a whole to be successful.

For example, if a fragment is inserted within a region of code that is really two merged flow paths, path-determiner breakpoints must also be inserted before the start of the merged region. These breakpoints determine which flow path was actually taken, the path to the region of code that actually contained the insertion point, or the matching region of code that was merged to it. If either the true breakpoint or its path determiner breakpoints cannot be inserted, then the whole insert operation fails.

6 Implementation of the Debugger

In general, an object-oriented framework for the debugger seems helpful, such as Cargill's representation of processes and frames as objects[Car86]. Portions of Cargill's design appear to be adaptable to the needs of our design. Some of the messages he describes would require extra arguments to account for the increased complexity caused by optimization. By changing the class hierarchy, the mappings can be decomposed into the individual mappings performed by the compiler.

A debugger built in accordance with our design would probably be part of an integrated programming environment. The large amount of information shared between the compiler and the debugger mak s a programming environment a desirable setting. Some of the code implementing the compiler could be adapted for use by both the compiler and debugger. Also, with minor modifications, language tools in current programming environments, such as source code browsers and editors, could be used by the debugger.

The debugging system for a compiler that performs optimizing transformations on its intermediate representation would comprise a single front end that maps from source code to intermediate code, a collection of transformation stages that map from intermediate to intermediate, and a final back end that maps from intermediate code to machine code. We discuss these phases in more detail below.

Front End

A particular front end for a debugger shouid be tailored to debug programs written in a specific source language. Ideally, the user interface allows the programmer to express as many debugging functions as feasible in the particular source language. The commands are parsed and compiled into the debugger's intermediate language, perhaps using parts of the compiler developed for the source language. The commands are then passed to subsequent stages of the debugger. The commands generated by the front end are either accepted or rejected by the back end, depending on whether or not the operation can be performed.

Transformation Stages

Each of the transformation stages of the debugger unravels the effects of a particular optimization. Requests from the preceding stage to create, insert, or remove fragments are translated into slightly different requests and sent to subsequent stages. A request to insert a single fragment can result in the insertion of many fragments in subsequent stages of the debugger. For instance, to place a breakpoint in a cross-jumped region, the original breakpoint must be supplemented with path-determiner breakpoints [Zel84]. References to memory in the original fragment are renamed to reflect the new locations of the desired values.

Back End

The back end maps the abstract machine presented between layers of the debugger onto the target machine. It can have a wide range of implementations, varying in efficiency, ease of implementation, and portability.

For fast prototyping, a simple interpreter for the debugger's intermediate language might be selected as a back end. This choice would yield a quick implementation, allowing the other parts of the debugger to be developed without delay. However, the resulting back end might be slow, adversely affecting the performance of the whole debugger.

If efficiency of the back end is a concern, the debugging operations sent to the back end, represented in an intermediate language for debugging, could be incrementally compiled, thereby reducing the amount of interpretation performed by the debugger. These compiled fragments could then be patched into the compiled code, reducing handshaking and context switches when performing the debugging operations implemented by these fragments.

In between these two extremes, is the possibility of a hybrid scheme that runs the unchanged code in its compiled form and interprets any changes. As an example, under UNIX, the back end could consist of two separate processes, the back end and the monitored process, with the monitored process controlled via calls to the operating system. We would expect that the efficiency of a debugger using this hybrid back end would be no better than a straightforward UNIX debugger having a complexity similar to our back end. Since the UNIX system calls employed are the bottleneck in these debuggers, efficiency would only be improved if the number of these calls were reduced, perhaps through more thorough analysis of the program

6.1 Interaction with Program Edits

A sophisticated programming environment should support incremental compilation. In response to program edits, the system incorporates corresponding "edits" into the compiled version of the program. If the program is being debugged when an edit occurs, it may be desirable to edit the state of the monitored process, without restarting the program, if possible.

Although edits of the process text and state could be implemented using the *patch* and *unpatch* primitives, these debugging operations are inappropriate for supporting process edits. Edits of the process text differ from the patches applied to support debugging. Debugging patches are transitory, being applied to return control to the user or to monitor some condition. Edits of the process text are more permanent, reflecting a decision by the programmer to change some part of the program.

6.2 Impure Code

Some of the proposed intermediate instructions can be called meta-instructions, since they can alter the text of the program. That is, they represent, and are easily implemented as, self-modifying code. This implementation may be a problem when debugging programs on machines that were designed with the assumption that self-modifying code is rare or unnecessary. For example, such an assumption might be made in designing an instruction pipeline, in order to improve its speed. However, this restriction will be a problem for all interactive debuggers written for such architectures. The abstraction present in our design actually gives greater freedom to easily change the machine dependent part to suit the particular hardware.

Since we can specify debugging requests in an intermediate language, there are several possible implementations of requests. We can convert code modifications into data modifications, reducing the number of modifications to the instruction stream by modifying data instead. To accomplish this conversion, the back end allocates a new memory cell to hold a flag, and adds a branch, dependent on the flag's value, around the patched fragment. When an *unpatch* is performed on this fragment, the action of removing the fragment can be performed by changing the value of the flag.

6.3 Debugging at Lower Levels

A programmer using any source-level debugger will in some instances need to use a lower level of abstraction. For conventional debuggers, this lower level is the generated machine code. The desire to change to this level can occur when the user is unsure of the apparent behavior of generated code. This uncertainty can arise from bugs caused by errors in a compiler, defects in the hardware, the user's disbelief of the events actually happening, and a host of other causes.

In our model, each of the interfaces between stages of the debugger represents a level of abstraction that may be interesting to a programmer. By examining the program at various levels, an experienced user can understand the effects of optimization. When the user inspects a program object at one viewing level, the corresponding object or objects at a different level can be emphasized. By displaying these correspondences between objects from different levels of the program, the state of the transformed versions can be related to each other, and back to the original source program. For example, if the user selects a program location in the source, all the possible corresponding program locations in an intermediate language version of the program could be highlighted

Also, optimizations that hinder debugging can sometimes be avoided by dropping to a lower level of abstraction. At lower levels of abstraction, the insertion of a fragment is more likely to succeed, because the translation of that insertion into machine code is more straightforward. For example, setting a breakpoint in an inlined procedure would entail patching fragments into all the copies of the procedure body replacing the call sites. The patch operation for one of these copies might fail, because of optimizations occurring only in that copy. By dropping to a level where the user can set breakpoints for individual call sites of a procedure, the one troublesome patch can be avoided.

7 Research

Previous Experience

As part of Rice University's $\mathcal{R}^n$ project [CCH+87] to build a programming environment for scientific software, we designed and built EXMON, a debugger for large, computationally intensive programs[CH87]. In the course of that effort, we encountered several problems in extending our design to provide more functionality. We chose a hybrid design for EXMON, allowing the monitored process to be a mixture of compiled and interpreted code. This mixture allowed trusted parts of the program to be executed quickly, but with minimal debugging support, and suspect portions of code to be interpreted, with greater debugging support. For our representation of interpreted subparts of the program, we chose the abstract syntax tree, which was the source representation used by $\mathcal{R}^n$.

Unfortunately, this representation proved to be too close to the source language, and too closely determined by the needs of the source editor. The addition of new source language dialects, the evolution of the source editor, and changes to other parts of the programming environment repeatedly changed the structure of these trees, and these changes in turn required corrections to the interpreter and other parts of the debugger. This experience with EXMON provided some of the impetus and insight for this design. The research we propose is an attempt to overcome some of the problems with conventional debuggers that we have encountered. We hope to demonstrate that this new design for debuggers is feasible, effective for debugging optimized code, portable, and efficient.

Available Tools

For our debugger's intermediate language, we have chosen to extend ILOC, a fairly simple three-address code in use at Rice University. ILOC was designed as an intermediate language for optimizing compilers, and is reasonably suitable as a foundation for our debugger's intermediate representation. A compiler for ILOC exists, and already has several transformation phases, including value numbering, dead code elimination[Ken81], strength reduction[CK77], and partial redundancy elimination[MR79][DS88].

A fully interpretive back end for this extended version of ILOC exists. We anticipate changes to the intermediate language, so the efficiency of the interpreter is not yet a great concern. As the language becomes more stable, we will investigate efficiency issues in the back end. A hybrid back end has been designed by Mike Jones and Bob Hood, which supports a compiled program text patched with interpreted fragments. Using this hybrid back end, the efficiency of common debugging operations will be studied. A modified version of the R^n debugger will be used for the initial version of the user interface and front end of the debugger.

Research Plan

We are designing parts of a debugger-compiler pair, trying to preserve opportunities for optimization by the compiler, rather than constrain optimizations to preserve debugging support. Thus, the emphasis of this research is on debugging optimized code, rather than optimizing debuggable code. Debugger phases compensating for the transformations dead code elimination, strength reduction, value numbering, and partial redundancy elimination are being designed.

- Dead code elimination is an important inclusion because its effects are difficult to reverse. Hennessy in particular noted global dead code elimination as an optimization that impaired debugging support[Hen82]
- Strength reduction was included because the multiple translations easily obtainable from this transformation are an important aspect of the fragment translation. Strength reduction and dead code elimination together show promise as transformations whose mappings are separable, but which together produce a non-trivial transformation, in this case loop induction variable elimination. Also, we feel that the intermediate language used in the compiler may be an interesting medium for the private communication of transformation information between the strength reduction phases of the compiler and debugger
- Value numbering and partial redundancy elimination were added to give the collection a realistic amount of complexity. Value numbering was chosen over similar data optimizations because it is currently used in the R^n compiler developed at Rice, as is partial redundancy elimination. The effects of value numbering on the debuggability of the generated code should be similar to that of any algorithm for folding constants and eliminating common subexpressions

Debugger transformation stages for register allocation, cross jumping, loop unrolling, and procedure integration also show promise in compensating for these transformations. However, a debugger phase that compensates for register allocation probably will not be studied directly as part of this research. A realistic treatment for register allocation would also involve all the other transformations found in the code generation phase of a compiler, such as instruction selection, instruction scheduling, and peephole optimizations. A debugger phase for the resulting mapping would be hard to design, implement, or study. If implemented, the performance of the debugger phase in terms of a particular back end transformation would be hard to evaluate, because the effects from all the different back end optimizations would be difficult to separate and measure.

We conjecture that a straightforward composition of many simple mappings can be used to compensate for many compiler transformations. For example, simple cross jumping appears to only produce simple merges, while repeated cross jumping produces more complicated mappings [Zel84]. The monolithic mapping developed for Navigator represents the composition of the individual simple merges that contributed to the accurately merged regions. We will compare the capabilities of a single monolithic mapping [illegible] the repeated cross-jumpings with those of a mapping produced by conceptually composing simpler mappings, each of which represents only simple merges without repeatedly merged regions.

The knowledge of when to inhibit a transformation may be difficult or inefficient to garner in a compiler that leaves all the mappings uncomposed. As an example of this effect, the composition of simple mappings that we propose may unfortunately yield a poorer handling of cross-jumping than that provided by a monolithic mapping. The level of debugging support should be similar to that described by Hennessy[Hen82], and will be compared with it.

We expect that changes to the collection of transformations will be simpler if the mappings of the transformations performed by the compiler are left uncomposed. We will investigate one aspect of this by studying the effects of interchanging and repeating the various transformation phases of the compiler and debugger. It may be the case that each stage of the debugger will contain hidden assumptions about the nature of the stages following them, and that disturbing the ordering of the stages seriously impairs debugging support. The character and seriousness of the degradation in debugger support resulting from these changes will be studied.

For a debugger using the monolithic mapping, the interaction between a new transformation and each existing transformation must be considered. We feel that by leaving the mappings uncomposed, the interaction between existing transformations and an additional one is reduced. We will investigate this theory by examining the effects of adding another control flow optimization, such as loop peeling[Lov77] to the two optimizations researched by Zellweger, and noting the impact on the overall design.

The ability to present different levels of abstraction, described earlier in section 6.3, will be studied. The adoption of this mechanism as a technique for following and understanding source to source transformations will also be attempted. Among the important problems remaining in developing this tool are mapping the cursor from one level to the next, displaying this information to the user in an understandable manner, filtering out extraneous intermediate language statements added during intermediate steps of the compiler's optimization process, and defining a small but useful set of debugging commands at these lower levels. We will study the benefits and problems resulting from allowing access to intermediate language representations of the program.

8 Conclusions

We have described a new design for debuggers that uses a structure analogous to that of modular compilers. The debugger is separated into stages corresponding to the parser, optimizer, and code generator of the compiler. Each of these stages of the debugger is responsible for unraveling the transformations performed by the corresponding stage of the compiler. The stages of the debugger communicate using an intermediate language derived by extending the compiler's intermediate language with primitives to support debugging.

This approach is designed to improve the cooperation between the compiler and the debugger, especially when debugging optimized code. Our framework should allow many debugger actions to be translated into intermediate code and perhaps machine code, resulting in more efficient execution of these actions. The [illegible] design [illegible] should promote portability and modularity of the debugger. When the [illegible] are modified to support a new source language, architecture, or transformation, [illegible] reuse of the debugger code.

References

[Car86] T. A. Cargill. Pi: A case study in object-oriented programming. In Norman Meyrowitz, editor, *OOPSLA '86 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 350–360, 1986. Also appearing as SIGPLAN Notices Vol 21, Number 11.

[CCH+87] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. M. Torczon, and S. K. Warren. A practical environment for scientific programming. *IEEE Computer*, November 1987.

[CH87] B. Chase and R. Hood. Selective interpretation as a technique for debugging computationally intensive programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, June, 1987*, pages 113–124, June 1987.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[DEC88] Digital Equipment Corporation, Maynard, Massachusetts. *VMS Debugger Manual, Version 5.0*, April 1988.

[DS88] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies'. *ACM Transactions on Programming Languages and Systems*, 10(4), October 1988.

[Hen82] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[Ken81] Ken Kennedy. Program flow analysis: Theory and applications. In Steven S. Muchnick and Neil D. Jones, editors, *A Survey of Data Flow Analysis Techniques*. Prentice Hall, 1981.

[Lov77] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, March 1977.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.

[Zel84] P. T. Zellweger. *Interactive Source-level Debugging for Optimized Programs*. PhD thesis, University of California, Berkeley, CA, 1984.

MDB – A Parallel Debugger for Cedar

Perry Emrath *Bret Marsolf*

emrath@csrd.uiuc.edu marsolf@csrd.uiuc.edu

Center for Supercomputing Research and Development
University of Illinois at Urbana–Champaign
104 S. Wright, Urbana, Illinois 6180

Mdb is a parallel debugger that was developed for debugging programs on the Cedar multiprocessor architecture, a prototype machine assembled at the Center for Supercomputing Research and Development, University of Illinois. Mdb was designed to provide users with the ability to interactively examine the data and control structures of an executing parallel program.

The design of mdb is different from many interactive debuggers in that rather than being a separate process which controls the target program through the operating system, mdb is a package of functions that are linked with the target program. While this means that mdb is not totally isolated from the target, it has the advantages of being relatively simple to implement and can examine large amounts of program state very efficiently. Both these advantages stem from the fact that program state is directly accessible without requiring any operating system services.

The structure of the mdb package was motivated by the way processes execute on the target architecture. The Cedar architecture is a multi–cluster architecture with a hierarchical shared memory, with each cluster containing multiple vector processors. A parallel program runs on Cedar as a parallel process which consists of multiple tasks. Each of the tasks within the process executes on a single cluster, and may run on multiple processors within the cluster, from one up to all (8) of the processors in the cluster. The Xylem operating system, based on Unix, provides services to support these abstractions.

To control the process it is necessary for the debugger to have control over all tasks in the process. This is achieved by having the ability to communicate between tasks by placing data in shared memory and sending interrupts using facilities in Xylem. Mdb comes in two flavors, a standard version which is used by linking a program with the

This work was supported by the U. S. Department of Energy under grant number DOE DE FG02 85ER25001 with additional support from the U. S. Air Force under grant number AFOSR 90 0044 and I.B.M. through the C.S.R.D. Affiliates Program.

-ldebug library, and a stripped down version which is included in the C run-time library. The minimal version gets linked with every program, so users always get at least this level of debugging support, and can simply consider it part of the operating system.

To allow entry into the debugger, mdb initializes the signal handler interface to have most traps and some signals handled by the debugger. This initialization is performed as part of the run-time library startup code for the first task in the process. As other tasks are spawned in the process, these interfaces are copied so that all tasks will normally enter mdb when any of the caught signals occurs. Mdb catches the QUIT signal, so while a program is executing, the user may interrupt it and use mdb to examine the state of the process. Alternatively, the user may insert a call to *breakpoint()* or *panic()* in order to enter mdb at that point.

The first task that enters the debugger becomes the controlling task and interrupts all the other tasks within the process as quickly as possible, using facilities available in Xylem. As the other tasks enter the debugger, they block to wait for further commands from the controlling task. The controlling task then sets about reading and executing user commands.

To the user, mdb is similar in nature to a conventional breakpoint debugger, such as adb. However, mdb knows about the multi-tasking model described above, as well as the architecture of the Alliant multi-processor. Commands are provided which allow the user to examine the entire state of the program, which includes shared memory, private memories, any of the registers in any of the processors, and certain system state variables. Memory can be displayed in a variety of formats, for example hexadecimal or floating point. In the full version of mdb, an instruction format is provided to allow program disassembly.

All commands are executed by the controlling task, which limits the scope of the commands that can be performed. The controlling task is able to examine the memory shared between all tasks and to examine the memory that is private to itself. The controlling task is not able to examine memory that is private to any other task. To examine the private memory of another task it is necessary to change which task is the controlling task. A command switches (or selects) the controlling task by having the current controlling task send a message to the task which is to become the controlling task. After control has been transferred it is possible to examine the memory that is private to the new controlling task.

In addition to examining the memory of the task, the controlling task can also examine the hardware specific state that was saved when the task entered the debugger. This saved state includes general purpose registers, floating point registers, vector registers, and concurrency registers. These values are saved for all of the processors assigned to the task. Similar to the way that tasks are selected, a processor can also be selected and then all the saved values for that processor can be examined.

In the full version of mdb, the user can load the symbol table from the executable file, after which symbolic names can be used in expressions. Values can also be displayed as symbolic addresses. Decoded instructions and call stacks will also be displayed with addresses in symbolic format.

After the user has examined the state of the process, the proceed command allows all tasks which have not encountered an error to continue execution. Tasks which have encountered an error remain waiting in mdb until the process exits. A few other special commands are also provided. One returns the (start address of the) handler for a specified signal. Other commands give the user information about the object file from which the symbol table was loaded.

Since a minimal version of mdb is linked with all programs, it must also function when the program is run non-interactively. When a process is started, mdb attempts to determine if the process is being executed interactively or not. If the process is being executed interactively, then the debugger functions as described above. If not, then when the debugger is entered it executes a short list of commands, placing the output in a file, after which the process aborts. The commands attempt to provide the user with enough information to determine where the task was when it entered the debugger and why it entered the debugger.

Mdb has been developed as a tool to allow the user to examine a currently executing process, but it does not allow the user to make any changes to the process. Future work is being considered to allow the user to change values in the process and to set breakpoints. Even without these enhancements, though, mdb has proven to be a very useful tool for debugging parallel programs. The capabilities of mdb have evolved as it was used to debug itself during development and the user interface was refined repeatedly as continued use suggested changes to make debugging easier. Feedback from the user community has generally been favorable and a significant number of real bugs have been easily and quickly found once the faulty program was linked with ldebug.

A replay mechanism within an environment for distributed programming

S. Chaumette

LaBRI, Laboratoire Bordelais de Recherche en Informatique, URA numéro 1304, Université Bordeaux-I, 351 Cours de la Libération, 33405 Talence, FRANCE.

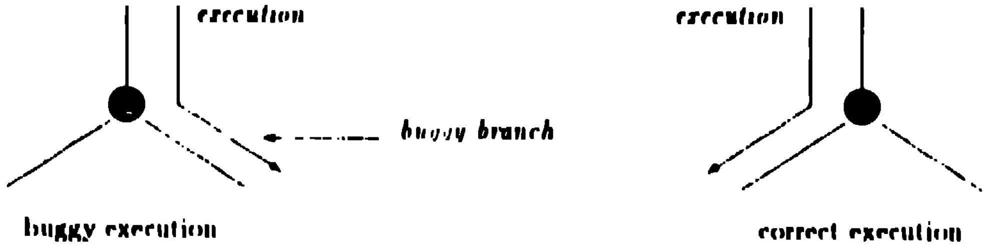
Most of the languages for distributed programming provide non-deterministic constructs. Although enabling efficiency enforcement and increasing expression facility, these constructs lead to the need of replaying an execution when willing to *debug an application*. The aim of this talk is to describe a *replay mechanism* and how it is used for debugging purpose within a *programming environment*. The description will encompass many aspects of this mechanism, from its semantics up to a use example which will be emphasized by means of a debugging session; an efficient implementation in a centralized simulator will also be expounded. I will conclude with remarks concerning non-intrusiveness of debugging mechanisms. This work* is part of a research carried out at LaBRI (Laboratoire Bordelais de Recherche en Informatique) which consists in designing a programming environment for distributed memory machines.

The model: proving non-intrusiveness of debugging mechanisms

Our model is that of *explicit parallelism*, that is a program is expressed as a set of processes communicating by message-passing, via a medium called a communication port (as in CSP). The non-deterministic primitive it provides (which is called *ready*) enables to select among a set of ports, one from which a message is ready to be received. The replay technics requires a reference execution during which local traces with minimum information are recorded. *Local monitors* are then derivated from these traces and are used to control a reexecution. It has been proven that this mechanism has no influence over the behavior of the set of processes under control: it is non-intrusive. The fact that the semantics of the language and that of the controllers have been modelized makes it possible to prove this property.

Replay: a debugging support mechanism

A non-deterministic program can be drawn as a tree, each branch of which is a possible path of the non-deterministic behavior. Assume a bug occurs in one of the branches; then, another execution may follow another path, preventing the bug to take place again. The replay mechanism ensures that the same branch is used, what will enable to reproduce the bug and eventually to understand it so as to correct it.



It is almost impossible to debug a non deterministic program (what is most often the case regarding distributed programs) without any replay mechanism.

Implementation: debugging versus efficiency

When implementing this mechanism on a *distributed machine*, in the most straightforward manner, some problems arise: the traces, which are generated locally (at each processor node), need to be collected on the host computer; the number of processes during a controlled execution is twice the number of processes of the application since one controller is created for each process;

*This work is supported by the French Coordinated Research Program C^3 of the CNRS, CNRS and Région Aquitaine

the recorded traces must be downloaded from the host computer into the controllers. This implies overhead and heavy load of the communication network. Anyway, this implementation is a good one, in that *it does not require recompilation of the application*, what is a time-saving and safe debugging technics.

Our environment provides a *centralized simulator-debugger* which is implemented so that the simulation of any number of processes requires a sole UNIX process. This makes it possible to build the controlling mechanism in the code simulating the communication system: this prevents increase in the number of (simulated) processes by suppressing replication of the controllers; the overhead due to the collection and distribution of traces is also discarded. No recompilation is needed.

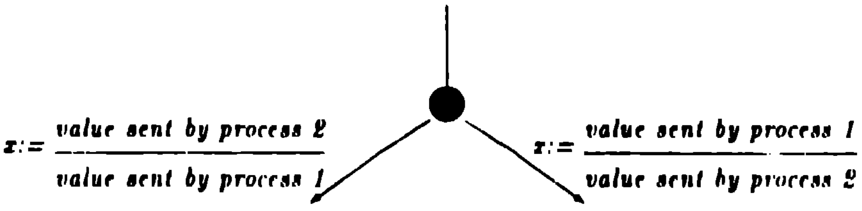
It should be noted that it is possible to implement this mechanism in an efficient manner on a distributed machine.

Debugging session

There are many *distributed applications*, such as ray-tracing or matrices block calculus, which can be expressed using non-deterministic constructs. For the sake of demonstration, use of the mechanism can be shown on a simple problem which consists in computing the quotient of two numbers. Assume two processes (process 1 and process 2), each computing a value. Another process (the one in which we are interested) receives these values (via ports p1 and p2) and computes their quotient x=(value sent by process 1) / (value sent by process 2). This last operation can be achieved using a non-deterministic algorithm in order to enforce efficiency by receiving the values "as soon as possible".

```
begin
   [...]
   set:=[p1,p2];
   p:=ready(set); receive(p,x1);
   set:=set-p;
   p:=ready(set); receive(p,x2);
   x:=x1/x2;
   [...]
end.
```

The sample buggy program



x:= value sent by process 2 / value sent by process 1
x:= value sent by process 1 / value sent by process 2

The two possible behaviors of the sample program

A proposed solution is shown on the above figure. Unfortunately, such a program will exhibit not only a non-deterministic behavior, but also a non-deterministic result: it is buggy (the error is made clear on the figure displaying the two possible behaviors of the process). If one could trace, at run-time, the values of the different variables when the result is not that expected, then the problem would be clear and the solution straightforward. This is why the replay mechanism is necessary: without it, any later experiment may lead the program to go through the branch which leads to the correct result, making any debugger useless. The replay enables to reproduce the buggy execution. The correction to apply to the program is to test which is the port on which we are receiving, so as to know if the received value is the numerator or the denominator (i.e. comes from process 1 via port p1 or from process 2 via port p2).

This is only one example, but the reader can easily imagine how replay can be useful in various (more complicated) cases.

Conclusion: debugging and non-intrusiveness

In this talk, I show the *importance of a replay mechanism* as a support for debugging, the importance of its *proof*, and how it can be *efficiently implemented in a centralized simulator*. At a higher level of abstraction, the approach used for this tool, especially regarding *modelization* and *non-intrusiveness proof* (which is necessary for debugging purpose), should be kept in mind when designing any debugging mechanism.

AN INTEGRATED APPROACH TO REPLAY ANALYSIS OF MESSAGE-PASSING PARALLEL PROGRAMS

CHAD HUNTER

The MITRE Corporation, Burlington Road, Bedford, MA 01730
Email: chad@linus.mitre.org
Phone: (617) 271-2146

ABSTRACT

Complexity and nondeterminism can produce unpredictable results in parallel programs. An effective parallel debugging environment must manage both of these impediments. Described here are five novel solutions resulting from a combination of existing techniques for execution replay and behavior analysis: (1) examination, analysis, and modification of events that have yet to occur; (2) cooperative analytical strategies that combine conventional debugging, graphic state mapping, and behavior-oriented analysis; (3) experimentation with event ordering in suspect program fragments; (4) communication-related performance measurement; and (5) analytical capacity that is potentially scalable with system size.

1. INTRODUCTION

Debugging is an important activity in program development and maintenance [1]. It can be particularly difficult with parallel programs, since they are often complex [21 and can behave nondeterministically [3]. To a great extent, this complexity arises from interactions between multiple loci of control in what are typically large applications. However, sequential and parallel debugging differ most in that concurrent programs may suffer from intermittent errors. Such problems are manifestations of *races*. This is a condition that may exist when two or more threads of control participate in unsynchronized access to a common resource.

