A NEW PARADIGM FOR COMPUTER USERS

by

Milton C. Wikstrom and John L. Gustafson
Scalable Computing Laboratory
Applied Mathematical Sciences

Ames Laboratory*
Iowa State University
Ames, Iowa 50011-3020

Date Transmitted: July 1995

*Operated by Iowa State University for
the U. S. Department of Energy under Contract
No. W-7405-ENG-82.
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.
A NEW PARADIGM FOR COMPUTER USERS

by

Milton C. Wikstrom and John L. Gustafson
Scalable Computing Laboratory
Applied Mathematical Sciences

Ames Laboratory*
Iowa State University
Ames, Iowa 50011-3020

Date Transmitted: July 1995

* Operated by Iowa State University for
the U. S. Department of Energy under Contract
No. W-7405-ENG-82.
Contents

Abstract

1 Introduction

2 Background and Motivation
   2.1 History of Entanglements
   2.2 Failures of the Current Approach
      2.2.1 Uncertain Accuracy
      2.2.2 Loop Unrolling
      2.2.3 Word Size

3 Participants
   3.1 End User
   3.2 Application Program
   3.3 The Compiler
   3.4 Hardware Specifications

4 Contracts
   4.1 End User to Application Program
   4.2 Application Programmer to Compiler
   4.3 Compiler to Hardware Contract

5 Conclusions and Future Work
   Authors’ Note
   References
   Distribution List
ABSTRACT

Why is it that mathematical results are useful for centuries, yet computer programs suffer tumultuous changes of technology? Can one write programs that will be useful in fifty years? Extreme efforts to preserve “dusty deck” programs have confused the roles and responsibilities of four participants: the end user, application programmer, system software designer, and hardware engineer. Users feel they must know hardware details like word size and megahertz rate — yet are oblivious to error in the output. Application programmers try to guess compiler behavior, often obtaining high performance only by trickery. With new hardware, that old guess is often wrong. The result is “high-performance code” that is slower than the original code. The hardware engineer tries to maintain compatibility through many generations, sacrificing performance and hardware affordability to meet presumed needs of the other three participants. We propose that the respective roles and responsibilities be clearly delineated via a three-stage contract between the four participants. The result is a new interaction paradigm that should allow programs to share the longevity of mathematical results and absorb technical advances gracefully.
1 Introduction

There have always been four identifiable groups, or participants that use computers: the end user, the application programmer, the system software designer (including the compilers, libraries, and operating systems), and the hardware engineer. In many scientific environments, the end user and the application programmer are the same person; the end user is also expected to be the author and maintainer of the source code. In other environments, such as business computing and personal computing, the end users are distinct from and far more numerous than the application programmers.

But mostly because of performance motivations, there has been a trend to blur roles and responsibilities for one another’s traditional tasks. The tendency to “pass the buck” between the four participants regarding who should make performance higher has led to problems that were never anticipated or intended by the computing community. Part of our solution will be to redefine and disentangle the roles using definitions given in Section 3. We propose that the roles of each participant be made explicit via a three-stage contract, described in Section 4, as an approach to solving these problems. Section 4 also steps the reader through a simple but nontrivial example. This is followed by a section containing concluding remarks, along with our long-term vision of future research. To summarize, this technical report presents the problems with the current use of computers that have arisen, why they have arisen, and what might be done to solve them.

2 Background and Motivation

2.1 History of Entanglements

In recent years, the roles of the four classes of computer participants have become confused and entangled to the detriment of everyone. These four types of participants are: the end user, application programmer, system software designer, and hardware engineer. A watershed was in 1975, when FPS successfully commercialized a wide-instruction word computer, the AP-120B [3]. That computer shifted the burden of raising performance from the hardware engineer to the designers of subroutine libraries that did “software pipelining.” It also put a burden on the application programmer, who had to rewrite applications in terms of those library routines. The CRAY-1 [18], introduced in 1977, forced the issue of “vectorization” on disgruntled application programmers. Those who accepted the new burden were (sometimes) rewarded with much higher performance, at the price of having to learn a new set of skills, reduced portability, and increased code development time. By about 1982, “parallel processing” was beginning to be debated as the unavoidable next advance in computer architecture [2, 6, 8, 10, 11].
Parallel processing has been with us for many years. Computer architectures of the 1950’s had parallel functions (for example, the IBM 7094 [14] fetched data and instructions in parallel), and even the use of n-bit words implies $O(n)$ parallelism at the finest grain. Yet “parallel processing” is treated as a recent revolution in computing. Why?

The difference now is that the application programmer is being asked to recognize parallelism and make use of it, whereas in the past the parallelism had always been the province of the hardware engineer, or beginning in the 1980’s, the compiler designer [1, 7, 9, 16, 17, 20]. As the term is used today, “parallel processing” means only that type of parallelism that cannot be disguised from the application programmer. The need for the application programmer to know so much about computer architecture is viewed by some as a major calamity, and the cost of rewriting the serial software in existence is enormous. Even the end user is no longer insulated from changes in computer architecture. In parallel computing environments, users are expected to specify how many processors they need, for example.