Races are well illustrated by the the *readers/writers* problem [4]. Imagine several independent processes that manage a bank account. Each of these may carry out three types of operations: obtaining an account balance, crediting deposits, and debiting withdrawals. The steps required for each operation are shown below:

Obtain Balance		**Credit Deposit**		**Debit Withdrawal**	
READ	Balance	READ	Balance	READ	Balance
		ADD	Amount to Balance	SUBTRACT	Amount from Balance
		WRITE	Balance	WRITE	Balance

Balance operations merely read the account balance without changing it. Consequently, any number of balance operations may occur successfully in parallel. In contrast, both deposit and withdrawal operations rewrite the account balance. If two such operations on the same account overlap, the balance will reflect only the last operation to complete. Transactions can be guaranteed only if operations that write to the same account are serialized. Without this synchronization, a race condition exists, and transactions are lost unpredictably. Furthermore, such nondeterministic results vary from run to run.

When attempting to identify the source of the problem, a programmer must contend with multiple processes. In addition, since the error is dependent on relative process speeds, it may only occur intermittently and may not be readily reproducible. Errors may also be masked by a debugger that impacts process speeds nonuniformly. This phenomenon is often referred to as the *probe effect* [5].

We are currently developing a prototype debugging environment called PARADIGM. This system is being used to investigate techniques for debugging programs on unmodified message-passing multicomputers. PARADIGM addresses both nondeterminism and the probe effect through execution replay [6–13]. This approach permits intermittent errors to be captured and reproduced. Execution replay is discussed in section 2.

PARADIGM manages complexity by means of behavioral abstraction [13–23]. This approach treats a parallel program execution as a collection of event streams. Distinct events represent selected program actions. By recognizing and matching related events, a behavioral abstraction system can reveal interactions, dependencies, and trends. This strategy is discussed in section 3.

Considered individually, execution replay and behavioral analysis are valuable debugging tools, but together, their potential exceeds the sum of their individual contributions. The benefits of this synergy are discussed in section 4.

2. EXECUTION REPLAY

Errors in parallel programs often involve races. Their reliable capture and study therefore require that both nondeterminism and the probe effect be addressed. Execution replay is one approach that has been applied successfully. With this scheme, debugging is divided into two phases. In the recording phase, information critical to the ordering of events is collected during execution. Checkpoints may also be made. In the replay phase, the execution is reproduced by enforcing the same partial ordering of events as recorded previously. Consequently, errors arising from nondeterminism are preserved. Furthermore, the replayed execution may be paused, single-stepped, or rolled back to a checkpoint without impact from the probe effect. During the recording phase, however, perturbation results unless special-purpose hardware is used to collect the event ordering information [24, 25]. A software-only approach must therefore minimize its data gathering during recording or risk invalidating the information collected. If sufficiently unobtrusive, software-based instrumentation might be left in place.

Future generations of parallel machines will most likely have at least some monitoring support hardware [26]. At the present, however, these facilities are costly, and the debuggers that use them are nonportable. For these reasons, PARADIGM employs a software-oriented approach that makes full use of existing hardware, but does not require the addition of debugger-specific equipment. Machine dependence is confined to an in-place component called the *event interface*. This portion of the debugger exploits any existing hardware to record and replay a parallel execution. During replay, the event interface also passes information on program behavior to resident analysis facilities.

The event interface supports passive collection of program events [27]. In other words, rather than capture data with special-purpose libraries or user-supplied instrumentation, kernel-resident facilities capture events as they occur. Kernel dependence has its drawbacks, namely, that porting the event interface to a new platform requires expertise in kernel programming and access to kernel source code. However, the benefits of implementing the event interface at the kernel level are substantial, for example:

- No special-purpose hardware is required, but any that is present can be used effectively.

- The probe effect is kept acceptably small, even without the aid of special-purpose hardware.

- Compiler or library modifications are obviated, and debugging is language independent.

- Replay capabilities are extended to conventional debugging environments in a transparent manner.

- Information required for performance analysis can be gathered.

- Debugging is still possible when source code is unavailable.

- Debugging information is maintained in a separate address space, where it cannot be modified by the program being debugged.

There are two styles of execution replay, which are distinguished by a logical or physical view of time. The use of logical time was pioneered in Instant Replay [7]. This system is based entirely on the order of events. Its recording phase captures the order of operations on shared resources. The same ordering is then enforced during subsequent replays of the execution. Since timestamps are not used, replay fidelity is independent of system clock precision. Such accurate replay is itself a valuable debugging tool. Having witnessed and captured erroneous behavior, a programmer may often be able to verify code corrections by subjecting them to the same conditions that previously elicited an error. However, this is not always possible. If an error directly involves the order of operations on a shared resource, a correction invalidates the captured history, and replay is not possible.

A sharply contrasting style of replay based on physical time was introduced with BugNet [8]. This system timestamps the messages of each process as they arrive during the recording phase. Replay is effected by delivering each message at approximately the same time as in the recorded run. BugNet's reliance on physical timestamps constrains its replay fidelity to system clock precision. However, replay is centered on external message events and is independent of intraprocess event order. Consequently, it is possible to verify any correction that does not affect external message events.

PARADIGM supports replay based on logical time. Unlike Instant Replay however, each event in the underlying partial order diagrams [28] includes a timestamp. The result is a timestamped partial order. Originally included to support physical time-based replay, these timestamps now appear to be more useful for performance analysis. This is discussed in section 4.4.

```
                ⋮
msg_id = irecv(3, buffer, BUF_SIZE);
for (i = 0; i < 1000; i++)
    if (msgdone(msg_id))
        break;
    else
        do_work(i);
if (i >= 1000)
   msgwait(msg_id);
                ⋮
```

Figure 1. Erroneous Program Fragment

Consider the program fragment in figure 1. On the Intel iPSC/2 multicomputer, irecv(), msgdone(), and msgwait() are all communication-related system calls. The irecv() in the first line requests that a message of type 3 be placed in buffer. Irecv() does not await the arrival of an appropriate message. Instead, it returns a handle that may be used later to determine the receive status or to await message delivery. This handle is passed to the msgdone() call within the for loop. If a message of type 3 has arrived, msgdone() returns 1, indicating that the receive is complete. Until that time, however, msgdone() returns 0. Should the loop be exhausted before the message is delivered, msgwait() suspends execution pending its arrival.

The timestamped partial order for this fragment is shown in figure 2. In this simplified diagram, the single timeline represents the history of events for the node on which the fragment was active. Events occurring prior to the fragment's invocation are not shown. Here event 100 corresponds to the irecv() system call. Further along, the incoming arc represents an arriving message. As with the irecv() event, the arrival is timestamped. The identity of the originating node is also recorded. (In reality, the arrival information is stored in the record for event 100. It is shown separately here to highlight the message arrival's temporal relation to program events.) The final event shown is numbered 620 and represents a successful msgdone() call. Between this event and the irecv() are 519 intervening and unrecorded events. These correspond to the msgdone() events that could not succeed until the arrival of the message at time 1:08. As explained below, these events need not be recorded. Replay requires the capture of only those events that involve communication. Also not recorded are message contents, as these are regenerated by the replayed program.

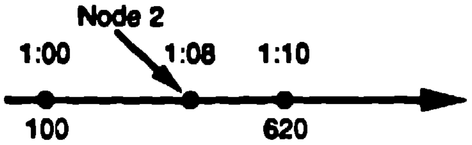


Figure 2. Timestamped Partial Order for Erroneous Program Fragment

Imagine that the program fragment in figure 1 produces incorrect results when do_work() is executed more than 500 times. The error is dependent on the relative speeds between the process that contains the fragment and the process that supplies the message. Consequently, the problem only occurs intermittently. Furthermore, it may never manifest itself if a traditional debugger is used to study the fragment's process in isolation. In this case, monitoring activity may slow the process, causing the message to arrive much earlier in its execution.

PARADIGM's execution replay facilities ensure that any errors, including races, are preserved. No special action need be taken until an error occurs. At that time, a timestamped partial order is recovered using the replay system. The behavior can then be reproduced by comparing communication events in the re-executed program against those in the timestamped partial order. When the irecv() is re-executed, the replay system can determine that node 2's message is to be received. Should a message from a different node arrive earlier, it is ignored. The next 519 events are unrecorded msgdone() calls. By their absence, the replay system infers that these reproduced events must fail regardless of the arrival of node 2's message. At the 520th msgdone() event, the call is allowed to succeed. If the expected message has not yet been received, execution is suspended pending arrival.

Because replay is not dependent on timestamps, PARADIGM provides accurate replay even when high precision system clocks are unavailable. Imagine that the programmer replays the erroneous execution within a conventional debugging environment. After determining the cause of the error, he decides to modify do_work() and recompiles his program. The new executable may be tested using the replay system and the original timestamped partial order. If the error has not been corrected, it will manifest itself again.

A problem occurs when the changes to be made affect the order of communication events. For example, imagine that the programmer corrects the fragment in figure 1 by limiting the `for` loop to 500 iterations. This modification also reduces the number of `msgdone()` events and, hence, invalidates the event sequence recorded in the timestamped partial ordering. When the replay system detects such a deviation from the recorded partial ordering, it will disable monitoring and return to the recording phase. Debugging can be subsequently resumed from the newly recorded branch of the modified execution.

3. Behavioral Abstraction

Execution replay helps a programmer cope with complexity in two ways. First, it allows intermittent errors to be examined interactively, and this enables the user to better focus his attention. He may single-step, breakpoint, or even alter execution. Second, it provides a simplified representation of program behavior, being based on the event model. This is not sufficient, however, as even a sequential program can generate thousands of events per second. Although a monitor can assimilate such quantities of information, it is unlikely that a user can. Parallelism only magnifies the problem. One solution is to allow the user to filter out all but the events he is interested in. This can still be an unmanageable volume of events, however, and the relationship between events can be unclear.

PARADIGM's analysis capabilities will be provided by node-resident *monitor agents*. By themselves, behavioral abstraction systems [14–23] do not address the probe effect or errors arising from nondeterminism. In our system, however, the event interface converts reproduced program actions into a stream of primitive events. These are provided to the monitor agents, where they are compared against patterns in rules. This comparison may result in abstract events which describe higher level behaviors. Abstract events can, in turn, be combined, and may represent activities across processors.

Behavioral abstraction plays four major roles in PARADIGM. Its first role is to enable automated reasoning about program behavior. This capability is needed in order to identify meaningful occurrences and trends from the great volume of events posted by parallel programs. The second role is to control program behavior either manually or based on this analysis. Having control over a program's behavior will enable the user to enforce specific event orderings, alter communication patterns, or take whatever other actions are necessary to isolate an error. The third role involves direct querying of the debugging environment by the user. Debugging is a creative activity, and will often require the full faculties and intuition of the programmer. These can best be brought to bear through an integrated query facility that allows the programmer to test assertions about a program's execution and to draw his own conclusions. The fourth and final role is to drive the user interface. Graphic displays enable the programmer to rapidly obtain an overall understanding of global state and to identify particular program components which are suspect.

Conclusions

- Debugger should be part of a integrated environment that provides a variety of views into a program
    - A view of the program for controlling the execution with break-points, checkpoints, or other means
    - A view of the source code that supports debugging along with editing and compilation
    - A view of memory that supports annotated displays, symbolic browsing, and system information
    - A view of files in use that shows both status and contents including any buffers
    - A view of program calling sequence with parameters by name and value
    - A view of inter-task relationships for parallelized code - timing, status, events, locks, semaphores, etc.
    - A view of generated low level code for corresponding high level source

Conclusions - Specific Features

- Integration with standard compilers and editors
  - Debugger should support full syntax used in a program including pre-processor directives and intrinsic functions
  - Symbolic information with all compilations as the default
  - all levels of optimization
- Fast response time and easy to use
- Debugger should work for running programs, core dumps, and checkpoints
- Fast conditional break-points and memory watch break-points
- Bounds checking for arrays
- A symbolically annotated window into memory with selectable formats and symbolic searching
- A display of memory layout - a memory map
- Tracing of selected code segments

- Code splicing - a way to insert new code into a running program
- For parallel processing:
    - per processor windows and break-points
    - memory reference trace for shared variables
    - complex queries of parallel variables or arrays
    - message passing traces for distributed memories
    - graphical displays of events across all processors
    - reliable real-time clock

bdb: A library approach to writing a new debugger

Benjamin Young, Cray Computer Corporation

Abstract

bdb is a new source level debugger being developed by Cray Computer Corporation. Work has been underway since May, 1990 and it was officially released to customers in October of 1991.

To accomplish our design goals and to simplify implementation, we chose a library approach to the debugger design. We split debugger functionality into several different areas (many of which were common areas for other tools). For each area we designated a new library to be written or used existing libraries from other sources where possible.

The end result of this design technique is a very modular debugger which has been or can be extended to multi-tasking debugging, distributed debugging, process monitoring, symbol table debugging, dump debugging, and many other useful tools.

Preface

This is a work in progress paper about a new debugger named bdb being designed and developed by Cray Computer Corporation (CCC). Although there is much work still to do on bdb, it has been used internally at CCC since November of 1990 and was first released to customers in October of 1991.

bdb is a source level debugger designed to debug both C and Fortran user code. It is able to functionally debug single and multiple processes as well as single and multi-tasked applications. It currently supports three user interfaces (a dbx-like line mode, an X Window system Athena Widget mode, and an OSF/Motif mode) with a forth user interface currently being tested internally (OPEN LOOK mode).

Initial bdb design goals

When work started on bdb, we had several design goals in mind. These included:

- Full symbolic capability for Fortran and C.
- Ability to debug multiple independent processes.
- Ability to debug macro/micro tasked processes.
- Ability to connect to several different user interfaces.
- Ability to rapidly prototype new user interfaces.
- Ability to share debugger data with external processes (intended mainly for data visualization).
- Ability to debug distributed processes.
- Ability to debug optimized C and Fortran code once the symbol tables supported it.

Functional decomposition for bdb

To support these goals and to allow for incremental development for each design goal, we identified five main areas of debugger functionality that should be split out of the debugger into their own library or set of libraries. These were the areas of symbolics, data displays, process control, user interface, and distributed communications.

For the area of symbolics, a new library (libsym) was developed that provides a simple (or at least a "more" simple) interface to the loader and compiler generated symbol tables.

A new library named **libdis** was created to handle all data displays needed by the debugger. These included disassembly displays, symbolic displays (where data is formatted via a symbolic type definition), and dump displays.

The process control library (**libbdb**) provided the interface between the debugger and the processes being debugged. All communication between **bdb** and debugged processes occurs via this library.

The user interface required several libraries. The lowest level library was the Tcl library (**libtcl**) that provided a consistent low level, string based, interface to the debugger. On top of that was built the windowing library (**libwatson**) that provided the windowing capabilities needed by the debugger.

The final library (**libdoyle**) is for distributed communications. This library is still under development. Its function is to provide a simple Tcl based programming interface between the application (in this case, **bdb**) and other systems on the network and it is integrated with the window system central loop.

Symbolic Library (libsym)

The purpose of **libsym** was to provide a standard interface for application programmers to use to access the symbol tables generated by our compilers and loader.

When **libsym** was developed, there were two main design goals in mind. The first was that the library should be at a high enough level to hide most of the symbolic formats used in the symbol tables. The second was that **libsym** should be at a low enough level as to not hide any of the information available from the symbol tables.

The abstraction that the library lays over the symbol tables involves breaking up the tables into sets cf modules (one module for the loader tables and one module for each compilation unit and common block). Within each module, the library associates a number of variable definitions, type definitions, line definitions, and scope definitions.

Structure pointers are used to pass information to and from the library. These structures also hold all "static" information needed about the symbol tables thus freeing the library from requiring any static storage. This allows the library to easily handle any number of symbol tables from any number of binaries, all within a single program.

When a user first initializes a binary's symbol tables, the library will return to the user the number of modules contained within the tables. The user can then query the library for information about each individual module by number. For example, a user could request information about module "1", module "2", up to module "n", the number of modules in the tables. The module information returned includes the number of variable, type, scope, and line definitions that are included within the module. From this, the user is able to get information about individual variables, types, scopes, or lines by number. For example, a user could request information about variable "1", variable "2", up to variable "n", (n being the number of variable definitions in the module).

As a short cut, much of the information from **libsym** could also be referenced by name as well as by index. For example, a user can ask for a variable by name, and have the library perform the name comparisons, returning a variable definition only if the name is found.

As our symbol table formats change (to eventually include symbolics for optimizations), the structures used to pass information between the user and **libsym** will grow to include the new information, but the base information will remain the same to provide backward compatibility for the library users.

Display Library (libdis)

The display library (**libdis**) was designed to produce all data displays needed by bdb. It can produce disassembly displays, dump displays, and symbolic displays.

To use **libdis**, the user selects the type of display needed (disassembly, dump, or symbolic), passes in a pointer to the data to be displayed, passes in a symbolic type definition of the data (generated by **libsym**) if a symbolic display is requested, and passes in any additional formatting overrides needed. **libdis** will then return to the user a char pointer that points to a formatted string.

Process Control Library (libbdb)

The process control library (**libbdb**) provides a simple interface to controlling a process and reading or writing a process' memory and register space.

Using **libbdb** a user can:

- Attach to a process
- Stop a process

- Restart a process
- Read or write its memory space
- Read or write its registers
- Send signals to the process
- Stop a process just prior to it receiving selected signals
- Stop the process on exec
- Stop the process on entry to a system call
- Stop the process on exit from a system call
- Read system level process structures

To perform any actions on a process, the user must first "open" the process via **libbdb**. Processes are identified using the system process id (pid) number. **libbdb** will allow any number of "opened" processes and the user identifies the process of interest via the pid number.

Once the user is finished with a process, the user then "closes" the process via **libbdb**.

User Interface Libraries

Although several libraries were developed to support the object style programming needed to support graphic user interfaces, **bdb** depends on one key library, **libtcl**.

libtcl was developed by Professor John Ousterhout of the University of California at Berkeley. Tcl (which stands for Tool Command Language) provides the low level interface to **bdb**. Tcl is, in many ways, a string manipulating programming language. It is an interpreted language that can be run by typing in individual Tcl commands at a terminal or can be run from files containing Tcl programs (we call them Tcl scripts).

There were many advantages to using Tcl as the low level interface to **bdb**. As a start, Tcl provided the following language features to the **bdb** interface:

- Global and local variables
- Arrays
- Callable procedures (with parameters)
- For loop constructs
- If-Then-Else constructs
- Formatted output
- User defined language extensions

Of the above features, bdb makes heavy use of the user define language extensions available in Tcl. The set of low level bdb commands are defined as extensions to the Tcl language. As new features and commands are added to bdb, they are in turn added to the bdb version of Tcl.

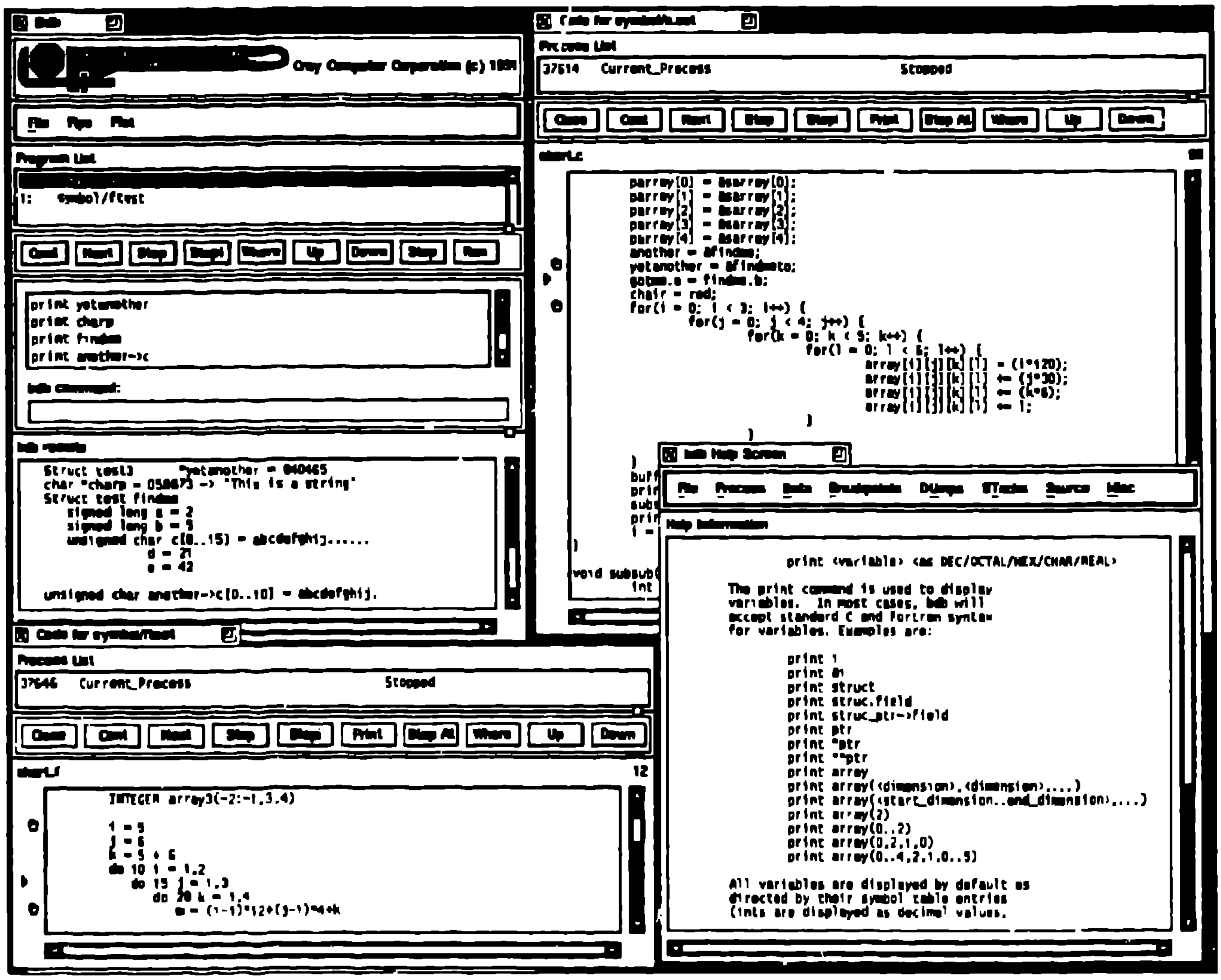
A Debugging Programming Language

In many ways, users can view bdb as a debugging programming language with which users can write interpreted programs. By creating the bdb extensions to the Tcl, the user is really dealing with a superset of the Tcl language. When the user is running bdb in line mode, the user is simply typing in Tcl commands to a Tcl interpreter which is started when bdb is invoked. The user could just as easily write Tcl scripts (programs) and invoke the scripts while running bdb to control the behavior of their debugging session.

To develop the window interface for bdb, we added a new library (**libwatson**) to bdb. **libwatson**, much like bdb, added several extensions to the Tcl language that would allow the user to create and manipulate several different types of windows. Once the **libwatson** extensions were added to Tcl, the window interface was created by writing a set of Tcl scripts that would create the windows and drive bdb.

libwatson was developed to support several different windowing look and feels that include Athena Widgets, OSF/Motif, and OPEN LOOK. The multiple window support was developed in such a manner as to make the look and feel styles transparent to bdb and the Tcl scripts that run it. The only changes needed in bdb to support OSF/Motif instead of Athena Widgets is to link in the X Window OSF/Motif library instead of the X Window Athena Widgets library. There are no code changes needed in either bdb or the Tcl scripts. All differences are hidden from bdb and Tcl by **libwatson**.

Figure 1 bdb OSF/MOTIF Windows


bdb
Cray Computer Corporation (c) 1991
File Rpe Plot
Program List
Cont Next Step Stepi Where Up Down Stop Run
bdb command:
bdb results
Code for symbol/ftest
Process List
Close Cont Next Step Stepi Print Stop At Where Up Down
shortf.f
12
Code for symbol/fs.out
Process List
shortl.c
bdb Help Screen
File Process Data Breakpoints DUmps STacks Source Misc
Help Information

```
1:   symbol/ftest
```

```
print yetanother
print charp
print findme
print another->c
```

```
Struct testl       *yetanother = 040465
char *charp = 058673 -> "This is a string"
Struct test findme
    signed long a = 2
    signed long b = 5
    unsigned char c[0..15] = abcdefghij......
              d = 21
              e = 42

unsigned char another->c[0..10] = abcdefghij.
```

```
37646   Current_Process                              Stopped
```

```
        INTEGER array3(-2:-1,3,4)

        i = 5
        j = 6
        k = 5 + 6
        do 10 i = 1,2
           do 15 j = 1,3
              do 20 k = 1,4
                 m = (i-1)*12+(j-1)*4+k
```

```
37614   Current_Process                              Stopped
```

```
        parray[0] = &sarray[0];
        parray[1] = &sarray[1];
        parray[2] = &sarray[2];
        parray[3] = &sarray[3];
        parray[4] = &sarray[4];
        another = &findme;
        yetanother = &findmeto;
        gotme.a = findme.b;
        chair = red;
        for(i = 0; i < 3; i++) {
                for(j = 0; j < 4; j++) {
                        for(k = 0; k < 5; k++) {
                                for(l = 0; l < 6; l++) {
                                        array[i][j][k][l] = (i*120);
                                        array[i][j][k][l] += (j*30);
                                        array[i][j][k][l] += (k*6);
                                        array[i][j][k][l] += l;
                                }
                        }
                }
        }
        buf
        pri
        sub
        pri
        i =
}

void subsub(
        int
```

```
          print <variable> <as DEC/OCTAL/HEX/CHAR/REAL>

     The print command is used to display
     variables.  In most cases, bdb will
     accept standard C and Fortran syntax
     for variables. Examples are:

          print i
          print an
          print struct
          print struc.field
          print struc_ptr->field
          print ptr
          print *ptr
          print **ptr
          print array
          print array(<dimension>,<dimension>,...)
          print array(<start_dimension..end_dimension>,...)
          print array(2)
          print array(0..2)
          print array(0,2,1,0)
          print array(0..4,2,1,0..5)

     All variables are displayed by default as
     directed by their symbol table entries
     (ints are displayed as decimal values.
```

Putting It all Together

In many ways, bdb is a driver program that drives the above mentioned libraries. To give an example of how the libraries work together, the following steps illustrate what bdb does when the user asks bdb to display the value of a variable.

- User types "print foo".
- The bdb Tcl interpreter evaluates the command and calls appropriate low level bdb routine, passing in the string "foo" as the parameter.
- bdb calls the symbolic library (**libsym**) to look up the variable named "foo".
- **libsym** returns the symbolic definition of foo along with its address and bit size.
- bdb calls the process control library (**libbdb**) requesting a read from the address returned by **libsym**.
- **libbdb** returns a data pointer that holds the value of "foo".
- bdb calls the display library (**libdis**) passing to it the symbolic definition returned by **libsym** and the data pointer returned by **libbdb**.
- **libdis** returns to bdb a formatted string that includes the symbolic definition and the formatted value of "foo".
- bdb returns the string to Tcl which in turn displays it to the user.

Other Tools

Many other tools at Cray Computer Corporation have been created or enhanced using the libraries initially targeted for bdb. These include:

- **stb** - A graphic symbol table browser.
- **dasm** - A symbolic disassembler.
- **sim** - A hardware simulator used to debug operating system code.
- **holmes** - A graphic Tcl environment.
- **nflow** - A graphic post processor for evaluating process flow and timing.
- **vsm** - Visual System Monitor used to monitor system operations and individual processes.
- **symdiag** - An internal symbol table diagnostic.
- **va** - Visual Administrator for system configuration.

Acknowledgments

The author wishes to acknowledge the help of a number of individuals at Cray Computer Corporation without whom bdb would still be an idea on my white board. These include Scott Bolte, Randy Murrish, Darragh Nagel, and Tom Engel.

A special thanks also goes to John Ousterhout from the University of California at Berkeley, the developer of Tcl.

References

John Ousterhout, "Tcl: An Embeddable Command Language," Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990. (1990A)

Information about Tool Command Language, along with the latest source code, may be obtained from John Ousterhout, University of California at Berkeley. A mailing list exists which is devoted to Tcl questions. To join, mail a request to *tcl-request@sprite.berkeley.edu* and ask to be included on the distribution.

Copyrights

Watson, Wiggins, Doyle, Holmes, stb, vsm, and bdb are trademarks of Cray Computer Corporation.

OSF/Motif is a trademark of the Open Software Foundation, Inc.

OPEN LOOK is a registered trademark of USL in the United States and other countries.

Tool Command Language (Tcl) was developed by Prof. John Ousterhout of the University of California at Berkeley.

Author Information

The author can be contacted by mail at Cray Computer Corporation, 1110 Bayfield Drive, Colorado Springs CO, 80906 or by e-mail at *bby@craycos.com*.

The Los Alamos Debugger

ldb

Jeffrey S. Brown
Los Alamos National Laboratory

January 31, 1992

Introduction:

The ability to debug and validate computer software is an essential part of the development process. This becomes increasingly difficult as the complexity of algorithms and supercomputer hardware 'ncrease, requiring more sophisticated tools. To address this need C Division embarked upon a project to write a new debugger for use on Los Alamos supercomputers.

The Los Alamos Debugger (ldb) is targeted to meet the needs of large scale scientific code developers. To that end, ldb supports debugging computationally intensive fortran programs on production supercomputers, with minimal impact upon performance, taking full advantage of advanced capabilities provided by the UNICOS operating system running on Cray Research hardware. To support emerging technology, such as the Connection Machine, ldb is designed to be portable to other supercomputer platforms .

Los Alamos has considerable experience enhancing and using debuggers on supercomputers. In order to leverage off this experience and provide a familiar user interface, ldb design philosophy and command syntax is based upon the ddt debugger which has been for many years the primary production debugger under the CTSS operating system on Cray supercomputers. The migration from CTSS to UNICOS provided an opportunity to develop a debugger which would exploit unix features and expand upon the ddt base by incorporating modern debugger technology. An advantage of this approach is that users migrating to UNICOS are immediately productive with ldb thereby accelerating the migration effort by providing a means to quickly isolate bugs arising from porting codes. Advanced debugger capabilities are introduced incrementally and in an upward compatible way on top of this familiar base.

Current Production Debugger:

Ldb is running in production on all Los Alamos Crays running UNICOS, and includes the traditional debugger functionality of the CTSS debugger ddt, such as the ability to control process execution by setting breakpoints in the user code, save and restore capability, symbolic access to process variables, macro capability including debugger variables and flow control of ldb commands, the ability to "patch" a process, and calculator style expression evaluation. Ldb expands upon this base in many areas.

A significant enhancement, is the ability to set more complex conditional breakpoints, while retaining the low overhead required to process the condition . A code running under debugger control can be instrumented to stop at a code location when a condition is met. Under ldb, this condition is processed in the user code by re-routing execution flow to evaluate the condition during execution. Traditional unix debuggers (dbx) evaluate the condition in the debugger which can cause a code to run 20,000 times slower and effectively eliminate the conditional breakpoint capability for computationally intensive codes. The ldb implementation supports much more complicated conditions than its predecessor, ddt.

A powerful new feature of ldb is software watchpoints. Watchpoints allow a user to stop his code when a particular condition is met regardless of the code location. This is an improvement upon conditional breakpoints which evaluate the condition only at a specific code location, but requires more overhead to process. The watchpoint capability can be of great use to the code developer in locating the root cause of data corruption

For users with the necessary terminal hardware and software, ldb provides a graphical user interface using X-windows which facilitates source-level debugging and provides a natural interface to debugger capabilities.

Ldb has the ability to pipe formatted data from the process under debugger control to another unix process. This opens the door for any type of data analysis to be done during the debugging session, such as visualization.

An interesting new capability which interfaces to ldb is auditory data analysis. The user pipes data from the process being debugged to a workstation with sound capabilities which "plays back" the data. The user can then "hear" his data recognizing anomalies over a large range of data. Visualization and traditional debugging analysis can then be used to home- in on the problem area.

Version 2.0, scheduled for release second quarter 1992, will include a first cut at debugging support for shared memory parallel codes.

Design Goals

Significant Features

Future Challenges

Design Goals

fortran

large-scale scientific codes

familiar look and feel

multi-level debugging

unobtrusive

minimize impact upon program performance

portable

flexible

facilitate migration from CTSS to UNICOS

all features available from "dumb" terminals

exploit unix features

Significant Features

output piping (visualization and sound)

software watchpoints

conditional breakpoints

execution tracing

multi-level debugging

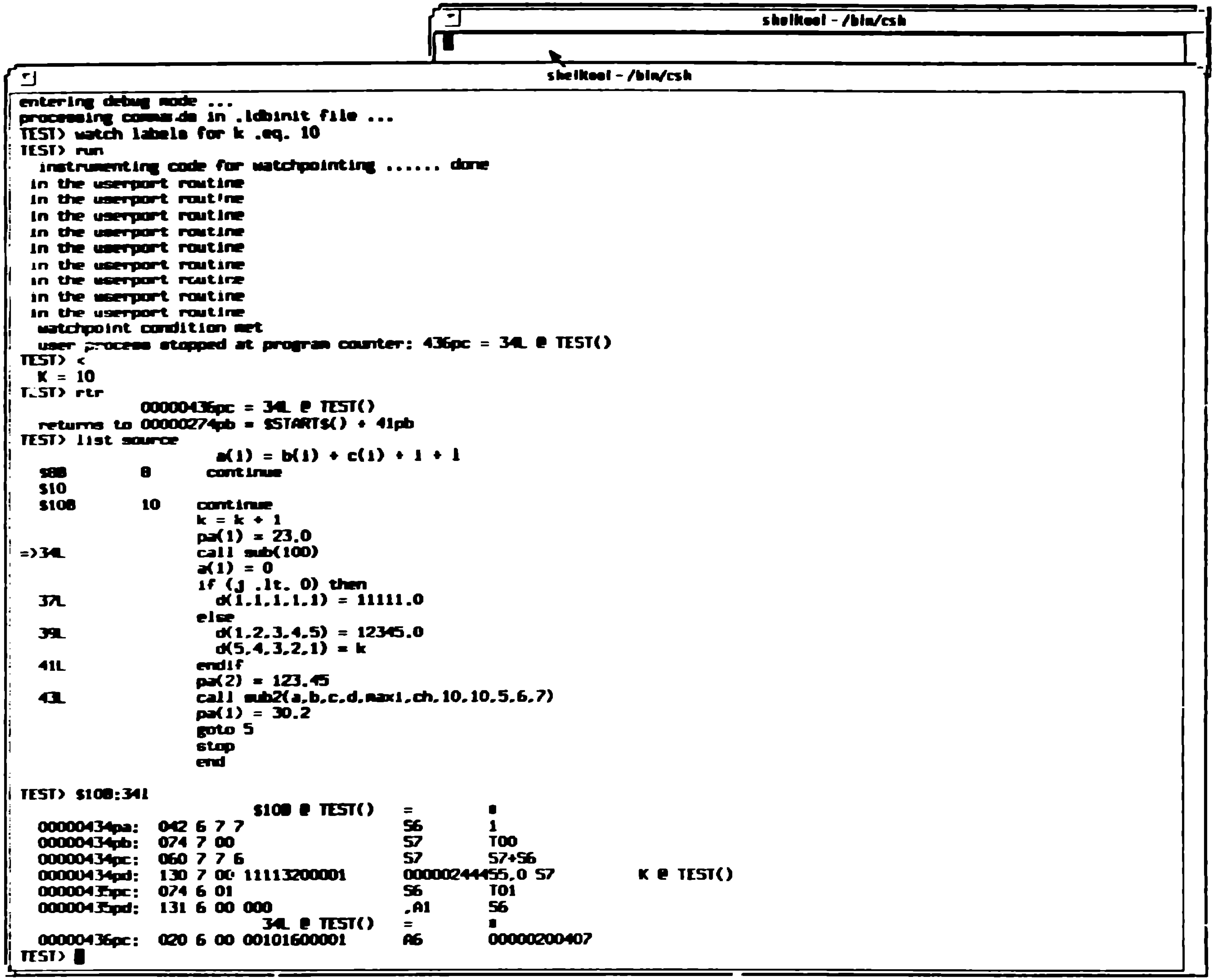
formatted prints

macros

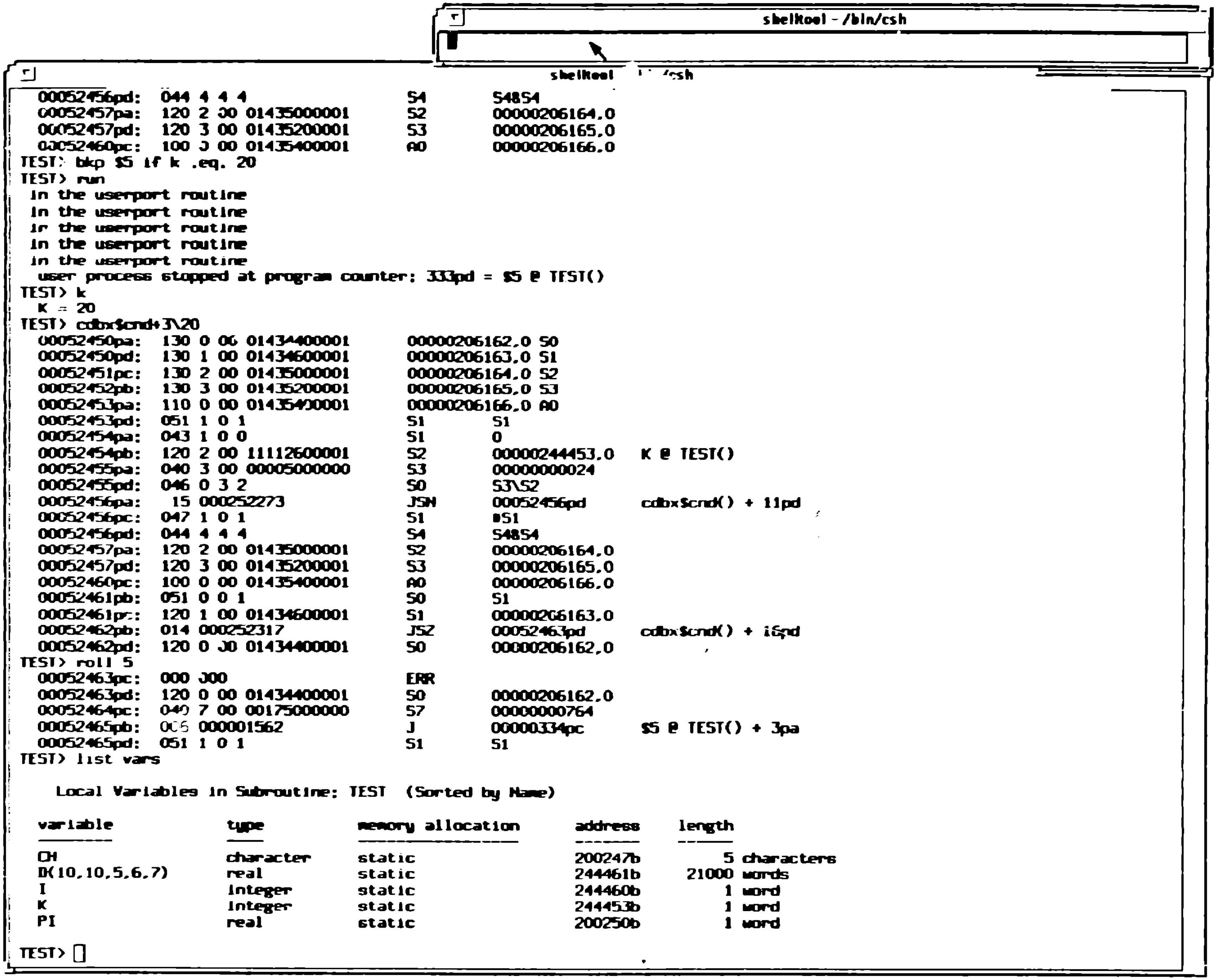
symbol table browsing

attaching to a running process

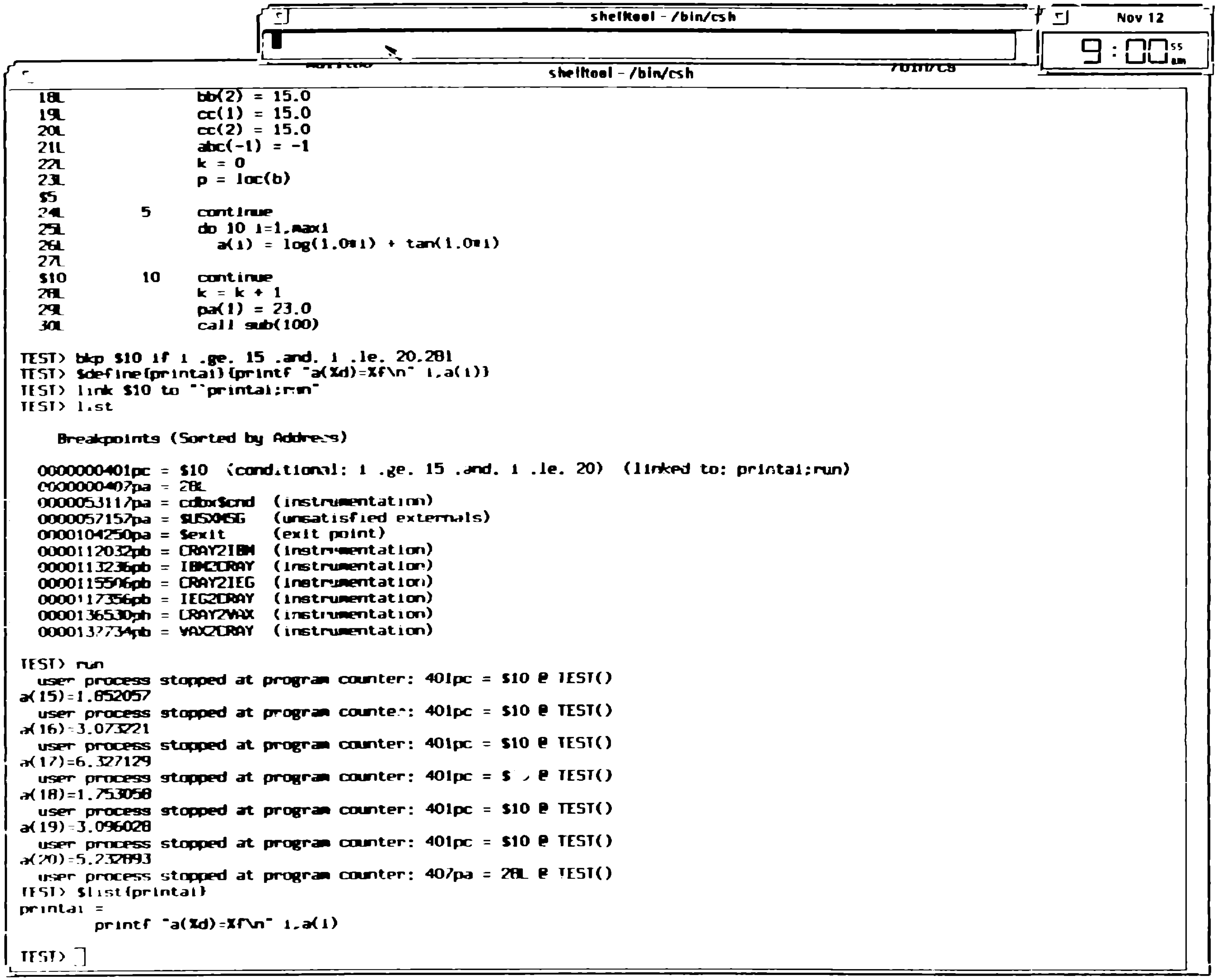
graphical user interface via x-windows



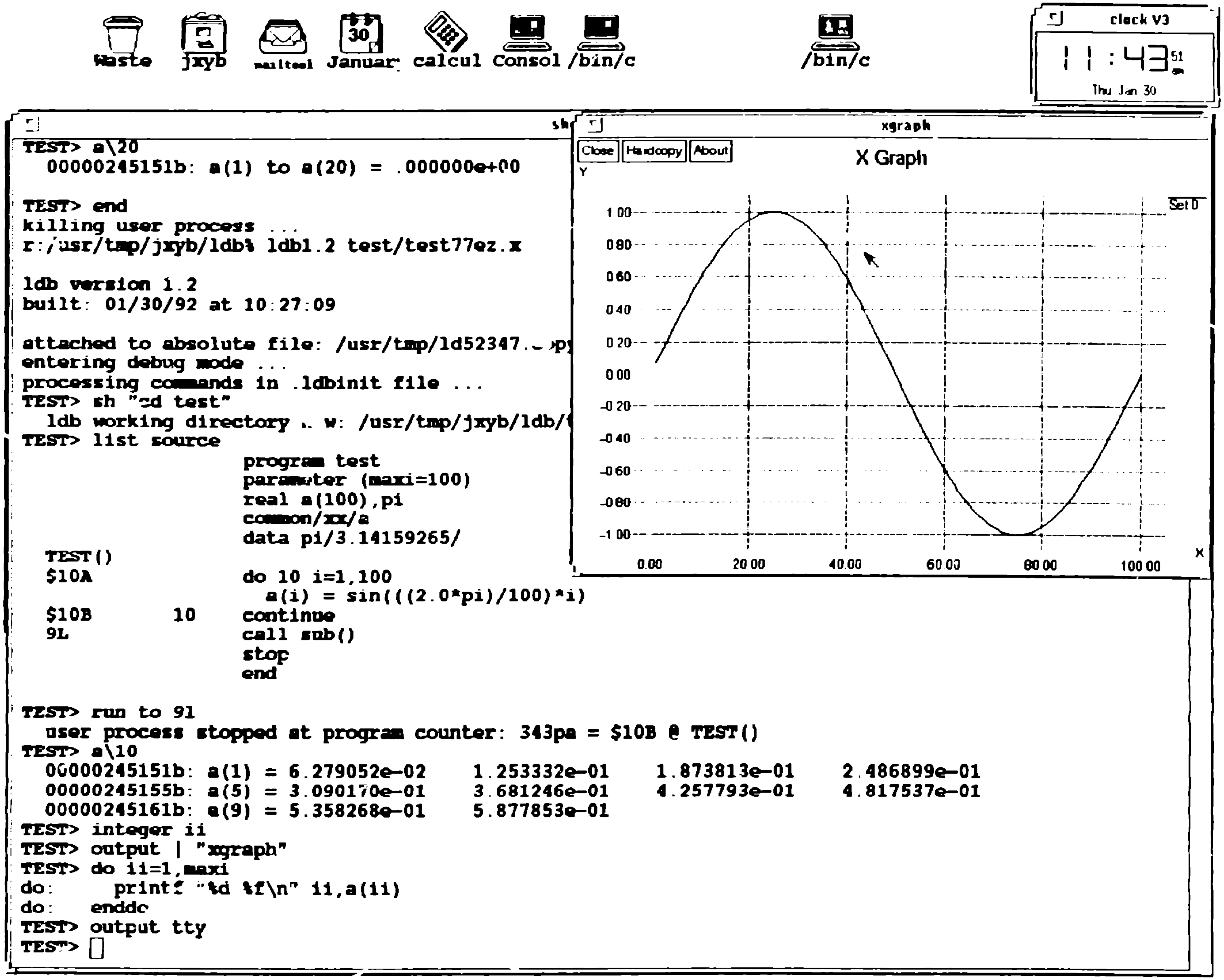
```
entering debug mode ...
processing commands in .ldbinit file ...
TEST> watch labels for k .eq. 10
TEST> run
  instrumenting code for watchpointing ...... done
 in the userport routine
 in the userport rout!ne
 In the userport routine
 in the userport routine
 In the userport routine
 in the userport routine
 in the userport rcutire
 in the userport routine
 in the userport routine
  watchpoint condition met
  user process stopped at program counter: 436pc = 34L @ TEST()
TEST> <
  K = 10
T.ST> rtr
             00000436pc = 34L @ TEST()
  returns to 00000274pb = $START$() + 41pb
TEST> list source
                     a(i) = b(i) + c(i) + i + 1
  $88        8        continue
  $10
  $108       10     continue
                    k = k + 1
                    pa(1) = 23.0
=>34L               call sub(100)
                    a(1) = 0
                    if (j .lt. 0) then
  37L                 d(1,1,1,1,1) = 11111.0
                    else
  39L                 d(1,2,3,4,5) = 12345.0
                      d(5,4,3,2,1) = k
  41L               endif
                    pa(2) = 123.45
  43L               call sub2(a,b,c,d,maxi,ch,10,10,5,6,7)
                    pa(1) = 30.2
                    goto 5
                    stop
                    end

TEST> $108:34l
                        $108 @ TEST()   =        ■
  00000434pa:  042 6 7 7                S6       1
  00000434pb:  074 7 00                 S7       T00
  00000434pc:  060 7 7 6                S7       S7+S6
  00000434pd:  130 7 00 11113200001     00000244455,0 S7        K @ TEST()
  00000435pc:  074 6 01                 S6       T01
  00000435pd:  131 6 00 000             ,A1      S6
                         34L @ TEST()   =        ■
  00000436pc:  020 6 00 00101600001     A6       00000200407
TEST> ▮
```