A popular school of thought is that we must find ways to exploit parallelism at the application level that are gradual and painless. Others feel that only revolutionary change is the solution and that we must discard old software and start over with some assumption about what “all” future computers will have in common. There is no agreement about what that future will be and the result is paralysis.

The debate is often couched in terms of computer languages. Those who oppose all changes at the application level want to lock in existing definitions of languages like C and Fortran. Others look to extensions to languages by compiler directives and new keywords, asking the application programmers and compiler designers to do the “dirty work.” The High Performance Fortran (HPF) [13] initiative belongs to this category. Still other want to invent entirely new languages like Sisal or Occam, that can express parallelism or data flow, requiring a complete rewrite of every application.

It is not our intent to offer, in this technical report, a new language definition. We see the choice of computer language as neither the problem nor the solution. The problem lies in the correct assigning of responsibility for tasks among the four participants. Computer languages are inherently ephemeral. Consider Knuth’s series *The Art of Computer Programming* [12]. His “assembly language” examples written in MIX are of little value today; however, the English and mathematical examples are still relevant and useful. A combination of English and standard mathematical notation to delineate tasks and responsibilities seems likely to remain readable over many decades of technological change and is the focus of this technical report.

2.2 Failures of the Current Approach

In this section, we provide three examples illustrating the failure of the currently-used programming paradigm. The first example deals with assumptions by certain system software designers regarding end user input accuracy and output error. The second example illustrates how low performance can result when the application programmer tries to guess the compiler behavior. The last example notes how personal computer purchasers (end users) often use irrelevant hardware
criteria in making their purchase decisions.

2.2.1 Uncertain Accuracy

In the 1980's, the DEC and Unix C libraries for trigonometric functions, assumed that the end user's input is exact. Because a number like $10^{300}$ expressed to fifteen decimals is really uncertain to $\pm 10^{285}$, it is absurd to insist that the sine or cosine of that number be exact. Who knows where, on the unit circle, the angle really lies? But without permission from the end user to treat the input as approximate, the only way to unwind the number around the circle accurately is to use multiple-precision arithmetic with $\pi$ stored to thousands of decimals! The resulting performance, needless to say, is abysmal. The fault lies in the misplacement of responsibility in the implicit contract between the end user and the application programmer. A concession (by the end user) of inaccurate input can greatly improve performance that one can reasonably demand.

2.2.2 Loop Unrolling

In the 1980s' LINPACK benchmark report [4], there is the annotation "(rolled BLAS)" after many of the numbers in the table. What does this cryptic comment mean? Well, the BLAS are Basic Linear Algebra Subroutines [5] like dot product and vector scaling for which the LINPACK benchmark supplies Fortran subroutines. The loop operation $y_i = a \cdot x_i + y_i$ is given two pages of Fortran with the loop unrolled 2-fold, 4-fold, 8-fold, and 16-fold. A portion of this code is given in Figure 1. Unrolling is done to avoid the overhead of loop management. In more recent machines and compilers, there is a higher performance penalty associated with unrolled code than there is with rolled code. So people "roll" the unrolled loop back up. Then LINPACK goes faster.

What went wrong in these events? The application programmer was guessing both compiler and hardware behavior. What the programmer wanted to accomplish was to multiply a row of a matrix by a constant and then to add the result to another row. Conventional Fortran forces the programmer to write a "DO" loop to perform this sequentially, even though the order of the $i$ subscripts does not matter. There is plenty of parallelism in the operation and opportunity for vector arithmetic also. Compilers for vector and parallel computers have to do some extra work figuring out if the serial ordering the programmer specified is safe to ignore. All the unrolling does is make that analysis more difficult.

2.2.3 Word Size

In 1993, one could eavesdrop on 10-year-olds debating the merits of 8-bit versus 16-bit Nintendo™ video game systems. One wonders whether those arguers had much understanding of register sizes, data bus widths, and the technical merits and costs. It seems more likely that the number 16 was considered to be twice as good as the number 8.

Long ago, one of the authors witnessed a marketing executive insisting he needed a 16-bit personal computer to do word processing, because an 8-bit PC simply would not do the job.
When asked to explain, the depth of understanding of 8-bit versus 16-bit proved no different than that of your average 10-year old.

Why are computer customers concerned only with hardware specifications instead of actual delivered performance? Similarly, the megahertz rating is often cited as though it were a sure indicator of application speed. The point of this example is that end users have involved themselves in hardware choices that are only weakly correlated to their actual needs.