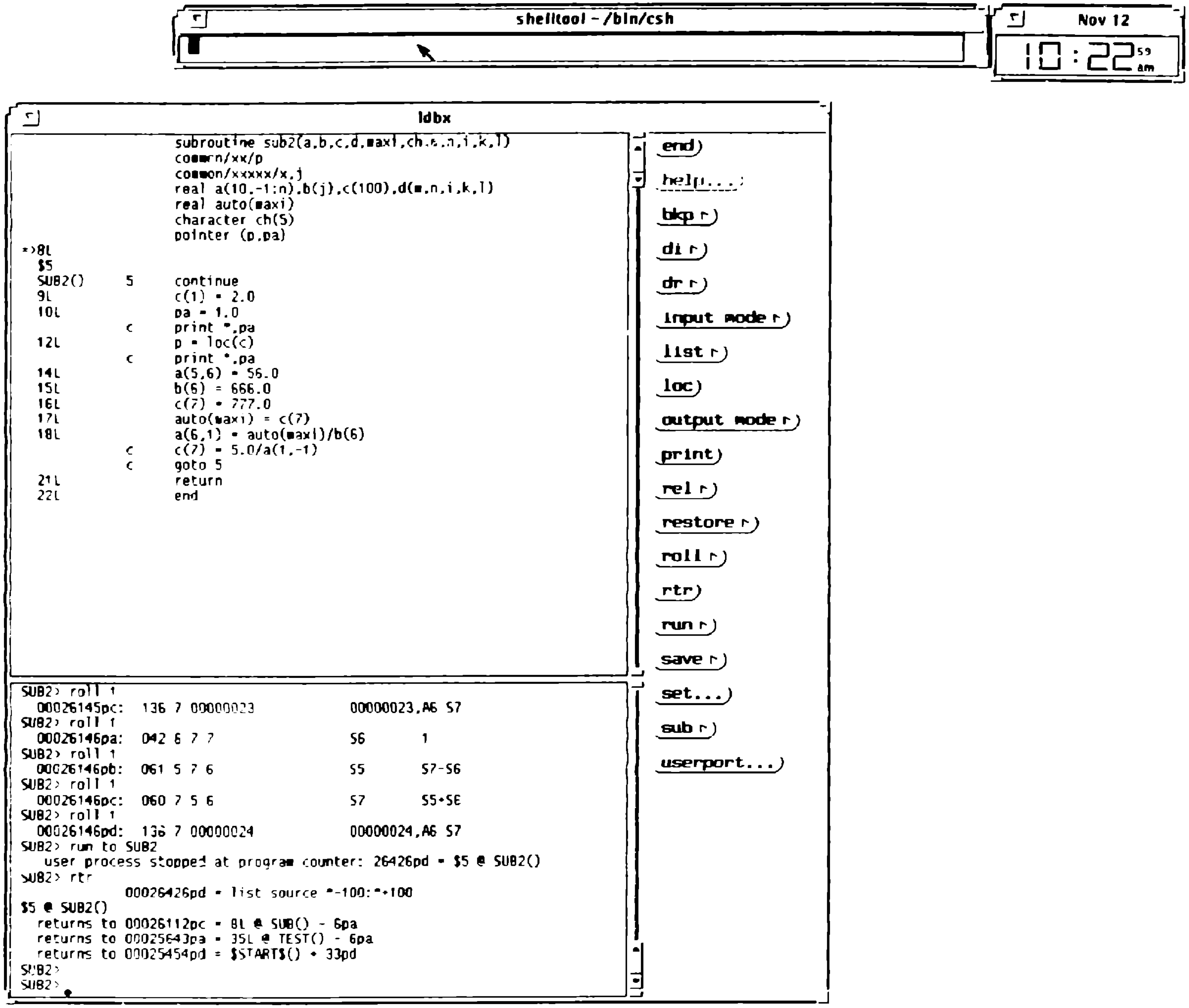
```
  00052456pd:  044 4 4 4                S4        S4&S4
  00052457pa:  120 2 00 01435000001     S2        00000206164.0
  00052457pd:  120 3 00 01435200001     S3        00000206165.0
  00052460pc:  100 0 00 01435400001     A0        00000206166.0
TEST> bkp $5 if k .eq. 20
TEST> run
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
  user process stopped at program counter: 333pd = $5 @ TEST()
TEST> k
  K = 20
TEST> cdbx$cmd+3\20
  00052450pa:  130 0 00 01434400001     00000206162.0 S0
  00052450pd:  130 1 00 01434600001     00000206163.0 S1
  00052451pc:  130 2 00 01435000001     00000206164.0 S2
  00052452pb:  130 3 00 01435200001     00000206165.0 S3
  00052453pa:  110 0 00 01435400001     00000206166.0 A0
  00052453pd:  051 1 0 1                S1        S1
  00052454pa:  043 1 0 0                S1        0
  00052454pb:  120 2 00 11112600001     S2        00000244453.0   K @ TEST()
  00052455pa:  040 3 00 00005000000     S3        00000000024
  00052455pd:  046 0 3 2                S0        S3\S2
  00052456pa:   15 000252273            JSN       00052456pd      cdbx$cmd() + 11pd
  00052456pc:  047 1 0 1                S1        #S1
  00052456pd:  044 4 4 4                S4        S4&S4
  00052457pa:  120 2 00 01435000001     S2        00000206164.0
  00052457pd:  120 3 00 01435200001     S3        00000206165.0
  00052460pc:  100 0 00 01435400001     A0        00000206166.0
  00052461pb:  051 0 0 1                S0        S1
  00052461pc:  120 1 00 01434600001     S1        00000206163.0
  00052462pb:  014 000252317            JSZ       00052463pd      cdbx$cmd() + 16pd
  00052462pd:  120 0 00 01434400001     S0        00000206162.0
TEST> roll 5
  00052463pc:  000 000                  ERR
  00052463pd:  120 0 00 01434400001     S0        00000206162.0
  00052464pc:  040 7 00 00175000000     S7        00000000764
  00052465pb:  006 000001562            J         00000334pc      $5 @ TEST() + 3pa
  00052465pd:  051 1 0 1                S1        S1
TEST> list vars

    Local Variables in Subroutine: TEST  (Sorted by Name)

  variable            type         memory allocation     address    length
  --------            ----         -----------------     -------    ------
  CH                  character    static                200247b        5 characters
  D(10,10,5,6,7)      real         static                244461b    21000 words
  I                   integer      static                244460b        1 word
  K                   integer      static                244453b        1 word
  PI                  real         static                200250b        1 word

TEST> []
```



```
  18L             bb(2) = 15.0
  19L             cc(1) = 15.0
  20L             cc(2) = 15.0
  21L             abc(-1) = -1
  22L             k = 0
  23L             p = loc(b)
  $5
  24L       5     continue
  25L             do 10 i=1,maxi
  26L               a(i) = log(1.0*i) + tan(1.0*i)
  27L
  $10      10     continue
  28L             k = k + 1
  29L             pa(1) = 23.0
  30L             call sub(100)

TEST> bkp $10 if i .ge. 15 .and. i .le. 20,28l
TEST> $define{printai}{printf "a(%d)=%f\n" i,a(i)}
TEST> link $10 to "`printai;run"
TEST> list

   Breakpoints (Sorted by Address)

  0000000401pc = $10  (conditional: i .ge. 15 .and. i .le. 20)  (linked to: printai;run)
  0000000407pa = 28L
  0000053117pa = cdbx$cmd  (instrumentation)
  0000057157pa = $USXMSG   (unsatisfied externals)
  0000104250pa = $exit     (exit point)
  0000112032pb = CRAY2IBM  (instrumentation)
  0000113236pb = IBM2CRAY  (instrumentation)
  0000115506pb = CRAY2IEG  (instrumentation)
  0000117356pb = IEG2CRAY  (instrumentation)
  0000136530pb = CRAY2VAX  (instrumentation)
  0000137734pb = VAX2CRAY  (instrumentation)

TEST> run
  user process stopped at program counter: 401pc = $10 @ TEST()
a(15)=1.852057
  user process stopped at program counter: 401pc = $10 @ TEST()
a(16)=3.073221
  user process stopped at program counter: 401pc = $10 @ TEST()
a(17)=6.327129
  user process stopped at program counter: 401pc = $10 @ TEST()
a(18)=1.753058
  user process stopped at program counter: 401pc = $10 @ TEST()
a(19)=3.096028
  user process stopped at program counter: 401pc = $10 @ TEST()
a(20)=5.232893
  user process stopped at program counter: 407pa = 28L @ TEST()
TEST> $list{printai}
printai =
        printf "a(%d)=%f\n" i,a(i)

TEST> ]
```

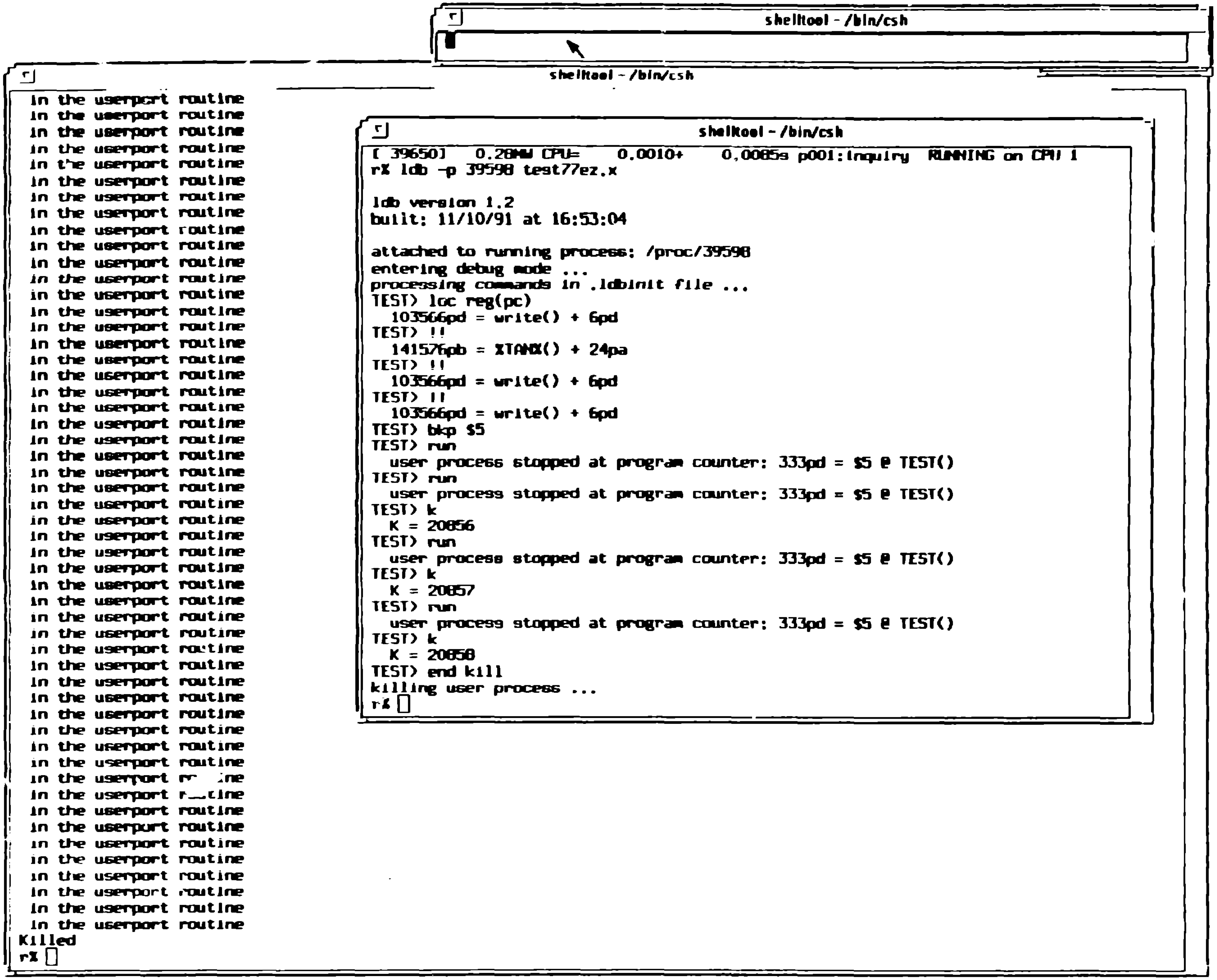


```
TEST> a\20
  00000245151b: a(1) to a(20) = .000000e+00

TEST> end
killing user process ...
r:/usr/tmp/jxyb/ldb% ldb1.2 test/test77ez.x

ldb version 1.2
built: 01/30/92 at 10:27:09

attached to absolute file: /usr/tmp/ld52347.‿py
entering debug mode ...
processing commands in .ldbinit file ...
TEST> sh "cd test"
  ldb working directory ‥ w: /usr/tmp/jxyb/ldb/t
TEST> list source
                   program test
                   parameter (maxi=100)
                   real a(100),pi
                   common/xx/a
                   data pi/3.14159265/
  TEST()
  $10A             do 10 i=1,100
                     a(i) = sin(((2.0*pi)/100)*i)
  $10B      10     continue
  9L               call sub()
                   stop
                   end

TEST> run to 9l
  user process stopped at program counter: 343pa = $10B @ TEST()
TEST> a\10
  00000245151b: a(1) = 6.279052e-02    1.253332e-01    1.873813e-01    2.486899e-01
  00000245155b: a(5) = 3.090170e-01    3.681246e-01    4.257793e-01    4.817537e-01
  00000245161b: a(9) = 5.358268e-01    5.877853e-01
TEST> integer ii
TEST> output | "xgraph"
TEST> do ii=1,maxi
do:     printf "%d %f\n" ii,a(ii)
do:   enddo
TEST> output tty
TEST> □
```



```
                    subroutine sub2(a,b,c,d,maxi,ch,m,n,i,k,l)
                    commrn/xx/p
                    common/xxxxx/x,j
                    real a(10,-1:n),b(j),c(100),d(m,n,i,k,l)
                    real auto(maxi)
                    character ch(5)
                    pointer (p,pa)
*>8L
  $5
  SUB2()      5     continue
  9L                c(1) = 2.0
  10L               pa = 1.0
              c     print *,pa
  12L               p = loc(c)
              c     print *,pa
  14L               a(5,6) = 56.0
  15L               b(6) = 666.0
  16L               c(7) = 777.0
  17L               auto(maxi) = c(7)
  18L               a(6,1) = auto(maxi)/b(6)
              c     c(7) = 5.0/a(1,-1)
              c     goto 5
  21L               return
  22L               end
```

```
SUB2> roll 1
  00026145pc:  136 7 00000023            00000023,A6 S7
SUB2> roll 1
  00026146pa:  042 6 7 7                 S6        1
SUB2> roll 1
  00026146pb:  061 5 7 6                 S5        S7-S6
SUB2> roll 1
  00026146pc:  060 7 5 6                 S7        S5+SE
SUB2> roll 1
  00026146pd:  136 7 00000024            00000024,A6 S7
SUB2> run to SUB2
   user process stopped at program counter: 26426pd = $5 @ SUB2()
SUB2> rtr
            00026426pd = list source *-100:*+100
$5 @ SUB2()
  returns to 00026112pc = 8L @ SUB() - 6pa
  returns to 00025643pa = 35L @ TEST() - 6pa
  returns to 00025454pd = $START$() + 33pd
SUB2>
SUB2>
```



```
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
 in the userport routine
Killed
r% 
```

```
[ 39650]   0.28MW CPU=    0.0010+    0.0085s p001:inquiry  RUNNING on CPU 1
r% ldb -p 39598 test77ez.x

ldb version 1.2
built: 11/10/91 at 16:53:04

attached to running process: /proc/39598
entering debug mode ...
processing commands in .ldbinit file ...
TEST> loc reg(pc)
  103566pd = write() + 6pd
TEST> !!
  141576pb = %TANX() + 24pa
TEST> !!
  103566pd = write() + 6pd
TEST> !!
  103566pd = write() + 6pd
TEST> bkp $5
TEST> run
  user process stopped at program counter: 333pd = $5 @ TEST()
TEST> run
  user process stopped at program counter: 333pd = $5 @ TEST()
TEST> k
  K = 20856
TEST> run
  user process stopped at program counter: 333pd = $5 @ TEST()
TEST> k
  K = 20857
TEST> run
  user process stopped at program counter: 333pd = $5 @ TEST()
TEST> k
  K = 20858
TEST> end kill
killing user process ...
r% 
```

Future Directions

- port to other unix platforms
- Connection Machine support
- component of decoupled I/O
- research into debugging parallel codes
- distributed debugging
- support for other languages
- support for other user interfaces
- debugging optimized code

The Auditorialization of A Running Code

Cheryl L. Wampler and Robert S. Hotchkiss
Los Alamos National Laboratory

Abstract

Investigations of the use of sound as a medium of feedback from an executing code have been initiated. The primitive variables that are the foundation of sound can be used to encrypt a large amount of information. We have found that sound can be used to successfully convey simple functions and data. It is possible to use sound for code debugging and for transmitting the output of code. We present the sound of an executing code, the sound of errors, and the sounds of arrays.

Introduction

The purpose of this research is to broaden our approach to computer programming by incorporating the use of sound as another avenue of feedback from machine to man. Until now, this feedback has generally taken the form of written messages, usually cryptic, and often misleading. Code debugging has traditionally been performed by stopping a running code at given breakpoints, and inspecting the status of the variables on a line by line basis. While this method has proven very useful, it is often difficult to localize subtle errors in huge codes. We propose that sound can provide a natural medium for the offloading of data from visual channels when needed, and can at times convey an understanding of processes that cannot be easily gained through visualization. Furthermore, the simultaneous perception of visual and auditory information can have a synergistic effect of the brain's ability to comprehend an event.

Our research has had the following immediate objectives:

1. to demonstrate, through simple and understandable means, the viability of using sound as a medium for feedback from a running code or from the data being manipulated in the code;

2. to develop basic software for translating data into sound;

3. to simultaneously lay the foundation for a more extensive user interface in the supercomputing

environment, and a functional interface into a supercomputing debugger;

4. to investigate the relations between various musical structures and typical code structures;

5. to couple visual and auditory communications from machine to man.

The Parameters of Sound

Because of the variety of parameters inherent in sound, it can easily lend itself to multidimensionality and thus provides a good medium for data interpretation. The most fundamental sound parameters that can be used as precise variables are frequency, amplitude, and time. There are other variables that span the domain of sound, such as timbre or tone quality, reverberation, brightness, etc. We have chosen to defer investigating some sound parameters, such as reverberation or brightness, because of the ambiguity in the terminology and our inability to aurally distinguish them with precision. There are an infinite number of ways in which the sound parameters could be applied to the variables of an event. The challenge for incorporating them into computer code that approaches being flexible enough to accomodate this great variety is the logic that must be written into the software.

Software

Our purpose for developing the software involved in this project was threefold:

1) to test the viability of manipulating the fundamental parameters of sound to represent data,

2) to provide groundwork for a more extensive, user-interactive audio/visual system for the supercomputing environment, and

3) to provide a useable sound interface into a mainframe debugger.

We have developed one basic set of sound software using a portable C compiler. For this phase of the project, a Yamaha SY77 electronic synthesizer and a NeXT with a DSP were utilized for the actual sound synthesis. The NeXT MusicKit software provided the conversion of sound-producing instructions into MIDI format or into a format acceptable to the DSP. A small library of functions have been developed to operate at execute time upon virtually arbitrary collections of data. This enables one to perform operations on scientific data, graphical data, or sound data through an interface as the code executes. This "evaluator" is capable of the normal mathematical library functions similar to a scientific calculator but allows a vector type (i.e., vector addition, multiplication, summing a vector and scalar, etc.). It is capable of permitting the user to write and invoke functions at execute time. It also allows the invocation of compiled functions as desired. The evaluator is dictionary driven to provide a flexible communication link between the user and itself for a specification of the means by which one intends to use the compiled primitive functions. Thus, sets of numerical data can either be generated within the program, or read in from outside files, manipulated by means of vector processes in a variety of ways, and converted into sound according to the user's specifications.

There will also be an option to run the code in a mode which will map a single set of data into frequency, setting default values for all other sound parameters. This mode will be used to provide quick sound "snapshots" of data being produced by processes such as, for example, an executing debugger.

We have discovered that audiblized data can create sounds that no man has ever heard before and truly excites even the non-musical mind. Humans have been so accustomed to looking at graphs of functions that a great richness of understanding has been sorely overlooked.

Implementing Sound In Supercomputing Environments

Supercomputers in the Cray XMP/YMP class of vector processors and massively parallel processors, such as Thinking Machines Inc. CM-2, are the only current types of machines capable of handling the transmission rates discussed above. If graphics and sonics are to occur simultaneously, a substantial engine is obviously necessary. Just to display and auralize data that have been stored as files requires very high-speed processing and disks capable of reading data at rates of one gigabit per second or greater. It is possible to transfer information at gigabaud rates from the memory of these machines. For sound, it is possible to compute sound files for many applications at the rates needed for three-dimensional auralization. That is, if one does not have the availability of DSP processors or music synthesizers, sound must be produced by computing the 44,100 16-bit samples needed to feed to an amplifier just as a compact disk does. We have developed software in the portable C language that is machine independent for a wide variety of machines. It does not port to the CM-2 directly. We have written code to compute the digital sound data from Fourier Series. This is a rather slow process on the workstation class of machines and a much more rapid vector process on Crays. The algorithms have been designed so they will map to the CM-2's SIMD (Single Instruction Multiple Data) architecture. At this early stage of development, we have not sought to specialize our efforts to any specific machine nor are we likely to. To maintain extreme portability is and will continue to be an underlying basis of this research.

Thus, one code generates sound formatted data on any machine. It vectorizes when compiled on a Cray thereby producing enormous speedups over workstations. We are taking the same approach with regard to data that will be sent to either DSPs or MIDI music synthesizers.

As a result of these approaches, we find that one must rely on the speed of supercomputers at times, but is also able to do most the development as well as testing on nearly any workstation.

Sound and Code Debugging

We believe that the use of sound has important implications in code debugging. In several instances, subtle errors in the code written for this project, although not obvious to the brain on paper or on screen, became immediately evident when the results were audibly played. Along these same lines, Nicolas Metropolis, pioneer of modern computing, reports that antennae connected to the voltage lines of Maniac electronic computers used at the Los Alamos National Laboratory in the 1950's provided acoustic feedback from the code as it was running. Metropolis notes that each code produced a particular cadence, from which an error or failure could be detected by a breakdown in the acoustic cadence.[1] On one particular occasion, the exact location of the error in the flow diagram was pinpointed by noted mathematician Robert Richtmyer on the basis of the sonic feedback. Current debuggers operate by allowing the user to stop the code while it is running and then to inspect its status on a line by line basis. In a very large code, this type of debugging can not only be tedious and time-consuming, but can at times be misleading in pointing to the real problem source. Giving a sound to a code as it is running can give a broader picture of the processes as they occur, and thus could help to localize subtle errors.

At Los Alamos, sound capability has been implemented with a mainframe code debugger. The current use of this sound in debugging is based on the breakpoint concept. At the traditional breakpoints, the contents of arrays can either be sent to a graphics package or to the sound package to be given a perceptual realization.

Our objective is to give a sound representation not only of variables, but also of the actual execution of any portion of the code upon demand of the user. In order to demonstrate the viability of this idea, a sound representation was given to an executing sort code. The sorting function, essentially a quicksort algorithm, calls itself recursively with successively smaller portions of the original array of data. The level of recursion and the entry and exit times are recorded for each call to the sorting function. The times are then mapped into an audible range, and a tone assigned to each level, to be held for the duration of that call to the function. In addition, the data, an array of integers, is mapped onto the pitches of a string of notes (within a two octave range). The state of the data for each call to the sorting function is played upon entry into, and before exiting out of that particular call to the function. In this manner, it is possible to hear the progression of the code through the various levels of recursion, and also to hear the progressive effects of the algorithm on the data. This type of sonic feedback can thus be used to represent the progression of the code itself or the manipulations being performed upon the data (or both). Sonic feedback can also be used to portray the state and/or changes of state of memory. We speculate that these uses of sound, as well as others yet unthought of, will provide valuable additions to the current arsenal of tools used to locate subtle and hard-to-find "bugs" in large codes.

Conclusion

As a result of our research to date, we predict that sound will be a viable medium for feedback from the machine to man, that multidimensionality can be clearly delineated with sound and thus that sound has great potential for the representation of multidimensional data, and that sound will provide an important additional tool for code debugging and for understanding the processes of a running code. We would also conclude that further research is necessary concerning the use of sound in computer debugging.

REFERENCES

[1] Nicolas Metropolis, private communication, Los Alamos, N.M., April 4, 1991.

THE AUDITORIALIZATION OF SCIENTIFIC INFORMATION

Robert Hotchkiss, X-7

(Project Leader)

Cheryl Wampler, C-10

(University of New Mexico, Los Alamos)

Greg Oakes, X-7

Michigan State University) - (Summer Student)

This project is being jointly conducted by the C-10 (User Services) and X-7 (Computational Physics) groups.

Los Alamos National Laboratory

PURPOSE OF THIS PROJECT ?

- TO AUDIFY SCIENTIFIC INFORMATION
- TO GIVE MACHINES A VOICE TO COMMUNICATE
- TO SYNERGIZE THE AUDIO-VISUAL
- TO DEVELOP CODE FOR AUDIO-VISUALIZATION
- TO INCREASE THE BANDWIDTH TO THE BRAIN
- INCREASE UNDERSTANDING

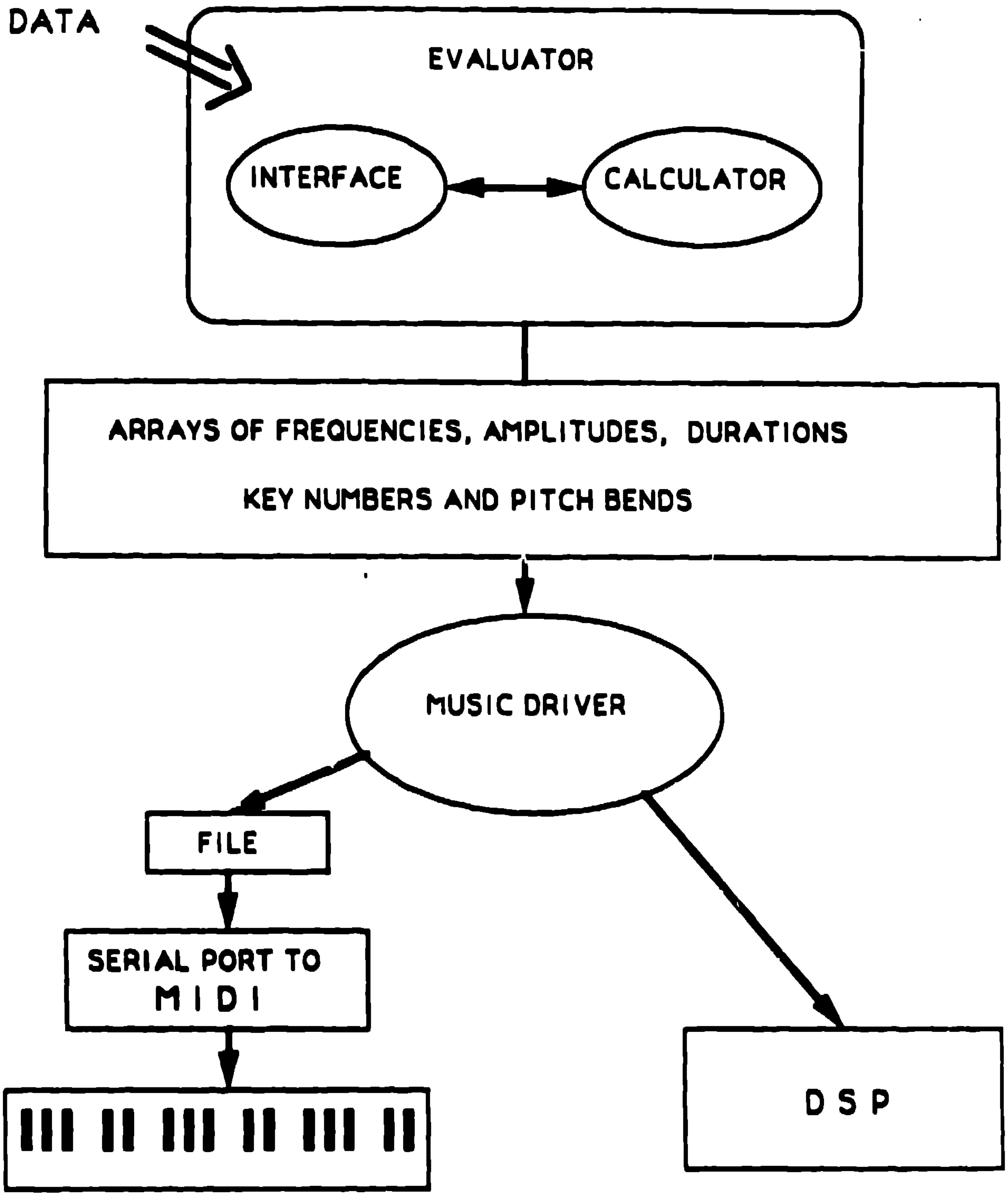


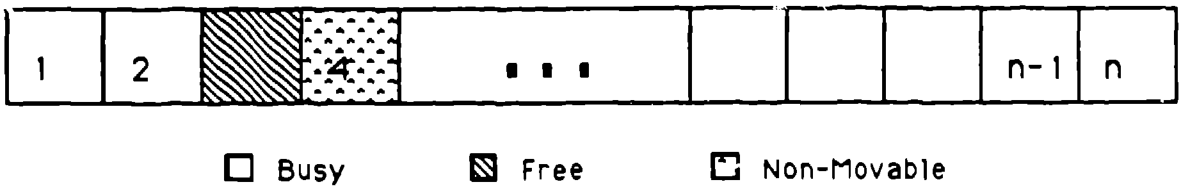
Fig 1 Sound production using "Evaluator" from DATA input to sound

PARAMETERS OF SOUND

- Frequency
- Amplitude
- Timing
- Duration
- Attack - Decay
- Timbre
- Reverberation
- Spatial Position

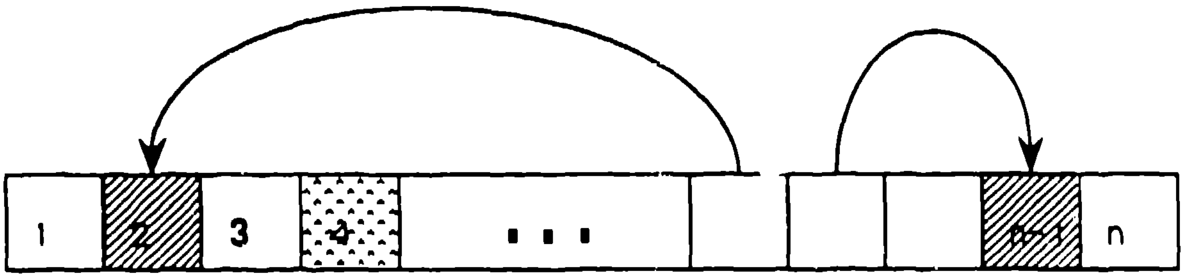
Los Alamos
COMPUTATIONAL PHYSICS

Dynamic Memory Manager Application



Partitioning of a Memory Pool

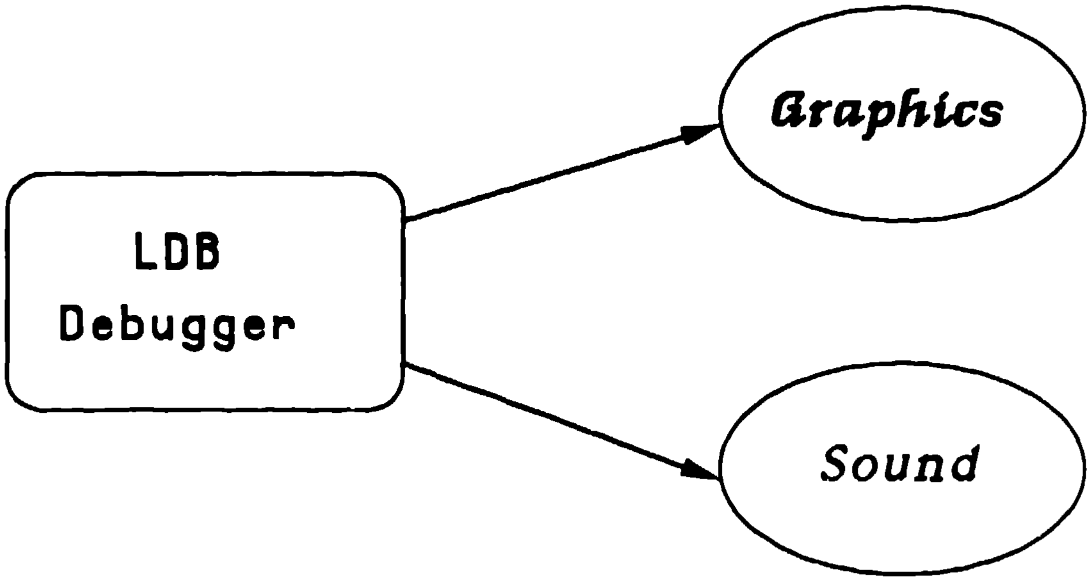
Sound can be used to study the operation of a memory management system. By mapping the spacing of the allocated blocks into a frequency (pitch) range with the lowest and highest pitches representing the boundaries of the pool, the current status of the memory pool can be aurally scanned from bottom to top. A special sound, such as a bell or a drum roll can signal the location of free or non-movable spaces.