3 Participants

In much the way that a building is constructed through contracts, contractors, bids, materials, and labor, a computer application may be thought of in a similar light with the final product being desired information. In this section we concentrate on the four participants involved with executing a computer application: the end user, the application program, the system software, and the hardware specification file. Of course, the application program, system software, and hardware specification file are created by application programmers, system software designers, and hardware engineers, respectively. But when the contract between the end user and application program is being negotiated, it is the software entities that are participating, not the people. The only human that is directly involved is the end user. After the participants have been described, we describe in Section 4 the interactions between the four participants in terms of the three-stage contract.
3.1 End User

An end user is a person wishing to execute some application but is not assumed to know or care about the architecture, topology, or any machine-specific details. Hence, the end user should only have to specify the task in terms of goals. Perhaps more importantly, the end user should not be allowed to specify anything but the goals. The primary goal of the end user is to obtain “accurate answers” in “a reasonable time” and at “a reasonable cost”. In practice, we observe two disturbing trends away from this common goal: a naive disregard for accuracy issues and focus on extraneous machine-dependent details instead of on time and cost.

Typically, when an end user executes a numerical application, end user assumes that the output accuracy is sufficient to meet his needs. On the few occasions when he feels uncomfortable with the answer, the common response is to switch from single precision representation to double precision. In fact, most end users of applications have no idea how accurate their output really is. Even in the cases where an output error estimate is supplied by an application, the output error is calculated under the assumption that the input is exact and error-free.

We propose that the end user focus on accuracy, elapsed time, cost, and reliability (tolerance for machine/software failure rates) by passing a set of demands and concessions to the application program\(^1\). For example, the end user can say, “I need three digits of accuracy in my answer.” However, the end user should not be allowed to say, “I want you to use the ‘double precision’ data type because on this computer I think it will be good enough. I do not trust single precision.”

Demands that an end user can specify include placing upper limits on:

- how long the job may run in wall clock time
- error in the output
- cost (preferably in real money)
- acceptable failure rates

Most paradigms involve demands alone. However, it is important that concessions be included too. Placing demands without concessions allows one to make wild and unrealistic requirements without consideration to their cost or feasibility. In terms of the construction analogy, an example would be asking for a mile high skyscraper in a day at no cost. Obviously, this would be impossible without conceding several points. Suppose one said that they would like the same skyscraper but would be willing to settle for a building of 80 floors in 5 years in exchange for $200,000,000. This is a contract that at least has a fighting chance. So accordingly, here are some concessions that an end user can give:

- a worst-case estimate of error in the input data
- a willingness to tolerate faults in hardware or system software

\(^1\)Depending on your point of view, virtually any attribute may be thought of in terms of demands or concessions. The point is that the end user must be willing to endure certain penalties or forfeit use of the computer.
• a willingness to pay for a run
• a willingness to wait for a run
• a list of lower priority goals that may be sacrificed (if necessary) in order to meet other more important goals

Of the demands and concessions listed above, the three most important may be specifying a maximum of wall clock time and output error while admitting the amount of input error. Yet our generation of end users is accustomed to having little control over these issues. Scientific simulations are typically done with inputs assumed to be exact, very unscientific confidence in the numerical validity of the answers, and execution times determined by trial-and-error for each system with no guarantees of repeatability.

It seems possible to create a first-stage contract\textsuperscript{2} between the end user and the application program that demands a minimum performance for execution time and answer quality, yet frees the application programmer to achieve these things by any method that works. Note that the acknowledgment of possible error in the input is a concession made by the end user that helps the application programmer. This subcontract makes up the first stage of the three-stage contract.

3.2 Application Program

The next participant is the application program. Currently, when an application programmer develops an algorithm, he or she passes to the compiler a great detail of information which is not only irrelevant, but is also damaging to the efforts of obtaining good performance. For example, in conventional languages like C or Fortran, the sequential ordering of instructions usually results in unnecessary and spurious data dependencies from one instruction to the next. Also, application programmers are notorious for introducing extraneous details like loop unrolling (Section 2.2.2) in order to obtain better performance. Later, when a computer is replaced by a newer computer, the relative performance of the same program suffers as a result of these extra details.

We believe that “an application program” should actually be a suite of many different, best-known algorithms that cover a wide range situations. For example, some algorithms would be targeted to a SIMD machine with a two-dimensional mesh. In addition, separate algorithms would exist depending on bits of precision, register availability, and the number of floating point adders. We propose that the algorithms be written in an English-like format with dependencies, suggestions, and shortcuts stated explicitly.

As the application program is downloaded to a particular machine, it should be sent to the system software for extraction of those algorithms suited to the particular hardware. At this point, much of the software it omitted due to incompatible topological or timing or precision requirements. However, the subset of algorithms that remain can confidently believed to be the currently “best known” approaches to meeting certain goals within the concession/requirements for this particular machine.

\textsuperscript{2}Henceforth, referred to only as the “contract” when the context makes it clear which stage is applicable.
After the remaining algorithms have been compiled, they would be individually tagged by the system software with timing and error estimates. When an end user submits a contract, it can be processed at submission time against this tagged information to locate the appropriate algorithm (if one exists at all) or to determine that the contract is not feasible. When a contract is not acceptable, alternative contract stipulations should be offered to the end user for consideration.