Block Movement to Create a Larger Free Block

Sometimes, several busy blocks may be reallocated to new memory locations to make room for the allocation of a block requiring a larger memory space. In the event of a failure in the memory manager, it can prove difficult to trace what went wrong, and to where the blocks have been moved. The repositioning of these blocks inside, or outside, of the memory pool can be traced with sound.

Code Debugging with Sound



At Los Alamos, sound capability has been implemented with a mainframe code debugger. At the traditional breakpoints, the contents of arrays can be sent to either a graphics package or to the sound package to be given a perceptual realization.

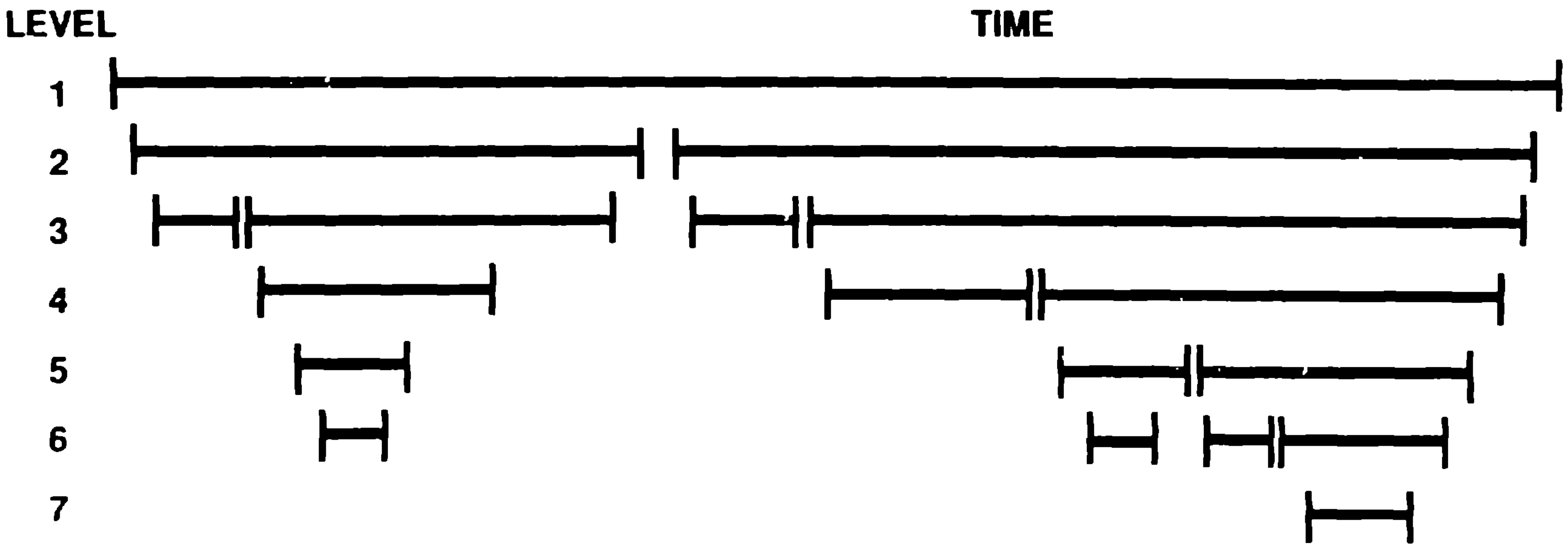
Our objective is to give a sound representation not only of static variables, but also of the actual execution of any portion of the code upon demand of the user. This representation could reveal the progression of the code itself, or the manipulations being performed upon the data (or both). Thus, it will be possible to hear an error as it actually occurs in the executing code. We speculate that this will be a valuable tool in locating subtle and hard-to-find "bugs" in large codes.

```
VOID QUICKSORT( DATATYPE V, INT LEFT, INT RIGHT, INT LEVEL )
{
   REGISTER INT I, J ;
   DATATYPE X, Y;
   VOID SWAP() ;

   ++LEVEL ;
   I = LEFT ;
   J = RIGHT ;
   X = $V_{(LEFT+RIGHT)/2}$ ;

   DO {
      WHILE( $V_I$ < X && I < RIGHT ) I++ ;
      WHILE( X < $V_J$ && J > LEFT  ) J-- ;

      IF( I <= J ) {
         SWAP( V, I, J ) ;
         I++ ; J-- ;
      }
   } WHILE( I <= J ) ;

   IF( LEFT < J  ) QUICKSORT( V, LEFT, J,  LEVEL ) ;
   IF( I < RIGHT  ) QUICKSORT( V, I, RIGHT, LEVEL ) ;
   --LEVEL ;
}
```

Los Alamos
COMPUTATIONAL PHYSICS

Sound Rendering of an Executing Code

A sound representation was given to an executing sort code. The sorting function, essentially a quicksort algorithm, calls itself recursively with successively smaller portions of the original array of data. The level of recursion and the entry and exit times are recorded for each call to the sorting function. The times are then mapped into an audible range, and a tone assigned to each level, to be held for the duration of that call to the function. In addition, the data, an array of integers, is mapped onto the pitches of a string of notes (within a two octave range). The state of the data for each call to the sorting function is played upon entry into, and before exiting out of the function. In this manner, it is possible to hear the progression of the code through the various levels of recursion, and also to hear the progressive effects of the algorithm on the data.



An Integrated Debugging, Analysis, and Visualization Environment for Large-Scale Multiprocessors[1]

Alva L. Couch
David W. Krumme
Tufts University

Abstract

The tasks of debugging and performance analysis have traditionally been separate pursuits, one undertaken at the beginning of development and the other viewed as a tuning phase after basic algorithmic function has been verified. Our experience indicates, however, that analysis and visualization techniques are essential for assuring even basic function for complex algorithms on massively parallel machines. A proper debugging environment for these machines should have scalable levels of user interaction and data portrayal: it should allow the user to function on any level, from traditional single-processor queries to global visualizations of performance based on user-specified probe requests.

We are currently implementing a prototype environment allowing integrated debugging, event trace analysis, and real-time performance visualization for the NCUBE/2. This paper describes the details and motivations behind the prototype design. Present and proposed capabilities are discussed, including a graph-oriented editor for visualization environments, the integration of debugging into this editor, and how other data sources (such as event traces) complement and enhance traditional debugging techniques in this environment. We also discuss the future of the method, a generalization of the data flow method based upon a graphical query language for the execution data viewed as a relational database.

1 Introduction

Unlike most of the other talks at the workshop, we consider the problem of debugging programs on massively parallel MIMD computers. A debugger for these architectures faces special problems. First, the 'window-per-process' approach used in most parallel debuggers fails completely, as the user would have to manipulate 1000 windows. One needs a histogram beside the source code just

[1] This is a very rough draft of a paper to be submitted to the Journal of Parallel and Distributed Computing special issue on visualization, to be published in 1993. Comments and suggestions are welcome; please send them to couch@cs.tufts.edu. Apologies in advance to anyone whose work was unfairly left unreferenced.

to represent the distribution of individual program counters. More important, printing a variable or value is not possible in the normal sense. One cannot efficiently interpret the results of 'print x' when there are 1000 x's. Likewise, distributed arrays are difficult to print and read, both due to their size and the variety of array allocations used.

In this paper, we attack the problem of printing a variable x, by exhibiting a way to integrate visualization and debugging environments. Unlike debugging, visualization has always dealt with reducing massive volumes of information into a compact, easily viewable form. Using visualization, we reduce the data to graphical attributes and render all values of x in a perusable form. Then, if the visualization system allows selective re-rendering of subsets of data, we can freely navigate the data until we find the information we need.

We come to this problem from the visualization perspective. Our first tool, Seecube[2], was a trace-driven post-mortem execution analyzer. Event traces collected on all processors were sorted into a global execution order and execution parameters and states were simulated from the trace. Traces were recorded for communication events by enveloping communication system calls within data recording versions. We later developed Seeplex[3], a real-time version of Seecube taking its data directly from a custom operating system for the processing nodes. Current work involves extending Seeplex to handle debugging as well as visualization.

Others have also developed related visualization environments, though most are aimed at comparing architectures rather than debugging programs on a single architecture. Paragraph[6] is a portable re-implementation of Seecube in X-windows, based on a similar collection of event traces using the PICL[5] portable communication library. Paragraph adds many new displays to the original suite provided by Seecube, but very few of the displays are scalable and none allow interrogation of actual values from the displays. Pablo[10] is a visualization system for any kind of event trace, using a graph-based user interface inspired by that of Seeplex. The interface uses object binding semantics and is thus harder to use, but its portable event format allows one to save the results of filtration in a file for later reprocessing. Pablo's base assumption is that visualization is a process which can ignore the semantics of the data being studied. Again, this is an assumption which works when one is comparing architectures, but not when one is trying to debug on a particular one.

From the debugging perspective, the most interesting debugger is the Prism system for the Connection machine. It allows a very limited though quite useful form of visualization: a distributed array can be perused slice by slice in several dimensions, either in textual form or as a graphical rendition. In graphical mode, each pixel is colored according to the value of a particular array cell, and a pixel can be queried for its value using a mouse graphical input device. Many debuggers are now providing interfaces to X windows renderers such as xgraph, and several are now providing interfaces to the scientific data visualization system AVS. These interfaces are not just useless bells and whistles: Lapolla[8] has developed a nice set of techniques for designing debugging experiments using data visualization. The most interesting of these is the concept of *injection*, in which the input data to the algorithm is modified to cause a particular pattern in the output data visualization.

This paper is divided into five parts. The second part identifies our integration strategy, the use of a common data format. The third part discusses how to convert event traces into this format, and describes a portable, self-defining format for conversion specifications[2]. The fourth part discusses details of the data flow visualization system which uses this format. The fifth part discusses the design of the next generation visualization system, which uses relational database query semantics instead of data flow.

2 The integration problem

Integration of debugging, performance analysis, and performance visualization is made difficult by the large number of subsystems that must be combined. We use the visualization system as the central core into which these parts are merged, unlike existing approaches which tack a visualization system onto an existing debugger as an afterthought.

Each major subsystem is managed from the visualization system while having essentially unchallenged control of some facet of execution monitoring. These subsystems ultimately produce streams of data which the visualization system filters and renders. At the top level, the user activates the subsystems and manages the streams of data by using a graphical editor to specify interrelationships between the subsystems and displays.

One subsystem is a conventional serial debugger that allows the user to interactively control the execution of a program, insert breakpoints, and to probe for values. It also provides access to the source code text.

Another subsystem deals with event logging in an executing program. The user can enable and disable event logging of various kinds, and this subsystem fetches the resultant event traces for use in the rest of the system or for archiving in files.

A similar subsystem manages the real-time data collection instrumentation. The user can configure this instrumentation to monitor state information describing communication activity, process states, the contents of program variables, and the like.

Yet another subsystem manages post mortem analysis of event traces stored in files. It has control features for stepping forward and backward at various speeds and for jumping to arbitrary points in a trace.

All of these subsysteins are largely independent of the visualization and analysis software. We are in the process of developing specifications for the interfaces

[2] This section, though not presented in my talk, is important to current discussions of portable event trace formats

between the components so that alternate versions of these subsystems can be substituted for new computer architectures. In this way we hope to achieve a useful degree of portability. To port the software to a different architecture, compiler, operating system, and instrumentation package, one would only need to rewrite these subsystems in accordance with the interface specifications.

2.1 Levels of debugging

In our debugging practice, there are essentially three kinds of bugs[3]. *Local* bugs include anything which can be seen to be incorrect in the context of a single isolated process, such as typing errors, loop limit problems, array boundary problems, or other algorithmic misunderstandings. These bugs may typically be studied by executing the program on a small architecture of one or two processors using traditional methods. *Global* bugs include any incorrect assumption about the relationships between processors, such as race conditions and deadlocks. These bugs are typically found by utilizing an event trace, either through direct perusal or an 'instant replay' interface for removing nondeterminism[9]. *Performance* bugs include any incorrect prediction about the efficiency of a correctly functioning program. These bugs are typically studied through profiling, though for massively parallel machines visualization is usually necessary to render an animated representation of execution. Most performance visualization systems utilize an event trace, and can be considered as higher-level versions of the trace-driven debugging used for races and deadlocks.

Our goal is to unify the levels, by designing a data representation common to all levels and providing an environment in which that representation can be freely manipulated and rendered. Any common representation must be complex enough to embody the data from each source, while being simple enough to render and understand easily. We choose the state vector as the common representation. Obviously, variable values and real-time statistics may be interpreted as state variables. Events are converted to state variables by analyzing the effects of each event on global state, so that the effect of one event may be the change in many state variables. For example, the occurrence of the one event '16-byte message received on processor 9 from processor 13' might change the values of the following state variables:

1. number of messages sent from 9.

2. total bytes sent by 9.

3. size of last message from 9.

[3] We kindly refrain in this paper from discussing our definition of a bug, which has caused some consternation among colleagues; however, we feel it somewhat illuminating to mention our definition here. A bug is a deviation between actual and expected performance, i.e., a deviation between actual performance and the programmer's internal model of execution. A bug thus lives not in the program, but in the programmer's model: bugs are not found in the code, but in the comments!

4. number of messages received on 13.

5. total bytes received on 13.

6. size of last message to 13.

7. the state of each channel connecting 9 and 13.

8. number of events processed (an artificial clock).

9. and several other states more dependent on the specific architecture.

Obviously, an event is completely represented in the state space only if enough states variables are defined to completely encode the event's contents.

This encoding has advantages and disadvantages. An event is hard to render, because it inherently has no duration in time. Showing an event in an animation is inherently misleading, as the rendered representation must have a duration to be perceived, while the event does not. State information is much easier to render naturally, as a state inherently has a duration. This makes time lines and snapshots easier to construct and interpret. However, the choice of states certainly affects how much information is gleaned from the trace, and a bad choice can lead to a state vector containing the wrong information. It may also be important for the user to see the real trace unmodified. This is easy, but nevertheless increases the complexity of the analysis tool as well as the number of visual formats the user must learn.

2.2 A 'note' on terminology

Any instrumentation is selective, and the data available to the user is a small subset of the actual data available. We find it necessary to distinguish between the ideal data embodied in the execution and the data actually available to the user through a tool. Ideal *events* occurring within the computer are recorded as *notes*. The analogy seems to refer to the common debugging practice of taking notes on paper as one observes execution, though the real simile is musical: a series of related events is a *chord*. Likewise, ideal *statistics* embody all possible state information we could peruse at a given point in time, such as memory contents, throughput, program counter locations, and the like. When we record a statistic, we refer to it as a 'tally'. This word is really most related to the way we analyze the note trace (our version of the 'event trace'): as we encounter events we 'tally' the changes in all states caused by each event. This terminology may seem like a hair-splitting triviality, but the appropriate choice of words allows us to converse in concrete and well-defined terms, facilitating the design and construction of otherwise virtually inconceivable data manipulations.

3 Converting notes to tallies

As debugger probes and real-time statistics are naturally interpretable as tallies, the real problem with integration is to express note trace data in tally form. This is simplified by realizing that tallies are naturally organized as vectors, where each vector consists of tallies of a particular type, and the index set of the vector ranges over all locations within the architecture where that type of tally is meaningful. The problem is to specify how to compute these vectors from the note trace.

This problem is made more difficult by the current interest in standardizing event traces. The benefits of standardization are obvious, as analysis tools may then be reused on traces from many different architectures. However, standardization has serious problems. First, one cannot adopt a standard which keeps researchers from defining new notes and note semantics; any standard must be extensible as new ideas are developed. Second, one cannot standardize the semantics of the notes, as it is currently impossible for researchers to agree on the semantics of even the most simple notes. A "message transmission" means something quite different on a SIMD machine than on a MIMD one.

The main difficulty results from the disparity between the way traces are recorded and the way they are analyzed. Whereas one can reasonably only record local system notes literally as they occur, one would rather analyze the effect of notes upon the tallies representing the global state of the parallel computer. For example, the latency in sending a large message between processes on different processors is best determined by subtracting the times of the notes corresponding to its sending and its receipt. By note semantics, we mean a system for translating from the values appearing in fields of the notes to a set of values (tallies) that are the basis for the analysis.

It is these semantics that are difficult to generalize, leading many implementors to code their note semantics into a specialized trace interpreter that is integral to the analysis tool[2, 6]. We view the analysis process as consisting first of the conversion of information in the notes into appropriate tally values, followed by the application of the analysis tool to the vectors of tallies thus generated.

3.1 Current standards

To date, two approaches to standardization are prevalent. The first approach, proposed by Reed[10], normalizes note trace formats so that each note simply asserts the value of one or more system statistics The format is self-defining: a header to the note trace defines the record format and presents a mapping between note fields and ASCII names which define field semantics to the user. There is no semantic content to the field itself; it just specifies a named number which the user is left to interpret semantically. To convert a trace to this format, one applies a filter that utilizes note semantics to produce a semantics-free trace. While this provides a nice interface to analysis tools (which merely must accumulate fields), the format is actually quite far removed from the raw note trace recorded in the machine. In effect, semantic analysis has been done already, by transforming the raw note trace into this form.

A second approach[12] is to provide a packaged interpreter for the note trace, including access functions for all the statistics the interpreter knows how to compute. This approach again avoids the issue of note semantics, by providing a prepared filter which essentially performs the same function as Reed's translator to normal form. Event semantics are interpreted internally, and the results are several arrays of data, semantics-free except for the names with which they are labeled.

3.2 Our approach

We have developed a working prototype of an alternative approach, in which note semantics are coded into a declaration file using a relational query language based on the work of Snodgrass[11]. The declarations are compiled into a program for a universal trace interpreter, which is responsible for the final translation from trace to tallies. There is no need for special access functions in the analyzer: it need only understand the data format produced by the universal interpreter. This strategy subsumes the function of the interpreters for the trace formats we now use, and we have yet to find a trace format whose semantics are not representable using it. We propose this method as a possible standard for defining the semantics of note traces.

Compilation of the declarations to an internal form suitable for interpretation is accomplished by a simple recursive-descent parser/translator for the declaration language. Some care must be taken in the compiler to ensure that the program it produces for the interpreter is reasonably efficient. To illustrate the issues and methods, let us take some examples from a set of declarations that describe mainly application-level notes for a parallel program for alpha-beta search[3].

3.3 Examples

Here is a set of declarations for a tally encoding the concept of the state of an algorithm. This reflects the fact that the computation proceeds into different phases at different times (somewhat unpredictably and asynchronously) on different processors. The enumeration of the dozen or so different algorithm states, as well as the declarations describing how to infer the state from the note data, are provided by the user who wrote the application. The declarations describe "Attributes" which are fields of notes, "Variables" for referring to values within declarations, and "Tallies" which are the values to be computed.

```
Attribute code Type(Short) Name ("Note code") Fetch(buh,2);
Attribute nodeid Type (Short) Name ("node") Fetch(buh,0);
```

```
Variable nn Type (Int) ;
Tally Alg_state Type(Pernode) Name("Algorithm state") Initial (IDLE) ;
When Ev ( code==S_START, nn=nodeid ) Set Alg_state[nn] = START ;
```

The first two lines declare two fields of notes. The "code" field serves as a primary key telling the note's type, while the "nodeid" field indicates the processor on which the note originated. "Fetch(buh,n)" describes how to fetch a field from a note, in this case by fetching an unsigned halfword stored in big-endian byte order. There is a large, expandable set of such fetch-functions, including some that handle fields whose size and location within a note are not constant, as with Heath's[5] or Reed's[10] formats. The third line describes a variable used in following lines to record the numerical processor identifier of the processor on which the note occurred. The fourth line defines the tally as an array of values, one for each processor in the parallel computer. The last line is one of many describing the dependence of this tally on a note having a particular code field. ("IDLE," "START," and "S_START" are mnemonics for literal values.) This line contains simple expressions involving assignment and relational operators; the syntax supports more complex expressions using a C-like syntax.

The compiler's job in this case is to digest the first four lines, passing declarative information along to the analyzer, and to convert the last line into a procedure that determines whether a given note affects the tally, and performs the update if it does. The compiler constructs one such procedure for each "When" statement, by viewing the statement as an expression to be evaluated. If evaluation fails, then the note does not affect the given tally; if it succeeds, the tally is updated via an assignment statement.

The input to the interpreter is a set of such procedures, which are evaluated using an unbounded array of "registers." Each attribute fetched from a note is put in a dedicated one of these registers. The compiler avoids refetching any field from a note by remembering what register it has been fetched into once it has been referenced in some expression. An assignment of a value to a variable is handled at compile-time by binding the variable to the register in which the value occurs. Bindings of variables to registers span one "When" statement, while bindings of attributes to registers span all the expressions.

A slightly more complicated example illustrates the use of groups of notes. The following declarations define a tally that records the cumulative elapsed time between the appearance of notes of two particular codes on a processor.

```
Attribute time Type(Int) Name("time") Fetch(bw,4);
Variable t1 Type (Int) ; Variable t2 Type (Int) ;
Tally Timewaiting Type (Pernode) Name("Time waiting for workers");
When Ev ( code==I_RETURN, nn=nodeid, t2=time)
After Ev ( code==S_WAIT , nodeid==nn, t1=time )
Set Timewaiting[nn] += t2 t1
```

```
Until Ev ( code==S_EXPAND, nodeid==nn ) ;
```

The "After" statement tells the interpreter to scan backward through the notes until it finds one that allows successful evaluation. The "Until" statement tells the interpreter when it may assume the search has failed and abort the search process. Without such a bound, if a desired note is missing for any reason, then the search will proceed through the entire trace before failing.

4 The visualization system

Designing a visualization user interface for execution data is considerably more challenging than designing a scientific data visualization system. For one thing, the data format for scientific visualization data is fairly well fixed: scalar and vector fields in 2-space and 3-space. Most data is continuous in nature and one can meaningfully interpolate between adjacent measurements. Also, the data values needed to render each point are quite unimportant after the rendering occurs. Rendering is the only goal, and global appearance is the only deliverable.

By contrast, execution data has many problems. Execution tallies are indexed over discrete sets by nature. A particular tally may be stored in an array indexed by several sets, relating to locations not in space but in the computer's architecture. For example, the tally representing memory location values might be indexed by processor number, process number, segment number, and offset within the segment. Index sets may change dynamically during execution, e.g., processes can be initiated and terminated. There is often no easily interpreted notion of 'neighboring value', and interpolation is almost never a meaningful operation[4]. Tally data has very high dimension, partly due to the way tallies are derived from notes, so that comparison of multiple displays is typically more important than obtaining a single view. Most important, the individual data values are indeed important even after rendering is done. The global view only serves to lead the user to particular subsets of errant values, which must then be interpreted as a bug in the program.

This need to work backward from the visualization to the original data, from that data to the event trace, and from the event trace to the bug, is unique to debugging. Much work on locating bugs from the event trace has been done in PPD[1]; the approach is called 'flowback analysis'. We generalize the problem by adding a visual component. I prefer the term 'causal backchaining', from the theory of program correctness. Backchaining is the process of using logical

[4] One tool, which shall remain nameless, displays communications throughput on a hypercube as a contour map of a grid, where each row is a message source and each column a message destination, both sequenced in gray code order. This provides an attractive display which, however, is meaningful only if the algorithm being studied exclusively uses the grid embedding shown on the map with only nearest neighbor communication. Using this display for any other kind of algorithm is inherently misleading.

predicates about the end of execution of a procedure to infer predicates about each line of the execution. In our case, the predicates are the values of tallies, and the backchaining first determines what events affected those tallies, then (indirectly) what caused the tallies to become undesirable.

4.1 A good debugger visualization system

These comments motivate the following discussion of an ideal visualization system for debugging. First, there must be a direct and easily understood correspondence between locations on each display and the individual data values which were used to create it. For example, as in Prism, we must be able to map each pixel in a two-dimensional map of array contents back to the array element which determined its color. But more important, there must be a map from summary data to the values affecting the summary.