So what should an application program legitimately specify in its “contract”? The answer is: whatever it needs from the compiler to satisfy the goals set by the end user. The application divides the end user’s goals into subtasks, which sounds like top-down design but is something very different. The end user’s goals involve time and accuracy. The application program specifies an algorithm, with guaranteed performance at every step. That algorithm can be constructed by top-down, bottom-up, or whatever method one likes, so long as the final result can be read by a human as a proof of the achievability of the overall goal. The best way is by example and is given in Section 4.2.

There are two important features in this particular part of the contract that are not found in other approaches. One is to place a bound on the speed, latency, and precision of every data item. The second is to express as much parallelism as possible, understanding that there is generally more parallelism in the data dependency of an algorithm than can be exploited by a compiler. The compiler’s task is now discarding parallelism instead of trying to derive it.

These two details are the ones compilers are currently burdened with guessing about, and programmers have avoided specifying these things. So traditional programming languages tell the compiler extraneous information that it must work to ignore (like spurious sequential ordering of operations) and deny it essential information that it must work to guess or derive (e.g. which variables belong in registers).

Another useful piece of information the application programmer should supply is branch probability. Any time a conditional statement is made, some indication should be given about which way it goes most of the time, or that no such prediction can be made. Similarly, minimum, average, and maximum expected passes through a loop should be specified. Then, in addition to maximum execution time, average execution time could also be predicted. This example illustrates why this paradigm is new. Current paradigms insist that the compiler or other tools somehow derive these probabilities and bounds. It is little burden for the application programmer to specify this information. Often, it is the case that the programmer knows this information far better than the tools will ever be able to derive.

If the application programmer declines to supply this information, then the compiler is not obligated to be “clever” or go beyond what is patently explained in the program. The time for a sequence of operations is the sum of their times. The time for parallel operations varies with available parallelism in the hardware. The trade-off between porting to a wide number of computers and asking for the maximum performance is made explicit.
3.3 The Compiler

The third stage of the contract is between the compiler and the hardware. The compiler will examine, say, a block of application source code as well as its timing and precision requirements. It then consults a table that is specific to a computer system that describes the hardware: the number of functional units and the restriction on using them simultaneously, the size of each memory regime, and any other feature that must be known to establish timing and error estimates. Alternately, a microcode simulator can be used instead of a table, but it should still be able to give a maximum wall clock time and output error estimate for the conditions of the contract.

If the compiling of the block of an application program fails to meet performance requirements, then the compiler can:

- try alternatives, limited by the application programmer to compile within a certain time
- quickly give up and report where and why the problem occurred so that the end user can alter the terms of the contract or find a different host.

The compiler should not be burdened with \( \mathcal{NP} \)-hard optimization problems of resource allocation. If someone finds a clever solution to a common optimization problem, then that should be placed in the standard library of routines. That standard library can contain, say, all the best known methods for performing the \( y(i) = a \times x(i) + y(i) \) operation for the whole spectrum of computer architectures. That library function might be lengthy and complicated, but would be attached to a simple performance specification usable by the compiler to see if a contractual obligation to the application programmer can be met.

3.4 Hardware Specifications

The hardware specification file was already mention in the previous section. This file does not need to contain everything there is to know about the hardware. Rather, it should contain only the information necessary to the compiler for determining timing, cost, precision, range, and reliability requirements. Optionally, a hardware simulator could be used in place of the hardware specification file. Note that such hardware simulators already exist and are used by the manufacturer during the design phase of a new architecture. The simulator would be useful in many different situations. The first is to give an overall maximum wall clock time for a block of application code. Another is to select the fastest code among alternatives in an algorithm where machine dependent details matter.

For example, consider the quadratic equation. The denominator is the term ‘2a’. For many years, it has been standard practice in a situation like this to calculate the value \( 1/(2a) \) once and place it in a temporary variable. Then every time a division is called for, a multiplication by the inverse is used instead. The rationale is that a floating point multiplication is typically faster than a division. We believe that the application programmer should accommodate as many alternative hardware possibilities as is manageable.

Rather, this is a feature that can and should be tested at compile time and the application programmer should supply two versions of the code: one using the temporary variable and one
that does the division every time. Then after the compiler has done its speed test for division
versus multiplication, it can decide which alternative section of code to use. We can extend this
idea even further by having the compiler time both branches of code (i.e. an entire section of
code) and then select the quicker of the two.

In summary, we have shifted the burden off of the compiler and onto the application program-
mer. This is contrary to the prevailing academic goal. We believe this approach has the benefit
of reducing long term development costs - even though it is at the expense of greater short-term
cost.

4 Contracts

In this section we describe, in detail, our new three-stage contract paradigm that:

1. forces the end user to specify practical goals without being allowed to specify implementa-
dtion details
2. forces the application programmer to specify performance in the algorithm description with-
out being able to assume target hardware
3. enables the compiler to use a local hardware specification or microcode simulator to deter-
mine whether or not the application programmer's specifications are feasible - and if not,
to provide feedback and alternatives to the application programmer. The compiler is not
allowed to make clandestine changes to the code. This is a marked departure from current
programming methodology.

As an example, we provide a generic contract except in part of the second-stage contract where
specific contract information for solving a quadratic equation is provided. In the generic contract,
the notation is loosely based on the Backus-Naur Form (BNF) specification and is defined as
follows:

```markdown
## comment follows to end of line
itemlist required item or items
| alternation – pick one from this itemlist
[ ] enclosed optional item
```

We hope that all other notation is self-explanatory.

4.1 End User to Application Program

In this section, we discuss the first stage of the three-stage programming contract. Recall
that the first-stage contract is between the end user and an application program. Please refer to
Figure 2 throughout the following discussion.
TIME: {average | maximum | confidence n%} wall clock time) 
- n {seconds | minutes | hours | days} [excluding user interaction]

COST:
The total cost of this run is: 
- not to exceed n {dollars | units} when successful 
- not to exceed n {dollars | units} when not successful

NEGOTIATION:
if contract fails, the user [is | is not] willing to engage in a dialog

DATA REPRESENTATION:
{
  <variable-list>
  source: {user-supplied | computed}
  error: {exact | fixed error n | relative error on nonzero n%}
  numeric format: {{decimal | binary} digits | not applicable}
  ## specify digits of precision and absolute or relative error
  ## the number of placeholders can be varied by the end user
  [+|-]d.dd...d E[+|-]dd...d +/-d.dd...d E[+|-]dd...d [%] 
}*

GOAL: {speed | accuracy | thrift | reliability}

INTERFACE:
[ <input data variable-list> : <machine location> ]
<output data variable-list> : <machine location>
notifications : <machine location>

Figure 2: 1st Stage Contract Template
Time Clause

The time clause in the contract is to specify the maximum wall clock time that the end user is willing to wait for the completed execution of the application. The decision to use wall clock time instead of CPU time was not arbitrary. Under our new paradigm, the end user is insulated from hardware and system details. Therefore, it makes no sense to have the end user think about such things as time sharing, job swapping, and all the other events that cause CPU and wall clock times to differ. Rather, we view the end user as having little if any prerequisite computer science background; all that the end user cares about is reliable obtainment of accurate answers in a reasonable time and cost. Since human interaction can have a major impact on wall clock time especially when input is entered from the keyboard), we give the end user the option of excluding that interaction time in the wall clock time. We also give the option of specifying time as an average over many runs, a hard maximum, or a degree of confidence that it will finish within that time.

Cost Clause

In the cost clause, the end user states requirements for maximum cost for submitting a job. These costs are subdivided into two cases: a successful run and an unsuccessful run. These cases are analogous to other human contracts that specify a cost in case of failure to deliver, ranging from nothing to full cost.

Negotiation Clause

The negotiation clause represents the willingness to negotiate the contract. The application program contains the logic for assisting the user to relax requirements until they can be meet.

Data Representation Clause

The data representation clause of the contract specifies the source, type of error, and format of input and output data. There are two choices for source which are “user-supplied” and “computed”. User-supplied means that the data is input by user. Computed means that the value is output from the application program. This information is necessary for the computation of output error.

The term error applies to both input and output values. When an error of zero is explicitly demanded on either input or output for an numerical algorithms, the user incurs the sometimes high price of that demand. Otherwise, the error can be stated or calculated in either absolute or relative amounts. One way to calculate error is by use of interval arithmetic techniques [12, 15]. Ordinarily, error is applied only to numeric values. However, the term can be generalized. For example, a misspelled word in text may be thought of as a correctable and/or measurable error.

In the next subclause, the end user specifies the format of numeric values. The end user has the option of working in the binary or decimal number system. A wide variety of non-numeric formats can also exist. For now, we are choosing to ignore the variety of choices; however, in the future the non-numeric values may need to be developed.
Finally, the range, precision, and uncertainty of numeric values are specified by the user through a length adjustable, number template which is in exponential form. Absolute error is specified by omitting the trailing percent sign; relative error is specified with the percent sign.

All of the specifications in the data representation clause are applied to one or more variables specified in a variable list. Use of the clause may be repeated any arbitrary number of times for different variable lists.

**Goal Clause**

In the goal clause, the end user must state whether the primary goal is to reduce wall clock time, increase output accuracy, minimize cost, or to increase chances for a successful run. Since these goals are diametric to one another, this information is useful for selecting the correct algorithm to execute.

**Interface Clause**

Finally, the interface clause requires that the end user specify the source and destination for input data, output data, and side interactions with the system (such as error messages). In this respect, our contract paradigm is similar to that of UNIX™. This clause interacts with the time clause. If human interaction is selected, then no guarantees can be made regarding time.

Figure 3 illustrates the process that an end user experiences in submitting a job. In practice, the user will rarely have to go through such an elaborate process since defaults are associated with all contract terms. Nevertheless, contract terms are always available for perusal.