Suppose processor state is coded as green for running and red for idle. Suppose the current state distribution is rendered as two lines, one green and one red, where the relative length of the lines indicates the balance between running and idle processors. The display is thus a frequency summary, where the length of each line is proportional to the frequency with which processors are in the line's category. Sometimes each pixel of a line corresponds with one processor in the line's category. So selecting this pixel is equivalent with selecting the processor for further study. More often, however, one pixel is present because many processors are in the category. Selecting a pixel therefore selects a subset of processors, and selecting several pixels (or a line segment) selects the union of the subsets for each pixel. The mapping between pixels and processors is not uniquely determined, and can be constructed in any way, as long as some mapping is available. In our case it is induced by an underlying ordering of processors, which can be modified using a sorting filter.

Second, all displays provided should be inherently scalable. The user should not be forced to learn to interpret displays which will not be of use in interpreting massively parallel executions. Many inherently scalable displays already exist which can be applied to understand small scale executions with the same facility as the unscalable displays currently in use.

Third, there should be an easy way to construct multiple displays of the same data Since the data is inherently many-dimensional, the user will often be making comparisons of similar data. One way to accomplish this is to fix the mapping from value to graphical attributes for all time. This is not desirable, as we do not yet know the most efficient mappings. Thus an ideal system allows multiple mappings, each of which can be used on several tallies, each of which is displayed using the same mapping in several different ways.

Fourth, there should be an easy way to limit discussion to a specified subset of the whole data set, redisplaying it in a different way. If only part of a display is in error, it should be possible to single out that part and redisplay it alone with more detail. One should be able to 'select and scope' a display to zoom in on detail, as well as to 'expand and illuminate' data by showing finer detail.

4.2 The data flow approach

We approach the problem of providing the ideal visualization system by utilizing a data flow model of analysis, similar to that used in the scientific visualization package AVS and the Pablo[10] event trace viewer. The data that flows consists of vectors of tallies, indexed by some (possibly cartesian product) index set. The *data flow graph* describing the whole visualization environment is a directed acyclic graph whose nodes are data sources, filters, and displays.

There are several kinds of sources, all producing similar data. A source can produce a tally from a debugger variable probe, an event trace, or a real-time data collection subsystem. One source can provide several kinds of tallies, each an independent output. Each kind of source has its own control window with which one can control what data is shown. An instance of the debugger is the control window for the variable probe, while the event trace window controls the current time being viewed and the real-time window allows specification of the real-time data to be collected.

Filters in our system differ from those in Pablo. A filter is often not a data transformation, but a augmentation: extra data is added to each element of the input tally. Added data includes graphical attributes (such as color, shape, and texture), selection information determining a subset of data of interest, and ordering information determining the order in which data is presented on an axis of a display.

The concept of a display is not too well distinguished from that of a filter. Many filters have associated displays. Graphical attribute filters, for example, allow the user to paint graphical attributes directly onto data displays created by the filter. A display is a special case of a filter, rather than the other way around.

4.3 Icons and instances

Reusability of attribute maps (and other filtration schemes) is accomplished using an *instance* scheme. Each filter car act independently on several sets of input vectors, providing an output vector for each set. Each output vector is produced by copying tally data verbatim from one of the input vectors while computing augmenting attributes based on the values of all vectors. By applying several instances of a filter to different data streams, the user can reuse a complicated filter configuration without copying or redefining it. Changes to the filter's setup affect all data streams which flow through it simultaneously.

An example icon is shown in Figure 1. Each icon has an identifying glyph, shown in the top square. Below this there is a vertical bar containing all instances. Each instance consists of several inputs and a single output, where the fan-in on inputs is 1, while each output has infinite fan-out.

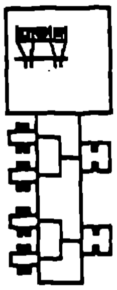


Figure 1: An icon with two instances and three inputs for each.

Icons are 'plumbed together' into a directed acyclic graph, as in Figure 2. Here two inputs from a real-time collection subsystem are mapped to graphical attributes, sorted by value, and shown on a couple of LED-style displays.

4.4 Filters

There are several kinds of filters we use to augment data before display. A *transforming* filter applies a mathematical function to get new data from old. Examples are logarithm, bit field extraction, integration (running sum) and differentiation (running difference). An *augmenting* filter adds attributes to data, such as color, texture, sorting order, etc. A *selective* filter selects a subset of its input. This is really another kind of augmenting filter, which adds a selection bit to each datum describing whether it is included in the subset. An *aggregating* filter groups data into groups, typically by summing. Currently we have only one kind of aggregative filter, which folds data indexed by hypercube processor number onto a subcube.

A key remark is that aggregating is to be avoided if at all possible. An aggregating filter in the chain is a many-to-one mapping. In backchaining to the original data, one thus has a one-to-many map in the reverse direction. This means that the only way the system can respond to a request for more information on an aggregate is to present a subset of values rather than a single one. The only way to get specific information on an aggregate is to back up in the chain before the aggregation operation and re-render the *input* to the aggregator in a new way.

4.5 Displays

Displays take inputs which have already been augmented by graphical information, and pick and choose within that information to come up with a rendering

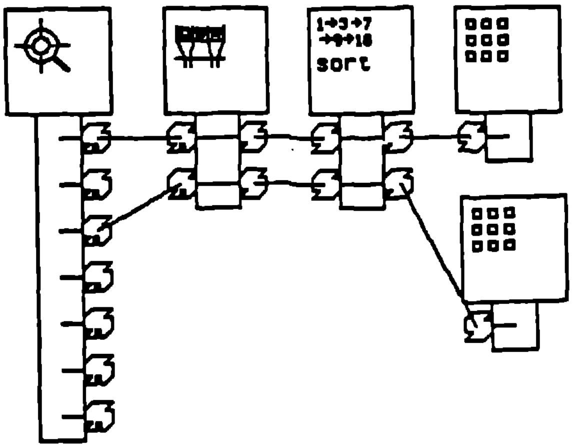


Figure 2: A sample data flow visualization scheme.

of the data. Each display has the right to utilize or ignore augmentations as it pleases: one display may utilize a textural augmentation while another ignores it.

The requirement that displays be scalable severely limits the kinds of displays we can construct. We must rely instead upon zooming to see detail. The only scalable displays are scrollable text, pixel maps, scatterplots, and backchainable summaries.

A pixel map is simply a display where each datum controls the color, brightness, or texture of a single pixel in a two dimensional array. The pixel in question can be rescaled for easy viewing; a display pixel may be 9 screen pixels wide. Selecting a group of pixels allows them to be redisplayed in a different way.

A scatterplot is particularly useful because it is inherently scalable. A one-dimensional scatterplot displays values as hash marks on a number line, while a two-dimensional scatterplot renders pairs of values as points in a Cartesian coordinate system. Scatterplots provide a global navigating point from which to peruse the data. If one is interested in maximal values, one selects them on the scatterplot and redisplays the output of the scatterplot to see them. Changing the selected region automatically changes the subsequent display of selected items.

Summaries are only provided if there is a well-defined notion of which data affect each part of the summary image. For example, in a bar chart showing frequencies of values, selecting a bar selects all values counted as within the bar's category. Then these values can be displayed subsequently.

4.6 Usage

The user of our system has two basic operations at hand. One can select and zoom in on particular data of interest from a global context, or augment and illuminate data by adding information. A typical use is to understand 1024 values of a variable x, obtained from the debugger. Suppose that x is a phase counter for your algorithm, and that you expect all values to be 2. The first action will be to scatterplot x in one dimension. Suppose all values are not 2, so that several other hash marks appear on the scatterplot. The second action is to select the other hash marks for further analysis. Then the output of the scatterplot is connected to a textual display one can peruse, showing only the errant values. But suppose this display is still too hard to understand. One can map the data instead into colors and show the results on a pixel map, to get an idea of how many processors there are in each incorrect phase. One can then select subsets of the pixel map and repeat the process.

4.7 A prototype

As a test case, we are implementing a debugging and visualization environment for the NCUBE/2. This prototype uses operating system instrumentation[7] to provide an event trace and real-time data for each execution. Compiler instrumentation is not used. The NCUBE/2 was chosen for this because there are now delivered systems with 1024 processors: the maximum configuration is 8192. While our prototype is aimed at one specific architecture, our real goal is to develop paradigms of use and debugging strategies which are applicable to any massively parallel MIMD or SIMD architecture.

The current prototype has the data flow visualization system in place, with data sources for both event traces and real-time statistics. We are working now upon the integration of our debugger "tdb"[4] into the visualization system. Currently the visualization system supports only 'augment and illuminate' style operations; we expect soon to have selection and zooming working properly.

4.8 Implementation details

It is important, especially when dealing with real-time visualization, to make one's visualization system as efficient as possible. This is done in our system by using a fragmented vector representation of data with access counts. Each tally is a dynamically allocated array of data, where each cell contains both the current and previous values of a datum for optimal display updating, as well as a pointer into a linked 'change list' of array cells which have changed in value since the previous display update. The array is passed from filter to filter by reference, and an access count is kept for each filter which references the array.

Augmentation data is kept in a series of arrays with parallel structure to the tally array, each also passed by reference with accesses counted. Augmentation arrays are created on demand by filters, and are controlled and updated only by the filter which created them. If a filter overrides an augmentation, that augmentation is still available before the filter; if one maps colors twice, then there is one color map before the second map filter and another afterward.

5 Beyond data flow

Unfortunately, our work has shown not only the benefits of data flow but also its limitations. The main limitation is the base data type: the tally vector. Defining index sets is difficul' especially since the index sets can change dynamically. Further, the model completely excludes direct manipulation of note data except in the tally projection of that data. A different approach is needed to completely integrate all forms of information into a complete whole.

The answer seems to be a simple generalization of the data flow model. For simplicity, consider the space of all tally values as a dat:base relation:

```
tally(name, location, value)
```

Then we can replace the vector of tallies of 'messages written' with a relational predicate

```
tally(name=="messages written", location, value)
```

which is true when the name of the tally is 'messages written', the tally has a concept of location, and the tally has a value. The predicate can replace the vector, provided there is a way to generate the relation from the predicate. Augmentation of a vector is achieved by augmenting the relation, so that a color map filter transforms the relation into a new one

```
tally2(name, location, value, color)
```

with associated predicate

```
tally2(name=="messages written", location, value, color)
```

meaning that the second version of the tally has the same name, a defined location and value, and in addition a color. Under this set of semantics, all data is globally available, relational predicates are the medium of communication between filters, and filters modify the global relational space for new relations, passing this on to the next filter by modifying the query predicate appropriately. In other words, the appearance of the data flow graph remains the same, but the semantics are those of a graphical query language.

The semantics of the revised graphical interface are quite simple. In the old system, an arrow from filter A to filter B asserted that data flows from A to B.

In the new system, the same arrow is interpreted as 'A scopes B'. That is, the output of A is a set of predicates which direct B to consider a particular subset of the data, and, more importantly, to ignore the rest.

This change in semantics has important ramifications. Several inputs can be meaningfully fed into a single input port; the result is the predicate which is the logical conjunction of the input predicates. For example, if one predicate says that the processor number must be between 0 and 7, and another predicate says that the parameter of interest is 'messages written', then the conjunction is the tally relation containing 'messages written' for processors 0 to 7. This means that parts of a single tally definition can be provided from different filters, none of which has a complete picture of the definition. Also, the graphical representation of the visualization environment no longer directly corresponds with the actual processing of data taking place; it is only a semantic representation of the system, not a description of the underlying data manipulation.

5.1 Our relational model

We restructure our database model to be easily usable in this context. First, we insure that a relation field with a given name always has the same semantic interpretation, i.e., there is a well-defined mapping from a field's name to what its value means, and hence to what it is appropriate to do with the value in that field. Second, we view each tuple in a tally relation as a tuple in a single global relation, where unmentioned fields in a tuple are assumed to be left blank. We thus combine tuples with dissimilar structure into a unified relational context. For example, the relations:

```
program_counter(processor, process, offset)
messages_sent(tally, time, processor, value)
```

are transformed into subsets of the same relation

```
tuple(relation=="program counter",
      processor, process, offset)
tuple(relation=="tally",
      tally, time, processor, name=="messages sent", value)
```

where the fields of the whole relation are

```
tuple(relation, processor, process, offset,
      tally, time, name, value)
```

and fields are left blank when not meaningful in context. Since we are dealing with a single global relation, we need not mention it in predicates. To select all tuples referring to program counters, an appropriate predicate is:

```
relation=="program counter"
```

For our purposes, we need predicates for equality and membership in a set.

This database organization is for ease of specification, not ease of manipulation. Internal representation will differ. In fact, many relations will not be stored at all, but simply queried from the computer architecture on demand.

5.2 The global relation space

For purposes of illustration, let us define a simple global relation in terms of its component relations. This is an intentionally oversimplified model, constructed only to illustrate a basic design point.

1.
   ```
   tuple(relation=="note",
         note,time,processor,event,value)
   ```

   This encodes the global space of events. `note` is a note serial number, in local processor context. `event` is a code determining the type of event, and `value` is an associated value.

2.
   ```
   tuple(relation=="tally",
         tally,time,processor,name,value)
   ```

   This relation encodes the global space of tallies derived from a note trace. `tally` is a tally serial number. `name` is the name of the tally whose value is given by `value`.

3.
   ```
   tuple(relation=="snapshot",
         snapshot,time,processor,name,value)
   ```

   This relation encodes the global space of real time snapshots of execution statistics. `snapshot` is a snapshot serial number. `name` is the name of the tally whose value is given by `value`.

4.
   ```
   tuple(relation=="program counter",
         processor,process,offset)
   ```

   This relation encodes the global space of program counter locations, where `offset` is the current value of the program counter for the given processor and process.

5.
   ```
   tuple(relation=="variable",
         processor,process,segment,offset,name,type)
   ```

   This relation encodes the global space of variable definitions, encoding both the `name` and `type` of each active variable.

6.
   ```
   tuple(relation=="source",
         offset,file,line)
   ```

   This relation encodes the relationship between program counter locations and source file lines, where `offset` is a program counter value

5.3 Inputs and outputs of debugger components

Our goal now is to characterize usage of these relations in an integrated debugging environment. Given that we have a globally consistent field name space, predicates about fields are used as the communication medium between debugging components. Each component uses the global database as it sees fit, scoped by field limitations input from other components. The result is a generalization of window based debugging which allows dynamic rescoping of views as a result of connecting or disconnecting components.

The *execution control window* is a control panel. Its inputs are predicates on program counter value, which are interpreted as breakpoints[5], and predicates on processor and process, which are interpreted as the scope of the panel's control. E.g., if only nodes 1 and 2 are allowed, then control operations initiated at the panel only apply to those nodes.

The *text window* displays program source. Its inputs are predicates for processors and processes to track. It uses the current values of the program counter (`relation=="program counter"`) as well as source code correspondence (`relation=="source"`) to display program counter locations and accept user input for breakpoints, which are predicates involving the program counter (the `offset` field of the program counter relation). These predicates may be input to the execution control window.

The *note window* shows the event trace. Its inputs are predicates for processors and event types to track. It uses the note trace (`relation=="note"`) to display a human-readable or graphical trace depiction. Its outputs are predicates on processors and event types. Processor predicates can scope any other window, while predicates on event type are used solely to scope a child note window to zoom in on detail.

The *tally window* allows the user to select tallies for display. It has no predicate input, and its output consists of predicates on the `name` field of the `tally` relation. This output is subsequently used by a display window to render the data.

The *probe window* allows the user to specify memory to examine. It has no predicate input, and its output consists of predicates on the `name` field of the `variable` relation.

The *display window* allows the user to display data. Its inputs are predicates describing subsets of tallies, real-time statistics, or notes. It uses the inputs to render the data selected. Its outputs are derived predicates, modified by user input and selection.

This is just a sampling of the components which could be utilized, intended to give the flavor of the design without descending too deeply into details.

[5] Excuse the lack of watchpoints in this model, for simplicity.

5.4 Existential and universal fields

This model is made user-friendly by defining defaults for unspecified scopes. We are greatly aided by the natural paradigm of use for a debugger: defaults take one of two forms. Either the default is to select all of a field, or none of it: fields are thus *universal* or *existential*. Characterizing a field is easy: it is existential if the normal operation is to ignore the field, and universal if the normal operation is to include all values. Things to be displayed are universal, i.e., the default is to show everything. Scope of processor control is universal, i.e., the default is to control everything. Breakpoints are existential, i.e., the default is to run without interruption. Other data is similarly characterized.

5.5 Paradigm of use

The fundamental advantage of this organization is that scoping operations can be applied and removed easily. Suppose one wishes to add a few breakpoints temporarily. One creates a new text window, sets the breakpoints there, and connects the window to execution control, where it asserts the breakpoints. Removing the breakpoints requires disconnecting the windows, and reasserting them just requires connecting them again. The text window containing the breakpoints has a life separate from the actions it specifies; disconnecting it retains the definition of the actions without their execution.

A second advantage of the organization is the ease with which one can move between levels. Selecting events in a note display can affect which processors are displayed in a visualization window, simply by coupling note selection and visualization through predicates. One visualization can be created relative to several different kinds of input: one can select the parameter from the real time statistics, select the processors from the notes, and the processes from a summary display of process behavior. This complete interdependency is necessary in order to fully integrate all forms of information we have available.

6 Results and conclusions

There is clearly great value in integrating the vastly dissimilar data available to a debugger into a coherent whole, but there are still many problems to solve. Transforming all data into a shared 'normal form' (such as tallies) is a partially successful approach, as one may still need to see data in its raw form due to omissions and deletions during the transformation. To the extent that the 'normal form' of tallies succeeds, data flow analysis has proven to be a good approach to understanding them. The base strategy of augmenting information instead of transforming it preserves relationships between the depicted image and the raw data which generated it, which in turn is related to the nature of the bug being analysed. The data flow graph depicts global context for the user, serving as a navigating point and allowing reuse of complex filtration schemes for visualizing distinct but similar datasets.

The alternative approach is to leave all data in essentially raw form, and express all displays as queries into a globally accessible database. This has benefits and problems. The major benefit is that the many facets of execution data may be viewed in essentially raw form, with each interaction from the user causing display scope modifications for data in other forms: selecting an event can modify display of a statistic. This tight interdependency between displays is a nice expression of the similar tight dependency between types of performance data.

But even this approach has problems. Query propogation is a lot harder for users to understand than data flow, being a foreign concept to most supercomputer developers, who come from a numerical analysis background. I expect a modicum of user resistance for this reason. It is because of this expected resistance that I have intentionally embedded data flow semantics within the new semantics as a subset. I even expect resistance to the multiple windows within which one views different data types.

While parallelizing compilers may someday provide acceptable performance for massively parallel machines, it is my view that there will always be a need for the programmer to understand the workings of the architecture to achieve optimal performance. Our tools and approach are aimed at this goal, and thus targeted at the small audience which really wishes to push these architectures to the limits of their capabilities. For this noble purpose, complete, interrelated execution data is essential. The relational model is designed for this purpose above all others.

References

[1] Choi, J., B.P. Miller, and R.H.B. Netzer, "Techniques for debugging parallel programs with flowback analysis", *ACM Trans. Prog. Lang.*, Vol. 13, No. 4, Oct. 1991.

[2] Couch, A., *Graphical Representations of Program Performance on Hypercube Message-passing Multiprocessors*. Tech. Report 88-4, Tufts Univ. Dept. of Computer Science, April 1988.

[3] Couch, A., and D.W. Krumme, "Monitoring parallel executions in real time." *Proc. Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press, 1990.

[4] Couch, A., and D.W. Krumme, "Multidimensional spreadsheets in a graphical symbolic debugger for the ncube", *Proc. Sixth Distributed Memory Computing Conference*, IEEE Computer Society Press, 1991.

[5] Geist, G.A., M.T. Heath, B.W. Peyton, and P.H. Worley. *PICL: a portable instrumented communication library, C reference manual,* Tech. Report ORNL-11130, Oak Ridge Nat. Lab., Oak Ridge, TN (1990).

[6] Heath, M.T, and J.A. Etheridge, *Visualizing Performance of Parallel Programs,* ORNL Technical Report TM-11813, Oak Ridge Nat. Lab., Oak Ridge, TN (1991).

[7] D. W. Krumme, "The SIMPLEX Operating System." *Proc. Third Conference on Hypercube Multiprocessors,* ACM Press, 1988.

[8] Lapolla, M.V., "Toward a theory of abstractions and visualizations for debugging massively parallel programs", to appear in *Proc. of the Two Day Miniconference on Parallel Programming Tools,* Hawaii International Conference on System Sciences, 1992.

[9] LeBlanc, T.J. and J.M. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay,* Butterfly Project Report 12, Computer Science Department, University of Rochester, 1986.

[10] Reed, D.A., R.D. Olson, R.A. Aydt, T.M. Madhyastha, T. Birkett, D.W. Jensen, B.A.A. Nazief, and B.K. Totty, "Scalable Performance Environments for Parallel Systems", *Proc. Sixth Fifth Distributed Memory Computing Conference,* IEEE Computer Society Press, 1991.

[11] Snodgrass, R. "A relational approach to monitoring complex systems." *ACM Trans. Computer Systems,* May 1988.

[12] "Performance Tools from the System Design Perspective," A white paper from the system design group, 1991 Workshop on Parallel Computer Systems: Software Tools, Bill Appelbe (ed.), In preparation.

Animation and History: Analyzing Programs Over Time

Kent L. Beck
Jonathan B. Rosenberg
MasPar Computer Corporation
749 North Mary
Sunnyvale, California USA 94086
408/736-3300 kentb@maspar.com

Serial programming environments have not provided explicit support for tracking changes in program state over time. The MasPar Programming Environment has recently been enhanced with program animation and history recording which provide more powerful support for understanding the behavior of complex, data parallel programs.

1. Introduction

Programming environments have traditionally included a rich set of tools for mapping programming language abstractions onto machine abstractions (e.g. compilers). The reverse mapping has not generally been as well supported. The ideal programming environment creates the illusion that the underlying machine directly executes the statements of the programming language. Figure 1 shows the MasPar Programming Environment (MPPE)[1], a tool for helping programmers understand the behavior of complex programs. MPPE maps the abstractions of a massively parallel, SIMD architecture, the MasPar MP-1[2], back into data parallel programming languages, currently represented by data parallel variants of C and Fortran[3,4]. MPPE delivers graphical, source-level control of the execution of a program, tabular and visualizer-based inspection of variables, and incremental, graphical statement and subroutine profiling[5].

One of our goals in writing MPPE has been to eliminate the need for recompilation as a debugging strategy. Unix™-based programming environments require recompilation to operate symbolic debuggers, to invoke optimization (which is mutually exclusive with debugging), and to profile. All of these operations are enabled simultaneously by the default switches of the MasPar compilers.

Enabling debugging and profiling are not the only debugging operations invoked by recompilation, however. One common debugging strategy is to insert diagnostic print statements. As the programmer's understanding of the location of a bug changes, the print statements need to be changed and moved around, necessitating the recompilation of at least one source file and the relinking of the entire executable. In interviews with users we discovered that one of the features of debug print statements they liked was the ability to review a variable's values over time, and to juxtapose two changing variables to understand their interaction. In data parallel programs, with their large data sets and parallel control structures, support for this style of debugging is particularly important.

We have added a history recording mechanism which, in conjunction with program animation, eliminates the need to recompile programs to track and compare variables over time. In addition, we have found the history mechanism amenable to a variety of extentsions to aid in the understanding of data parallel program behavior.

2. Execution Control

Mapping the machine's view of a program execution (expressed in terms of machine instructions) into the programmer's view (expressed as programming language source statements) is the job of the execution control commands. Execution control illuminates control-oriented bugs by showing which statements are executed, for instance in a conditional construct. As all displayed data values are updated whenever the program stops, execution control also provides entry points for inspecting data.

MPPE provides the three essential execution control commands: "continue", which runs the program until termination or until a breakpoint is reached, "step", which executes the program until the beginning of the next source statement, and "skip", which executes until the beginning of the next source statement in the current routine. Breakpoints are created and destroyed either by clicking in a dedicated screen region next to a source statement, or by selecting a routine name and issuing a menu command. Breakpoints, once created, can be given various attributes such as an ignore count, which associates a down counter with the breakpoint which is decremented every time the breakpoint is reached. When the counter reaches zero it is reset to its original value and the program is stopped. Finally, execution can be quickly controlled by selecting a line or routine name and issuing the "go to line" or "go to routine" commands.

One of the "accelerators" in the MPPE user interface is the user of the carriage return key to repeat the last menu command. When used with the execution control commands this feature allows the user to quickly step through a program, although it introduces the possibility of the "single step twitch", a distressing quivering of the pinky finger on the right hand.

3. Animation

While enhancing the demonstration value of MPPE we decided to implement a "follow the bouncing ball" style of animation, where the program would continuously single step without any user intervention. At first we added an "animate" menu item for this feature, but a user quickly asked for a a version of animate that worked with "skip" instead of "step". Rather than add yet another menu item we took a step back to see if we could devise a more general solution.

As shown in figure 2, we decided to add a check box, labelled "Animate", underneath the machine icon. If the user checks the box, any subsequent execution control command is repeated until either the box is no longer checked or the program stops for some reason other than the command. For instance, by checking "Animate" and then issuing the "step" command the program will continue to single step until the program terminates or encounters a breakpoint. The same is true of "skip", "go to line", and "go to routine". "Continue" is slightly different, as encountering breakpoints does not stop the animation.

While the "Animate" button adds no functionality to the environment that could not be duplicated with a sturdy right pinky, we have noticed in ourselves and our users a qualitative difference between just watching a program execute and having to intervene at every step. The user need no longer pay attention to when the program has stopped in order to issue the next command, and is instead free to concentrate on the program. This "program as movie" style of debugging has proved particularly useful for uncovering control bugs, especially those introduced by data parallel control constructs where the then and else parts of a conditional can both execute on disjoint sets of processors.

4. History

Shortly after adding program animation we began to experiment with recording historical information. Our two inspirations were the desire to improve on debugging print statements by flexibly displaying and juxtaposing variables and the need to provide a tool for analyzing the processor utilization profile of programs. These seemingly disparate applications are both time varying, and our solution takes advantage of that fact to present and single, simple view of all historical data.

Our design uses a logic analyzer or oscilloscope metaphor, two instruments which solve the "juxtaposition of values over time" problem. The history window (see figure 3) contains a number of "traces", which are arranged along the top of the window, while time occupies the vertical axis. The quanta of time, one instance of the program stopping, is marked along the left edge by a tick mark. Pressing and dragging one of these ticks changes the magnification of time. A vertical scroll bar provides motion through time.