### 4.2 Application Programmer to Compiler

The second-stage contract negotiations are between the application programmer and the compiler. These negotiations occur while the algorithm suite is being developed. In many ways, this contract is similar to but much more detailed than the first-stage contract. The application program creates a set of machine independent algorithms. These algorithms are written in an English-like style and all data dependencies are explicitly stated. Included are a set of requirements, namely: speed, cost, accuracy, and reliability of the algorithm and hardware. These same requirements may also be applied to the compilation process itself.

The application programmer bases these estimates on estimates of a hypothetical machine. If the programmer demands too high a performance, all users will be satisfied, but no compiler will agree to the contract. If the programmer is too conservative, every compiler will agree to the terms, but end users’ demands will not be met. In this way we make explicit what is currently implicit. Note also that this allows the application programmer to track decades of hardware change simply by altering performance demands without making any changes to the set of algorithms.

The algorithms and requirements are then passed to the compiler. In addition to the usual syntactical checks and translation, the compiler also uses its knowledge about the target hardware to determine whether or not the application programmer’s performance requirements are feasible. If they are, the object code is generated and tagged with the various performance values. If the
End User

Submits request

Object Code

Operating System

Is contract OK and was object code created with the latest hardware and compiler versions?

Yes

Successful Run!

No

Contract Failure! Explain reasons and exit.

Is the user willing to negotiate terms?

Yes

Inform user of possible contract modifications likely to result in success.

No

Negotiate with the user over new contract terms.

Figure 3: Life cycle of a First Stage Contract Submission
requirements are not feasible, then the compiler rejects the contract and notifies the application programmer of the reasons for rejection.

In general, low effort by the application programmer will usually result in a failed contract. As the application programmer expends more effort in writing efficient code and in giving the compiler more information on how portions of the code should be compiled, the more likely a successful compilation will be.

Consider the quadratic equation as an example. One way to express this equation is as:

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

Suppose that the application programmer does not give the compiler any information about the placement of values in memory leading to a contract failure. The compiler might respond back by suggesting that the timing requirement be relaxed. Suppose that the application programmer decides that the timing requirement not be changed. Then extra effort must be expended to help the compiler find a way to do the same equation in less time. One way to achieve this would be for the application programmer to suggest to the compiler that the values of \( a \) and \( b \) be stored in registers. Another suggestion to the compiler could be to calculate \( b^2 \) as \( b \times b \) instead of by invoking a power function. Yet another suggestion would be to inform the compiler of the various parts of the equation to calculate in parallel.

The point is that we expect the application programmer to do most of the "dirty work"—not the compiler. As more and more details and suggestions are given to the compiler, the more tenable the contract becomes.

In the Second-Stage contract (see Figure 4), the following items included:

- variable latency hierarchy
- the application algorithm including looping probabilities and loop bounds
- data dependencies

The application program should give the compiler information of true utility. Included is a section containing suggestions for variable placements. Currently, we see two different approaches in which this can be done. The first is called the relative memory distance approach. In this approach, variables are ranked from fastest access needs down to those of slowest access needs. This is done by declaring levels so that variables of like access speed can be grouped together under a common level.

The second approach to specifying variable latencies is in terms of absolute distance as measured by elapsed wall clock time for access. For certain kinds of applications, this approach is more useful as it places definite, real-time limits on the system. However, this very precision also makes this approach more tied to current technological levels in its specification without anticipation of future improvements. We believe that this information could be changed from a suggestion to a requirement, if need be, but could be counter productive to overall speed obtainment.
VARIABLE ACCESS PRIORITIES

--Relative memory distance approach
Level 0: (fastest access) \(a, b\) : same type as specified by End User
Level 1: (next fastest) \(temp_0, temp_1\) : same type as \('a' and 'b'\)

--Absolute memory distance approach
{All of the INPUT and OUTPUT information comes from the End User.
INPUT
\(a, b\) are (floating-point, integer)
r:ange: \([m1E-e1, m2E+e2]\)
relative error: \([x-x*p_1, x+x*p_2]\)
maximum distance: 3 uSec (4 uSec total for both \(a\) and \(b\) brought in together)

OUTPUT
<generic> are (floating-point, integer)
r:ange: \([m1E-e1, m2E+e2]\)
relative error: \([x-x*p_3, x+x*p_4]\)
maximum distance: 3 uSec
Temporary
\(temp_0, temp_1\) are <same-as-INPUT>
maximum distance: 2 uSec

ALGORITHM
1. READ input values \(a, b,\) and \(c\) from source and store them in registers: \(temp_0, temp_1, temp_2\)
2. see if \(temp_0 = 0\)
3. see if \(temp_1 = 0\)
4. see if \(temp_2 = 0\)