Variable traces are added to the history view by copying it from the source code or an inspector and pasting it into the view. Figure 4a shows a textual trace of a variable, which displays the value of the variable in a box proportional in size to amount of time the variable still printed identically. If time has been compressed enough so that the printable representation no longer fits in the box, the printing is omitted, with the closely spaced gray lines indicating rapid change (figure 4b). Figure 4c is a graphical trace, where the user can set the bounds of the graph by dragging the limit axes horizontally. A final variable trace we have experimented with is the boolean trace, where the user can type a simple expression on the variable and the trace turns gray where the expression is true (figure 4d).

Program variables are not the only values to vary over time. We have also added a processor utilization trace which displays a histogram of the number of processors active, and a clock trace which shows the current usage of CPU time down to ten millisecond increments (figure 5). The latter makes apparent anomolies in the amount of time necessary to execute iterations of a loop. Other traces display the available heap and stack space, which are useful in finding memory leaks and otherwise analyzing the memory allocation behavior of a program.

The history view has a powerful synergistic effect with program animation. While animating it is not always possible to keep track of several variable values at once if they are in separate windows, but the history view makes their connection obvious. Also, a slip in concentration may cause a user to miss an important event in an execution, while the history view perserves the information. Animation serves the history view by making it easy for the user to collect long traces without user intervention.

5. Scenario

We will demonstrate animation and history with two examples: one showing how history can help the user discover data-oriented bugs and the second showing how it helps in tuning data parallel programs.

5.1 Wrong comparison operator

A common novice mistake in C is using the wrong comparison operator in a looping construct. The following program writes off the end of an array, mistakenly modifying the value of the outer loop variable while calculating the inner loop.

```
int j;
float a[5];
int i;

a[0] = 3.14159;
for (j = 0; j < 100; j++)
    for (i = 1; i <= 5; i++)
        a[i] = a[i-1] * 3.14159;
```

Figure 6 is a trace of i, j, and a[5] gathered by animating "step". The trace clearly shows that j does not advance beyond zero until its value is modified by the last iteration of the inner loop.

5.2 Processor utilization

Figure 7a is a trace of a benchmark program recently coded for a potential customer. It shows that all processors are not being used all the time. With the information in this trace the programmer was able to modify the program to produce the trace in figure 7b, which acheived an aggregate 25% speedup over the earlier version.

6. Conclusion

We were pleased with the simplicity and flexibility of our program animation facility. At a cost of one additional button in the user interface the user is able to watch "interesting" points of a program by a straightforward extension of familiar execution control commands. Program animation increases the illusion that the hardware is really a Fortran machine or C machine, and increases the user's understanding of the behavior of a program in those languages.

By adding history recording we were able to eliminate one of the remaining excuses for putting recompilation in the debugging cycle. In addition, online history has many advantages over the teletype version: more time can be viewed with the continuous pan and zoom features, the data can be viewed in more formats, and important derived data such as the percentage of active processors can still easily be juxtaposed with variable values. History and animation also have a synergistic effect, each supporting and making the other more useful, without introducing new complexities.

In the future we want to explore the use of language expressions entered at run time as the source of data for traces. We would also like to find ways of effectively making traces of aggregate data, such as large arrays. Finally, a simple extension of history makes it possible for the user to select a tick mark and see the corresponding source line.

Animation and history are a powerful combination in promoting the understanding of complex programs. We have shown how both facilities can be added at minimal cost to the user's model of the programming environment, and how the two facilities can work together to provide value beyond the sum of their individual contributions.

References

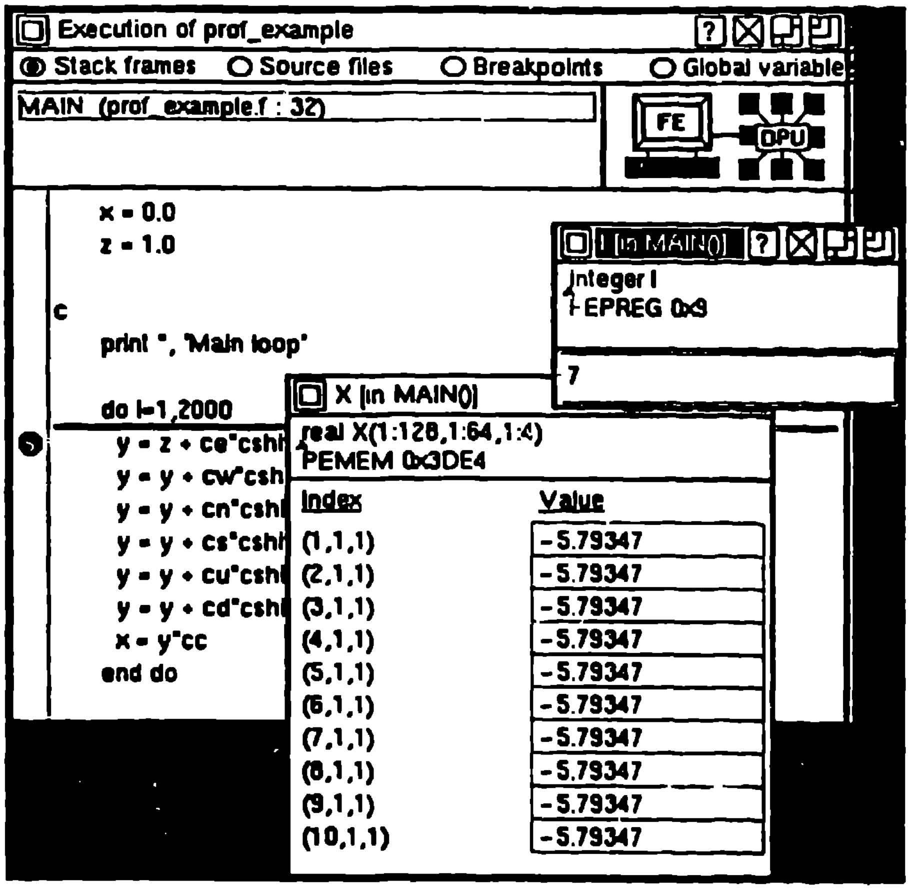
[1] *The MasPar Programming Environment Reference Manual*, MasPar Computer Corporation, 1990.

[2] Tom Blank, "The MasPar MP-1 Architecture", *Proceedings of IEEE Compcon Spring 1990*, IEEE, February 1990.

[3] *The MasPar Parallel Application Language Reference Manual*, MasPar Computer Corporation, 1990.

[4] *The MasPar Fortran Language Reference Manual*, MasPar Computer Corporation, 1990.

[5] "Integrating Profiling into Debugging," Kent Beck, Zaide Liu, Jon Becher, submitted to the International Conference on Parallel Processing.



Execution of prof_example
● Stack frames ○ Source files ○ Breakpoints ○ Global variable
FE
DPU
I (in MAIN()
X (in MAIN()

MAIN (prof_example.f : 32)

```
      x = 0.0
      z = 1.0

c
      print *, 'Main loop'

      do i=1,2000
         y = z + ce*csh
         y = y + cw*csh
         y = y + cn*csh
         y = y + cs*csh
         y = y + cu*csh
         y = y + cd*csh
         x = y*cc
      end do
```

Integer I
EPREG 0x9

7

real X(1:128,1:64,1:4)
PEMEM 0x3DE4

Index	Value
(1,1,1)	-5.79347
(2,1,1)	-5.79347
(3,1,1)	-5.79347
(4,1,1)	-5.79347
(5,1,1)	-5.79347
(6,1,1)	-5.79347
(7,1,1)	-5.79347
(8,1,1)	-5.79347
(9,1,1)	-5.79347
(10,1,1)	-5.79347

Figure 1: The MasPar Programming Environment Debugger

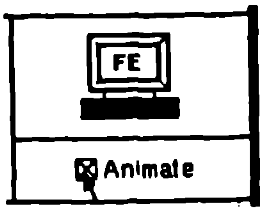


Figure 2 Animate button

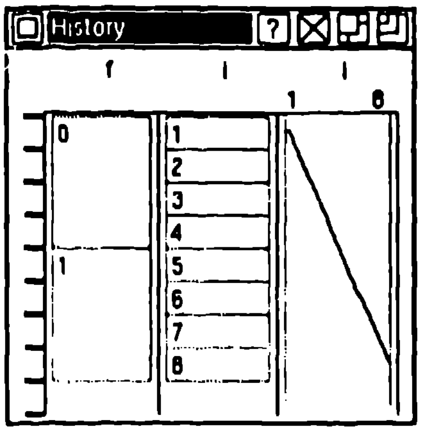


Figure 3 History Window

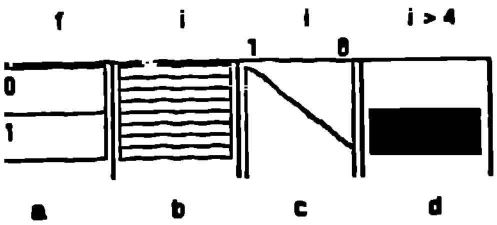


Figure 4: Variable Traces
a) textual
b) quickly changing textual
c) graphical
d) comparison

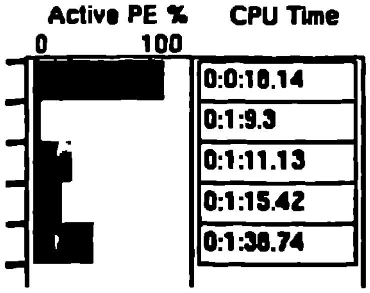


Figure 5: PE utilization and CPU time traces

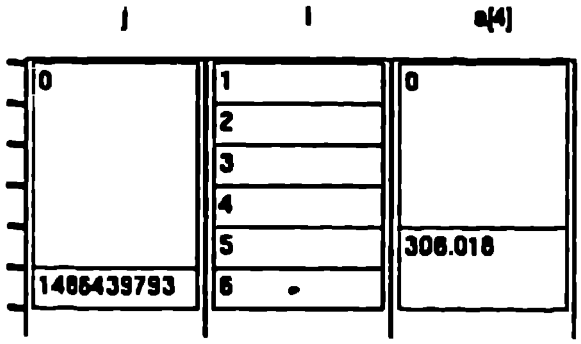


Figure 6: Tracing out of bounds array reference

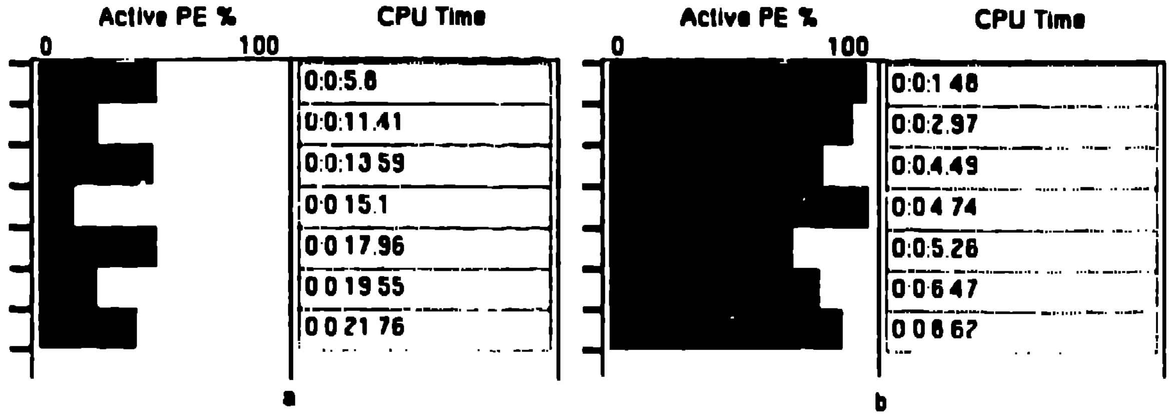


Figure 7 Processor utilization traces before and after tuning

A Distributed Debugger Architecture

Ann Mei Chang
Philip L. Karlton
David M. Ciemiewicz

Silicon Graphics Computer Systems
2011 N. Shoreline Blvd.
Mountain View, CA 94039

ABSTRACT

1. Introduction

Traditionally, most debuggers have been designed using a monolithic architecture, where all of the debugger functionality is contained within a single binary and a single process. While this model was sufficient for simple tasks, more complex debugging tasks involving multiple processes and remote execution are not well supported. Furthermore, very little flexibility is available for providing different user interfaces and for user extensions.

The *CodeVision™ Debugger*[1] architecture is based upon a client-server model. A central *Process Control Server* provides high level control and access features for debugging, while multiple client views communicate with the server and present data to the user. This architecture enables the debugger to support such features as multiple user interface presentations, multiprocess debugging, remote debugging, conference debugging, and user-customizable views

2. A Distributed Architecture

In the *CodeVision Debugger*, the user interface components function as clients to a shared server called the *Process Control Server*. Communication is achieved through a specialized protocol, the *Process Control Protocol*, built on top of TCP. The protocol handles synchronous and asynchronous requests as well as asynchronous events. Clients may make high level debugging requests such as a request to "single step process *x*". The *Process Control Server* responds to requests (either synchronously or asynchronously) and sends out events of interest as they occur, such as "process *x* has stopped at location *y*"

The *Process Control Server* is capable of managing multiple processes and multiple client views simultaneously. A central event loop dispatches events as they arrive asynchronously either through the /proc debugging interface (for target process state changes) or through a client connection. As a single *Process Control Server* controls all the processes being debugged on a given host, it is capable of synchronizing events for multiprocess debugging. Thus, it may handle actions that affect all processes in a multiprocess group such as explicit process control or traps which stop all processes in the group.

On the client side, views present a user interface for the debugging features available through the *Process Control Server*. The user interface may be as simple as a dbx-like interface implemented in a single view which communicates directly with the *Process Control Server*, or as complex as a graphical interface with multiple views implemented in multiple processes. Specialized views may be written as separate processes, so that they are only invoked when needed. For example, in the *CodeVision Debugger*, views for machine level debugging are in a different process from the more generally used views

[1] CodeVision is a trademark of Silicon Graphics, Inc.

The distributed architecture of the *CodeVision Debugger* provides several unique features. As mentioned before, a high degree of flexibility is available in the user interface, allowing different interfaces to be built on top of the underlying *Process Control Server* as well as interfaces consisting of multiple processes. Furthermore, this model directly supports the ability for end users to construct their own user interface components, tailored for their particular debugging scenarios. The client-server approach also provides the capability for remote debugging (where views run on a different machine th[illegible] server and target process) and conference debugging (where two different users may examine the same tar[illegible]ultaneously at different workstations).

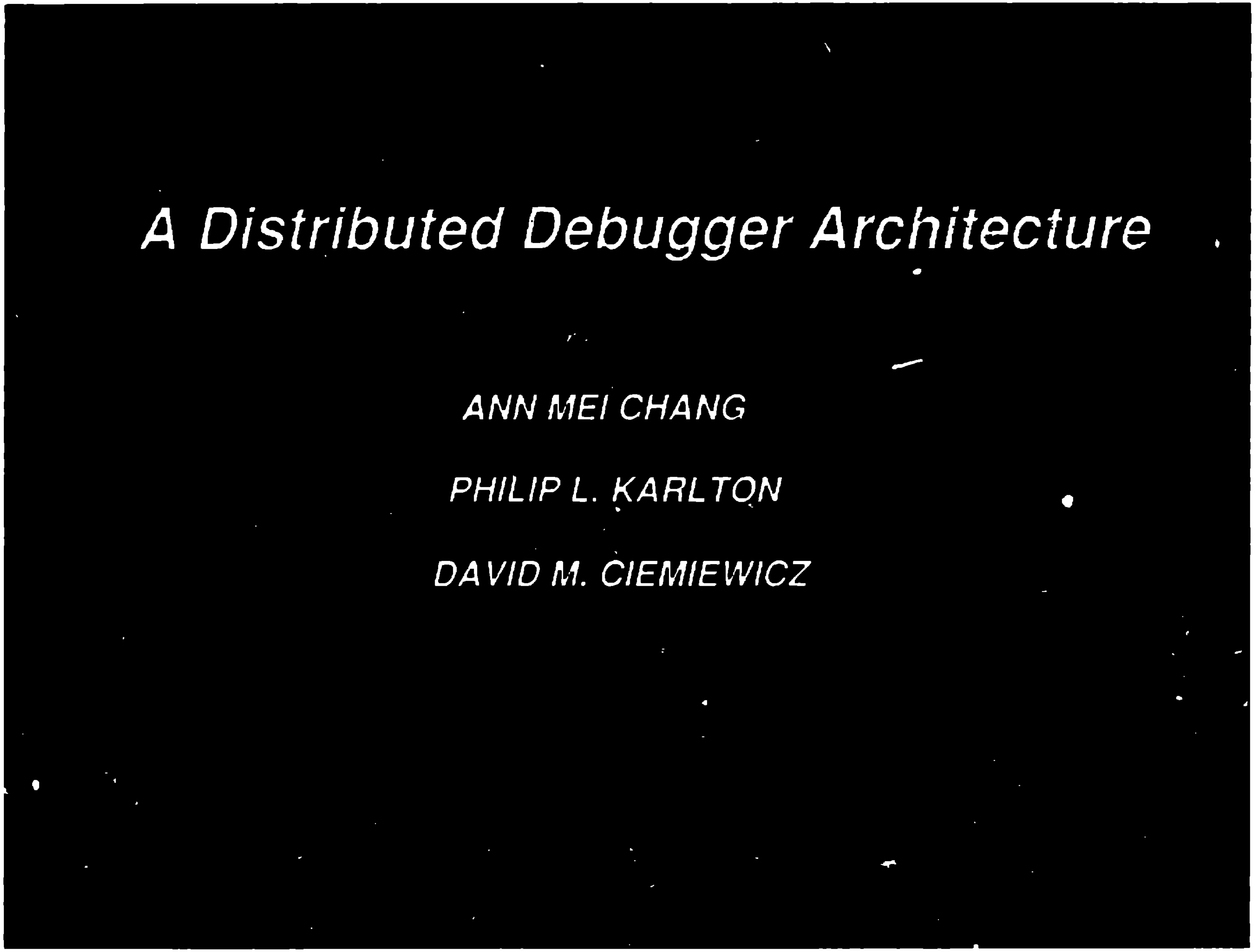
3. Multiprocess Debugging

The distributed nature of the *CodeVision Debugger* is especially useful for supporting multiprocess debugging. Traditionally, debuggers originally designed for single process applications have been extended to support multiple processes. In such cases, the handling of multiple processes is often clumsy, requiring the user to switch a single view of the target between the various processes.

By supporting multiple views based on the client-server approach, the *CodeVision Debugger* allows the user to choose between switching existing views to different processes or, alternatively, bringing up separate views for the processes of interest. Additionally, a specialized view, the *Multiprocess View*, displays the current status of each process in the group being debugged, updating the data dynamically as state changes occur.

4. Conclusions

By separating the internal debugger functionality from the user interface components, a great deal of flexibility is gained for user interface design, multiprocess support, and user customization. Distributed, client-server models for computing have become more common as faster workstations are built. We believe that this technology can also be utilized effectively in debugger architectures.

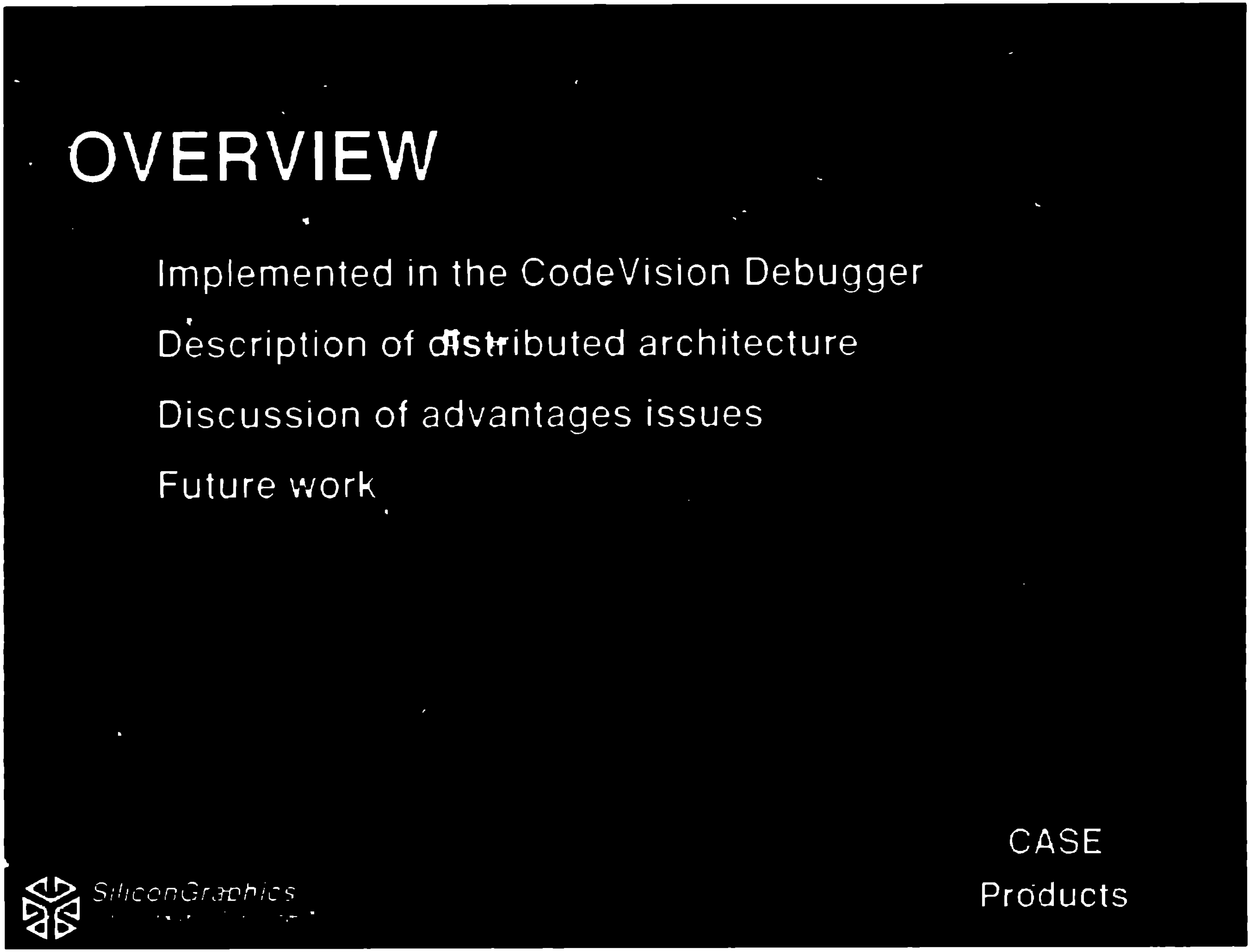


A Distributed Debugger Architecture

ANN MEI CHANG

PHILIP L. KARLTON

DAVID M. CIEMIEWICZ



OVERVIEW

Implemented in the CodeVision Debugger

Description of distributed architecture

Discussion of advantages issues

Future work

SiliconGraphics

CASE
Products

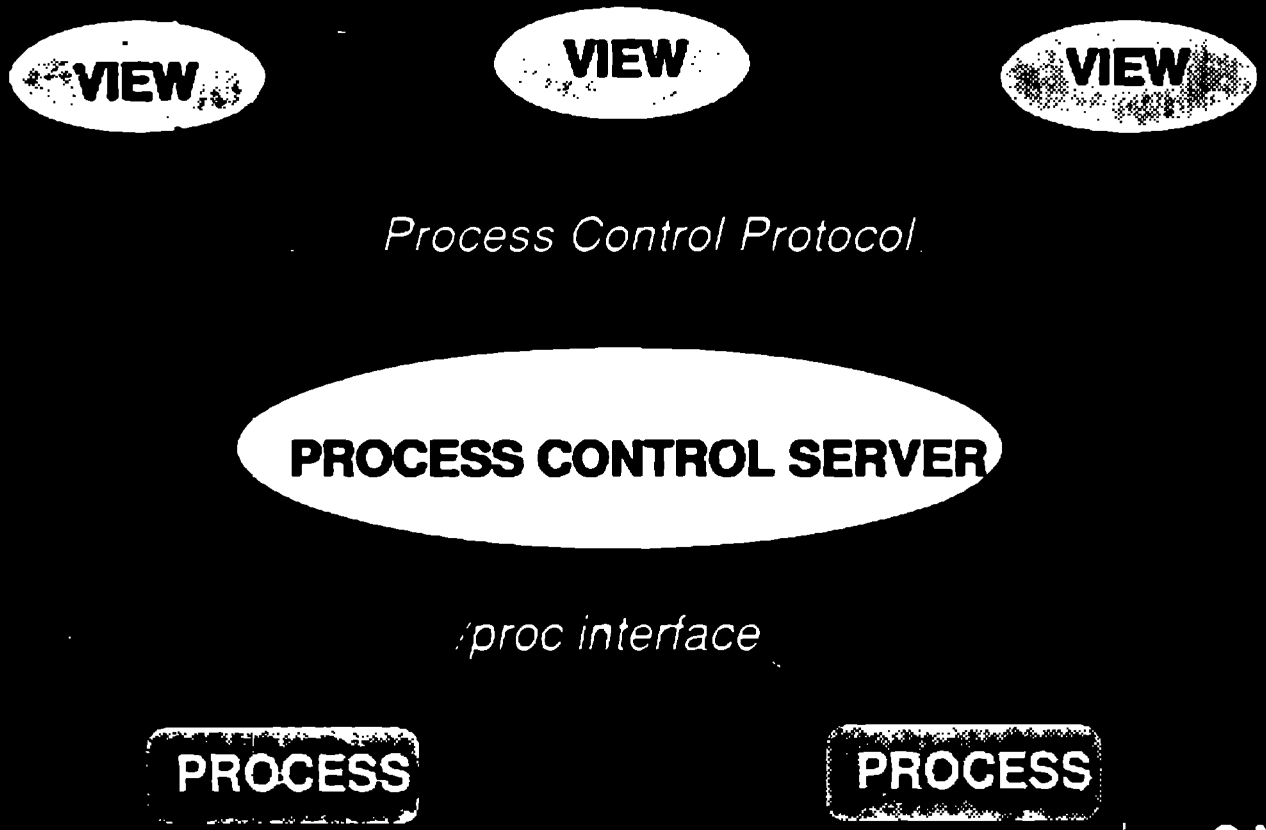
Current Technology

- Monolithic debugger process
- Single threaded
- Entire debugger on same host as target
- Minimal support for customization
- Cumbersome multiprocess model

SiliconGraphics

CASE
Products

A Distributed Architecture



VIEW

VIEW

VIEW

Process Control Protocol

PROCESS CONTROL SERVER

/proc interface

PROCESS

PROCESS

SiliconGraphics
The Developers Advantage ™

CASE
Products

Process Control Server (PCS)

Process Control

Stack frame construction

Trap handling

Symbol management

Expression evaluation

CASE
Products

SiliconGraphics
The Developer's Advantage™

Process Control Protocol (PCP)

Communication between PCS and Views

Operates on reliable byte stream (TCP/UDS)

Provides high level abstraction

Views

Display different "views" into process

Different views for different tasks

SiliconGraphics

CASE
Products

Advantages of Distribution (1)

Remote debugging

Supports more complex user interface

Multiple independent views for multiprocess debugging

Coordination of multiple target processes

Advantages of Distribution (2)

- End-user customizable user interface
- Porting abstraction layer
- Conference debugging
- Better utilization of MP hardware cycles

Challenges

- Performance
  - Protocol requests
  - Memory usage
  - Optimizing request granularity
  - Avoiding synchronous requests
- Appropriate distribution level
- Communication between views
- Race conditions

SiliconGraphics
The Developer's Advantage™

CASE
Products

Future Work

- Cross debugging
- Clean up and publish protocol
  - Custom end-user views
  - Porting to other architectures
- Remote (kernel) debugging nub
- Multithreaded PCS

Design of a Debugger for a Heterogeneous Distributed System

Arjun Khanna
Experimental Systems Lab
MCC, Austin, Texas

1.0 Introduction

This paper describes the design of DESK, a debugger for a distributed, object-oriented, heterogeneous operating system being designed as part of the Experimental Systems Project (ESP) at MCC. Some of the problems of debugging in the ESP environment are discussed. Requirements for symbolic debugging in a distributed environment are developed and a framework for accommodating heterogeneity in a transparent manner is presented. The research described in this paper builds on earlier work done by Hahn on the requirements for a debugger for ES-Kit[1].