$\$EXECUTE ANY OF THE FOLLOWING SEGMENTS HAVING A TRUE ASSERTION
5. (2\%) \(temp_0 = temp_1 = temp_2 = 0 \Rightarrow\)
6. WRITE "Ill-posed equation - Infinite solutions" to destination
7. (2\%) \(temp_0 = 0, temp_1 = 0, temp_2 < 0 \Rightarrow\)
8. WRITE "Ill-posed equation - Zero roots" to destination
9. (8\%) \(temp_0 = 0, temp_1 < 0, temp_2 = 0 \Rightarrow\) \{ \(a=c=0\) --- 1 real root \}
10. WRITE "1 real root = 0.0 to destination
    \{ Solution: \(x = 0\) \}
11. (8\%) \(temp_0 = 0, temp_1 < 0, temp_2 < 0 \Rightarrow\) \{ \(a=0\) --- 1 real root \}
12. \(temp_3 = -temp_2\) \{ \(-c\) \}
13. \(temp_4 = temp_3 / temp_1\) \{ \(-c/b\) \}
14. WRITE "1 real root" \(temp_4\) to destination
15. (8\%) \(temp_0 < 0, temp_1 = 0, temp_2 = 0 \Rightarrow\) \{ \(b=c=0\) --- 1 real root \}
16. WRITE "1 real root = 0.0 to destination
    \{ Solution: \(x = 0\) \}
17. (25\%) \(temp_0 < 0, temp_1=0, temp_2<0 \Rightarrow\) \{ \(b=0\) --- 2 real roots when \(ac<0\) \}

$\$EXECUTE ANY OF THE FOLLOWING SEGMENTS HAVING A TRUE ASSERTION
18. (50\%) unlikesign(temp0,temp2) \Rightarrow\) \{ --- 2 imag. roots when \(ac>0\) \}
19. \(temp_5 = -temp_2\) \{ \(-c\) \}
    \{ Solution: \(x = (+/-)\sqrt{-c/a}\) \}
20. \(temp_6 = temp_5 / temp_0\) \{ \(-c/a\) \}
21. \(temp_7 = \sqrt{temp_6}\) \{ \sqrt{-c/a}\} \{ First real root \}
22. \(temp_8 = -temp_7\) \{ \(-\sqrt{c/a}\) \} \{ Second real root \}
23. WRITE "2 real roots = \" \(temp_7\) \(temp_8\) to destination
24. (50\%) likesign(temp0,temp2) \Rightarrow\)
25. \(temp_9 = temp_2 / temp_0\) \{ \(c/a\) \}
26. \(temp_{10} = \sqrt{temp_9}\) \{ \sqrt{c/a}\} \{ First imaginary root \}
27. \(temp_{11} = -temp_{10}\) \{ \(-\sqrt{c/a}\) \} \{ Second imaginary root \}
28. WRITE "2 imaginary roots = \" \(temp_6\) \(temp_7\) to destination
29. $\$END OF 'EXECUTE ANY' CLAUSE

Continued next page...
...continued from previous page.

30. (10%) temp0 <> 0, temp1 <> 0, temp2 = 0  =>  
   { c = 0 --- 2 real roots }
31. temp12 = -temp1  { -b }
32. temp13 = temp12 / temp0  { -b / a }
33. WRITE "2 real roots = " temp13 to destination
34. (37%) temp0<>0,temp1<>0,temp2<>0  =>  
   {none=0 --- 1 real root when b^2-4ac=0 }
35. temp14 = temp1 * temp1  { b^2 }  
   { 2 real roots when b^2-4ac>0 }
36. temp15 = 4.0 * temp0  { 4a }  
   { 2 complex rt when b^2-4ac<0 }
37. temp16 = temp15 * temp2  { 4ac }
38. temp17 = temp14 - temp16  { b^2 - 4ac }

$$ELSE_QUICKER$

   { calculate reciprocal of the denominator once }
39. temp18 = 1.0 / (2.0 * temp0)  { 1 / 2a }
40. (10%) temp17 = 0.0  =>  
   { discriminant of zero }
41. temp19 = -temp1  { -b }
42. temp20 = temp19 * temp18  { -b / 2a }
43. WRITE "1 real root = " temp20 to destination
44. (0%) temp17 > 0.0  =>  
   { positive discriminant }
45. temp21 = -temp1  { -b }
46. temp22 = sqrt(temp17)  { sqrt(b^2 - 4ac) }