A brief review of the ESP environment follows in Section 2. Section 3 develops the motivation for this work. In Section 4 we specify the basic framework of the debugger. Contributions of this research are listed in Section 5. Finally, Section 6 concludes the paper by proposing some extensions to ongoing research.

2.0 The ESP Environment

The ESP environment is a distributed heterogeneous system that encapsulates an object-oriented paradigm toward application programming. Currently, it supports the C++ programming language. Several ESP applications have been successfully demonstrated on a network consisting of Sun3, Sparc, Motorola 88000 based ES-Kit application accelerator, and most recently the Motorola Delta hardware.

The ESP environment consists of a minimal kernel, a copy of which resides on each node of the distributed system, and operating system facilities encapsulated as a set of Public Service Objects that are linked to the kernel on a demand basis. The kernel facilitates context switching, exception handling, message passing, and lazy evaluation of return values allowing concurrent execution of objects. Application level objects are distributed across the network and communicate solely through messages.

3.0 Motivation for this Work

Two significant concerns provide the motivation for this work. They are: 1)The need for a distributed object-oriented debugger and 2)Designing a debugger that will operate in a heterogeneous environment. A brief discussion of these two requirements follows:

- **Distributed Object-Oriented Debugger.** It is possible to use a sequential debugger like gdb (dbx with extensions to support C++) to debug the ESP kernel under UNIX. But gdb is not object-oriented in the same paradigm as ESP. For example, in gdb breakpoints may be set only on a class basis. The inability to set breakpoints on a per instance basis is a severe limitation in an environment where potentially a large number of instances share class code and breakpoints need to be set only in a subset of the instances[1]. Moreover, gdb does not offer features such as replay, distributed breakpoints, message logging etc., that are essential for distributed debugging.
- **HeterogeneityConcerns.** The requirement of accommodating several heterogeneous platforms in ESP has made us look afresh at several basic issues. Namely:

  - Defining the minimal functionality required of a distributed debugger.

- Evaluating the pros and cons of an approach in which various components of the debugger are dynamically configured vis-a-vis being statically linked into a large monolithic program.

- Specifying the interfaces between the various components of the debugger. A significant design concern is identifying the level at which exception handling, symbol table formats and architectural dependencies should be encapsulated in order not to compromise the extensibility and portability of the debugger.

- Identifying the functionality in a distributed debugger that should be centralized vis-a-vis being distributed.

4.0 Partitioning the Debugger

We propose splitting the debugger into the following components based on the functionality that is critical for a distributed debugger targeted to a heterogeneous environment.

- Front end. The front end offers a user friendly, architecture independent view of the distributed system.
- Symbol manager. The symbol manager handles functionality which is difficult to distribute. The main task of the symbol manager is to manage type and symbol table information. Additionally, the symbol manager tracks the nodes on which an application is distributed.
- Debugger Public Service Object (DPSO). The DPSO is an object that provides the interface between the symbol manager and the back end. One of the main tasks assigned to the DPSO is to translate commands from the symbol manager to simple back end routines to access application memory. Instruction decoding and machine dependent stack frame manipulation are encapsulated in this layer.

  An instance of a DPSO would be necessary on each node of a distributed system that needs to be debugged (this is obvious since the DPSO provides the necessary interface to access memory on a given node). Thus the DPSO forms the distributed component of the debugger. Global halting, distributed breakpoints etc., are some of the concerns that may be appropriately handled by this layer.
- The back end. The major concerns at this level are: 1)Breakpoint management. This includes setting and swapping breakpoints as well as maintaining breakpoint tables. 2) Interfacing with the DPSO to read/write memory and set breakpoints. Processor level functionality, such as reading/writing to control registers is handled by the exception handling facility provided by the ESP kernel.

  It should be noted that the machine dependent portions of the debugger are restricted to the DPSO and the exception handling facilities available as part of the ESP kernel.

5.0 Contributions of this Research

While some work exists on the requirements for a symbolic debugger for C++, DESK proposes solutions for several concerns typical of a heterogeneous, distributed C++ environment. These include:

- Techniques to handle per instance breakpoints as well as the ability to set breakpoints in the *constructor* for an instance (a constructor is function call to create a new instance of a class). Inserting breakpoints in constructors is complicated since the ESP kernel uses run time heuristics to allocate objects on the nodes of a distributed system. Thus, there is no apriori knowledge of the fact where an instance will be constructed.
- Perhaps the most significant contribution of this research is expected to be the specification of the interfaces between the various components of the debugger in a machine independent manner. For example, the interface between the DPSO and the symbol manager is independent of the machine type for which the DPSO has been compiled. Similarly, the symbol manager is targeted to work with several object file formats (for e.g. a.out, COFF, ELF, etc.). The philosophy of separation of mechanism (i.e., object file format and technique of loading) from policy (it should be possible to add symbol tables dynamically) has a distinct advantage over a monolithic debugger that must be rewritten each time the object file format is changed.

- Finally, DESK is not statically linked into the ESP kernel. Instead it is dynamically configured when a user wishes to debug an application. For each network node used by an application, the symbol manager uses configuration information to match the node address with a property list associated with that node (processor type is one of the properties). Based on this information, the symbol manager is able to start the appropriate DPSO on each node being used by an application.

6.0 Conclusions

In this paper we have described the design of a debugger for a heterogeneous, distributed object-oriented environment. We have made a case for:

- Designing an extensible debugger with a minimal core. Additional functionality should be configured to the core at run time.
- Defining clean interfaces between the various layers of the debugger so as to minimize architectural and object file format dependencies in the debugger.
- Incorporating relevant issues involved in the design of a debugger for a distributed C++ environment.

  The philosophy of keeping the debugger open ended is significant. We feel that features such as replay, scheduling control, and more advanced assertion checkers should be viewed as handlers that may be incrementally added to DESK.

REFERENCES

[1] Hahn, Douglas., "Debugging in the ES-Kit Environment", MCC Technical Report (ACA-ESP-063 89), MCC Non-Confidential.

Supercomputing '91 Debugging Workshop

Debugging in a Loosely Coupled Heterogeneous Computing Environment: A Case Study

Superconcurrency Research Team

Navel Ocean Systems Center

Code 421, San Diego

Superconcurrency

Objectives

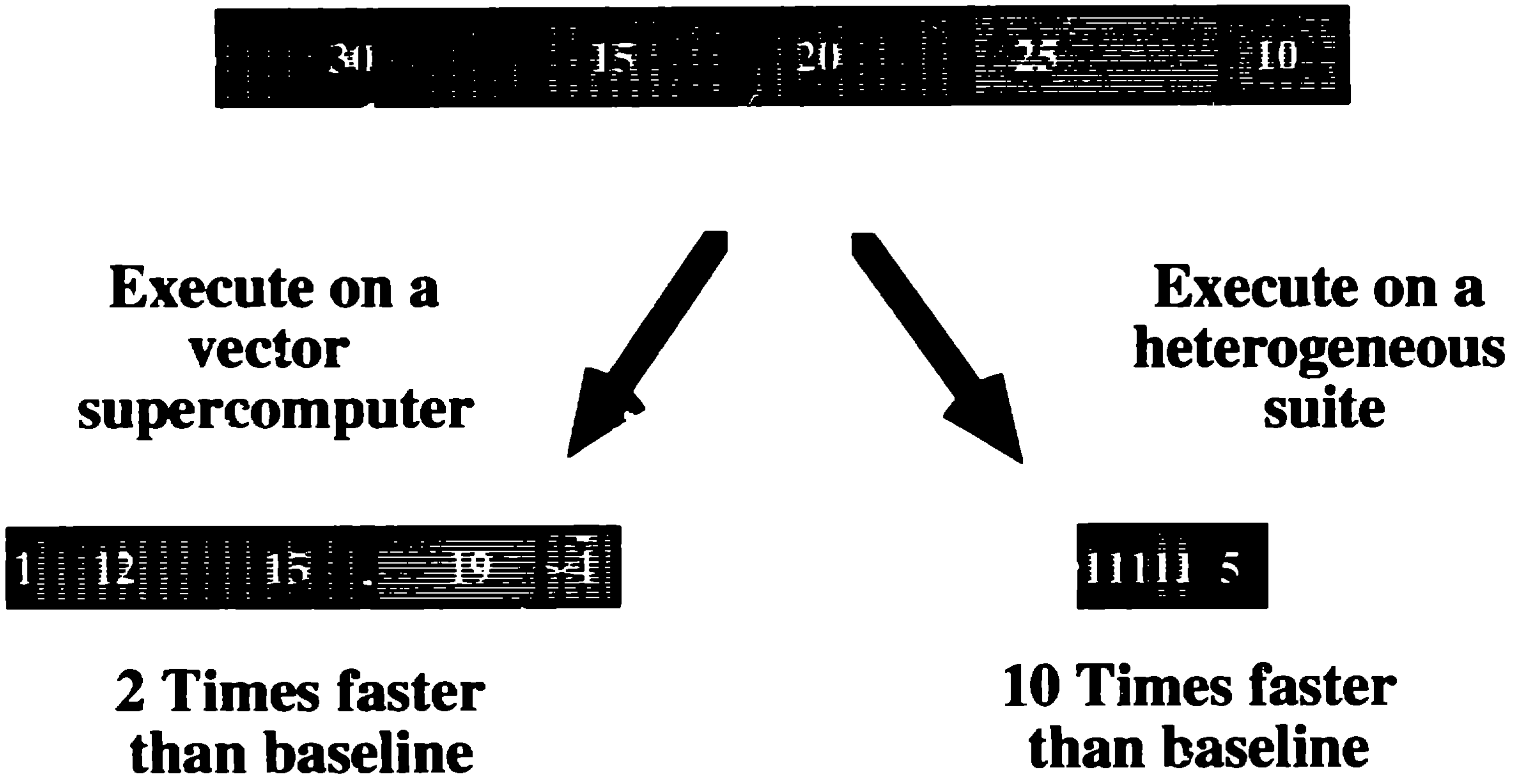
- Improve cost/performance ratio
- Reduce programming effort

Approach

- Match codes and algorithms to best suited architecture
- Intelligently manage a selected superconcurrent suite

Profiling and Benchmarking

Baseline Application



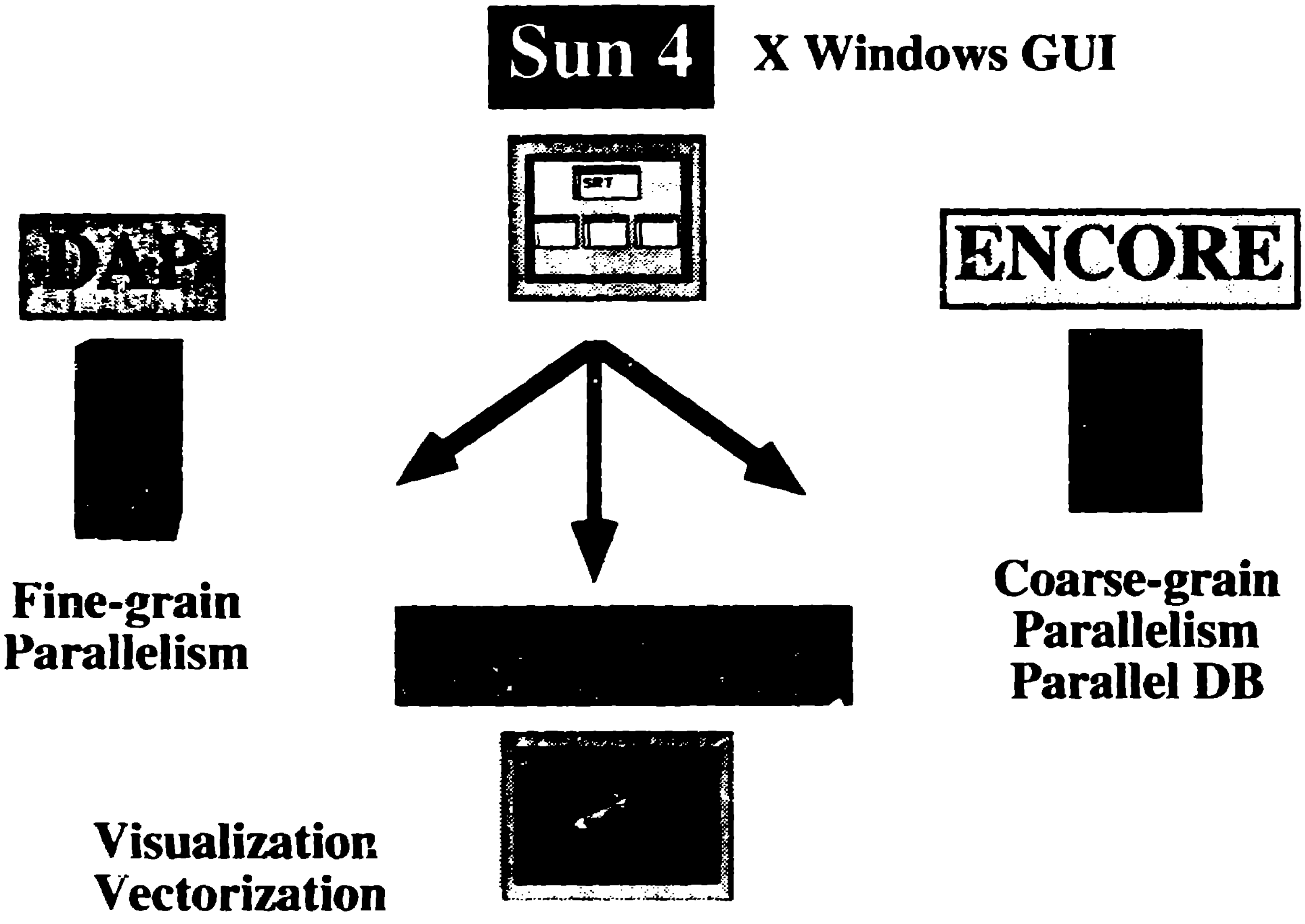
Execute on a vector supercomputer

Execute on a heterogeneous suite

2 Times faster than baseline

10 Times faster than baseline

Heterogeneous Processing Suite



Sun 4

X Windows GUI

DAP

ENCORE

Fine-grain Parallelism

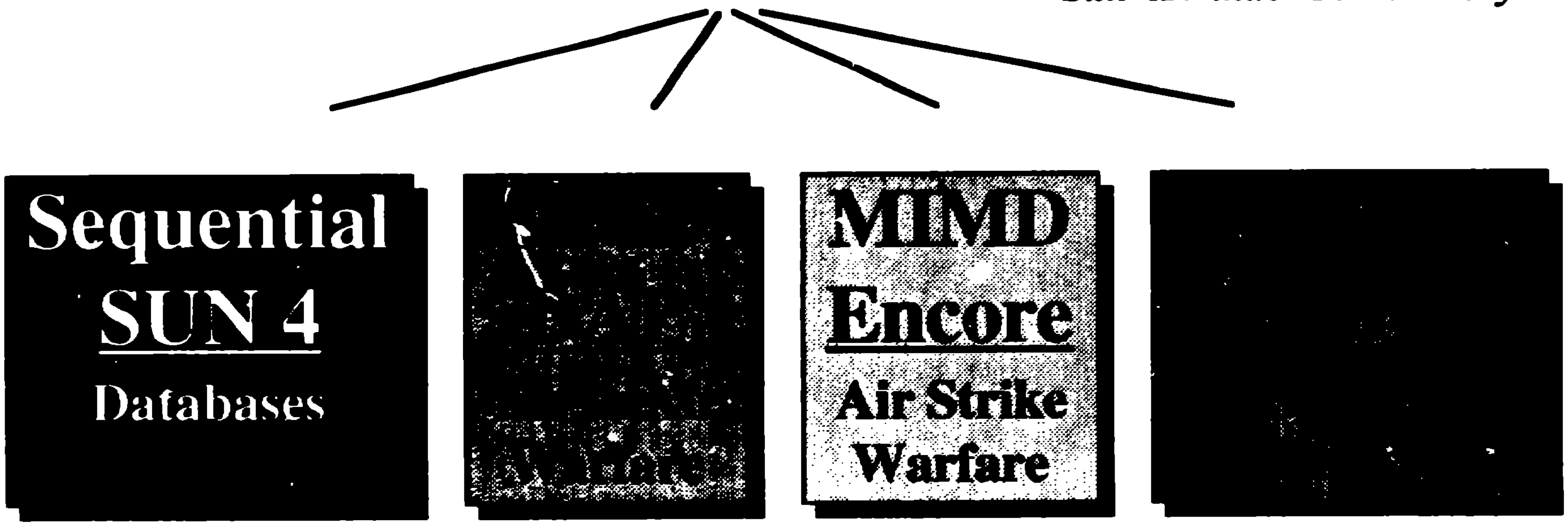
Coarse-grain Parallelism
Parallel DB

Visualization
Vectorization

CASES

Capabilites Assessment Expert System

Sun XWindows /MacIvory



Sequential
SUN 4
Databases

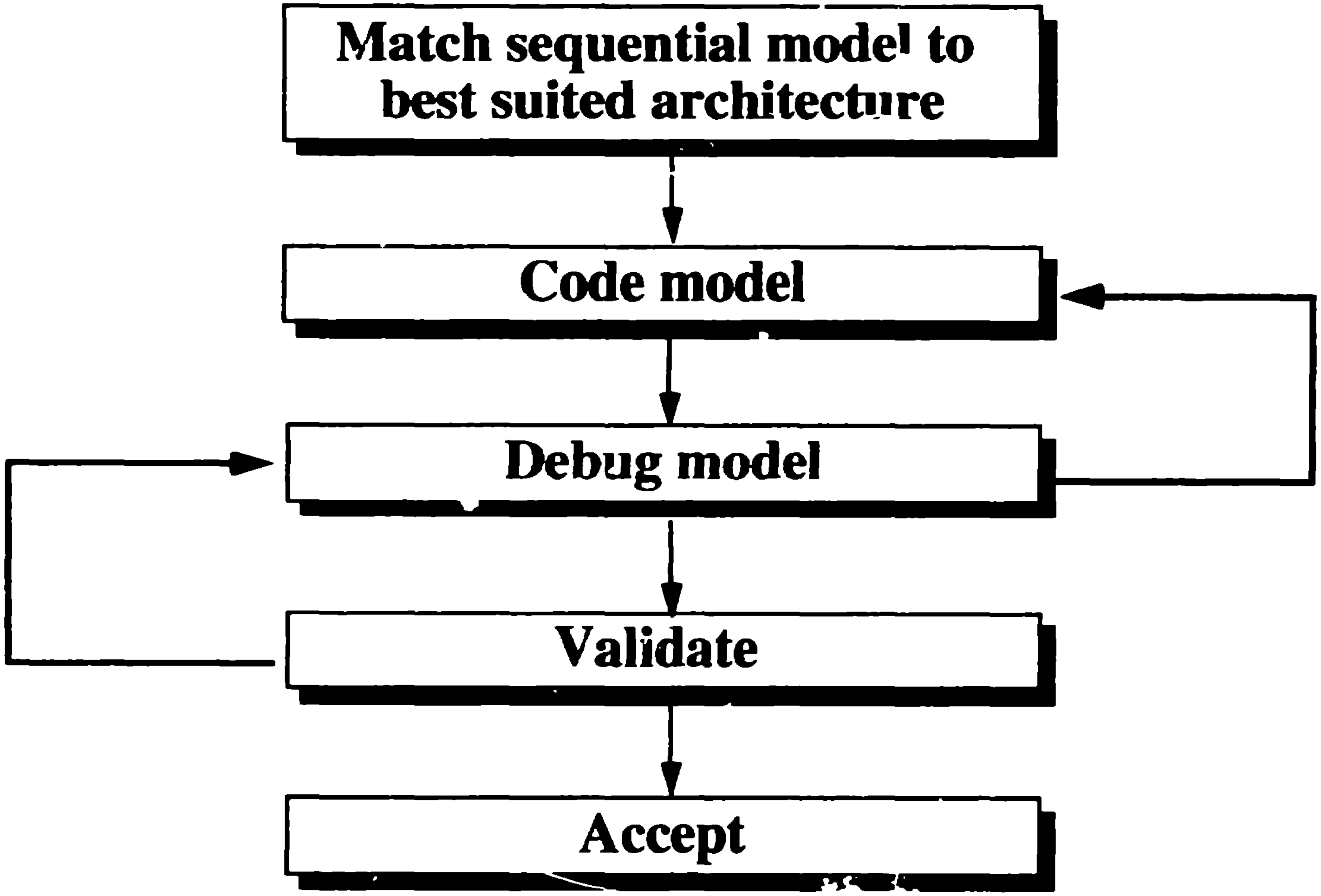
[illegible]

MIMD
Encore
Air Strike
Warfare

[illegible]

Campaign Simulation Models

Heterogeneous Coding Process



Match sequential model to best suited architecture

Code model

Debug model

Validate

Accept

Current Debuggers (SRT)

Name	Architecture	Machine
DBX	Scalar	SPARC
CMDBX	SIMD	CM2
CDB	MIMD	Multimax
PASM	SIMD	DAP
DBG	Vector	Titan
DEBUG	Scalar	VAX

Present Debugging

- Most development time spent debugging

- Software debugging tools follow hardware development

- Lack of support for simultaneous debugging of multiple processes

- Inability to specify array areas

Heterogeneous Debugging Issues

- **Official Standard**
  Dbx/gdb vs ???

- **Architecture Transparency**
  Scalar vs vector vs SIMD vs MIMD

- **Centralized Control**
  Central control of multiple threads over heterogenous network

- **Automated Verification**
  VMS vs UNIX, scalar vs parallel

- **Graphical Representation**
  Large data sets and process execution, ie CM2

Future of Heterogeneous Debugging

<u>**Parasight**</u>

- **Real-time nonintrusive parallel debugging and profiling**
- **User controlled program instrumentation**
- **User configurable parasite processes**
  - **Call tree analysis - "paragraph"**
  - **Thread tracking scoreboard**
  - **C and Fortran interpreters**
- **Dynamic code recompilation and insertion**
- **X Window/Motif interface**
- **Program interface for custom parasite creation**

<u>**Parallel Virtual Machine (PVM) /Heterogeneous Network Computing Environment (HENCE)**</u>

- **Heterogeneity and portability**
- **Tools for running, debugging, and analyzing programs on heterogeneous network**

Contacts

Superconcurrency Research Team

NOSC, Code 423

San Diego, CA 92152-5000

Chief Scientist:
Richard F. Freund
(619) 553-4071
RFFREUND@NOSC.MIL

Project Administrator:
D. Sunny Conwell
(619) 553-3994
CONWELL@NOSC.MIL

Scientists:
Mark Campbell
Francis Chiu
Laura Garbacz
Mike Gherrity
Javier Gudino
Kevin Kumferman
Matthew Kussow
Doug Sylliaasen

Lab: (619) 553-5322 FAX: (6 19) 553-5793

OVERVIEW OF DEBUGGERS USED BY SRT

Debugger:	DBX
Architecture:	Scalar
Machine:	SPARC
Company:	Sun
Languages:	C, FORTRAN

DBX is our standard scalar debugger. Its capabilities include setting conditional breakpoints, single stepping code, viewing and changing the values of variables, displaying variable type, and providing postmortem dump analysis. It has window interfaces for running in X (xdbx) and Sunview (dbxtool). Some of it's annoying features include the inability to conveniently specify areas of arrays, and the requirement of using C like syntax while debugging FORTRAN code.

Debugger:	PASM (Program State Analysis Mode)
Architecture:	SIMD
Machine:	DAP
Company:	Active Memory Technology
Languages:	FORTRAN Plus Enhanced
Usage:	Use -g compiler option, and include a "Pause" statement.

PASM, which stands for Program State Analysis Mode, has been useable in it's current form for about a year. It has the basic capability to examine variables (including registers and stacks), single-step through program code, set breakpoints, and resume execution. It uses its own unique command syntax. It has an easy to use syntax for displaying portions of mutlidimensional arrays, which is an important feature for a SIMD debugger. Some of the features the debugger lacks include the ability to change the value of variables, trace procedure calls, and rerun code from within the debugger.

Debugger:	CDB
Architecture:	MIMD
Machine:	MultiMax
Company:	Encore Computer Corporation
Languages:	C, FORTRAN, Encore Parallel FORTRAN (EPF)
Usage:	Use -g compiler option

CDB is a standard Unix like debugger with some additional features for handling multiple processes. It provides all of the functionality of dbx, and even includes some identical commands, but in general has a unique command syntax. The major enhancements for parallel debugging include the ability to set both global and process specific breakpoints, and to send commands (including those to continue running, single step, or print variable values) to either single or multiple processes.

Debugger:	CMDBX
Architecture:	SIMD
Machine:	Connection Machine
Company:	Thinking Machines
Languages:	CM-FORTRAN
Usage:	Use -g compiler option

CMDBX provides all of the functionality of dbx and also uses dbx syntax. It has extended commands to allow the printing of array areas. These extensions also apply to dbx expressions for example allowing like sized areas of two different arrays to be multiplyed together and displayed. One nice feature of CMDBX is that it accepts a more FORTRAN like syntax for example array references can be specified with parenthesis FORTRAN like instead of needing to use C brackets.

The CM uses the standard dbx debugger with some custom extensions for debugging C* code. The extensions mainly deal with defining "regions" of processors to look at, and simplify printing the values of parallel variables. When programming the CM in *LISP the debugging environment is integrated smoothly into the standard LISP debugging capabilities.