$$EXECUTE ANY OF THE FOLLOWING SEGMENTS HAVING A TRUE ASSERTION

47. (60%) temp1 < 1.0  { if b < 0 }  =>  
48. temp23 = temp21 + temp22  { -b + sqrt(b^2-4ac) }
49. temp24 = temp23 * temp18  { (-b - sqrt(b^2-4ac))/2a }  
   { 1st real sol. }
50. temp25 = temp0 * temp24  { a * <1st solution> }
51. temp26 = temp3 / temp25  { c / a * <1st solution> }  
   { Second real solution }
52. WRITE "2 real roots = " temp24 temp26 to destination
53. (60%) temp1 > 1.0  { if b > 0 }  =>  
54. temp27 = temp21 + temp22  { -b + sqrt(b^2-4ac) }
55. temp28 = temp27 * temp18  { (-b + sqrt(b^2-4ac))/2a }  
   { 1st real sol. }
56. temp29 = temp0 * temp28  { a * <1st solution> }
57. temp30 = temp3 / temp29  { c / a * <1st solution> }  
   { Second real solution }
58. WRITE "2 real roots = " temp28 temp30 to destination
59. $$END OF 'EXECUTE ANY' CLAUSE
60. (45%) temp17 < 0.0  =>  
   { negative discriminant }
61. temp31 = -temp1  { -b }
62. temp32 = temp31 * temp18  { -b / 2a }  
   { Both real solutions }
63. temp33 = -temp17  { -(b^2-4ac) }
64. temp34 = sqrt(temp33)  { sqrt(-(b^2-4ac) ) }
65. temp35 = temp34 * temp18  { sqrt(-(b^2-4ac) )/2a }  
   { 1st imag solution }
66. temp36 = -temp35  { -sqrt(-(b^2-4ac) )/2a }  
   { 2nd imag solution }
67. WRITE "2 complex roots = (" temp32 temp35 ")
   "(" temp32 temp36 ")" to destination
68. $$SEND OF 'EXECUTE ANY' CLAUSE
$$ELSE_QUICKER

   { calculate the denominator once }
69. temp37 = 2.0 * temp0  { 2a }

Continued next page...

Figure 4: 2nd Stage Contract
$$\text{EXECUTE ANY OF THE FOLLOWING SEGMENTS HAVING A TRUE ASSERTION}$$

70. \[ (10\%) \text{ temp17} = 0.0 \implies \{ \text{discriminant of zero} \} \{ 1 \text{ real root} \}

71. \text{temp38} = -\text{temp1} \{ -b \}

72. \text{temp39} = \text{temp38} / \text{temp37} \{ -b / 2a \}

73. \text{WRITE} "1 \text{ real root = } \text{temp39} \text{ to destination}"

74. \[ (45\%) \text{ temp17} > 0.0 \implies \{ \text{positive discriminant} \} \{ 2 \text{ real roots} \}

\{ \text{Avoid cancellation of significant digits} \}

75. \text{temp40} = -\text{temp1} \{ -b \}

76. \text{temp41} = \sqrt{\text{temp17}} \{ \sqrt{b^2 - 4ac} \}

\$\text{EXECUTE ANY OF THE FOLLOWING SEGMENTS HAVING A TRUE ASSERTION}$

77. \[ (50\%) \text{ temp1} < 0.0 \implies \{ \text{if } b < 0 \}

78. \text{temp42} = \text{temp40} + \text{temp41} \{ -b + \sqrt{b^2 - 4ac} \}

79. \text{temp43} = \text{temp42} / \text{temp37} \{ (-b - \sqrt{b^2 - 4ac})/2a \} \{ 1\text{st real sol.} \}

80. \text{temp44} = \text{temp0} * \text{temp43} \{ a * <1\text{st solution}> \}

81. \text{temp45} = \text{temp3} / \text{temp44} \{ c/a * <1\text{st solution}> \} \{ \text{Second real solution} \}

82. \text{WRITE} "2 \text{ real roots = } \text{temp43} \text{ temp45 to destination}"

83. \[ (50\%) \text{ temp1} > 0.0 \implies \{ \text{if } b > 0 \}

84. \text{temp46} = \text{temp44} - \text{temp45} \{ -b - \sqrt{b^2 - 4ac} \}

85. \text{temp47} = \text{temp46} / \text{temp41} \{ (-b + \sqrt{b^2 - 4ac})/2a \} \{ 1\text{st real sol.} \}

86. \text{temp48} = \text{temp0} * \text{temp47} \{ a * <1\text{st solution}> \}

87. \text{temp49} = \text{temp3} / \text{temp48} \{ c/a * <1\text{st solution}> \} \{ \text{Second real solution} \}

88. \text{WRITE} "2 \text{ real roots = } \text{temp47} \text{ temp49 to destination}"

89. \$\text{END OF 'EXECUTE ANY' CLAUSE}$

90. \[ (45\%) \text{ temp17} < 0.0 \implies \{ \text{negative discriminant} \}

91. \text{temp50} = -\text{temp1} \{ -b \}

92. \text{temp51} = \text{temp50} / \text{temp37} \{ -b / 2a \} \{ \text{Both real solutions} \}

93. \text{temp52} = -\text{temp17} \{ -(b^2 - 4ac) \}

94. \text{temp53} = \sqrt{\text{temp52}} \{ \sqrt{-(b^2 - 4ac)} \}

95. \text{temp54} = \text{temp53} / \text{temp37} \{ \sqrt{-(b^2 - 4ac)}/2a \} \{ 1\text{st imag solution} \}

96. \text{temp55} = -\text{temp54} \{ -\sqrt{-(b^2 - 4ac)}/2a \} \{ 2\text{nd imag solution} \}

97. \text{WRITE} "2 \text{ complex roots = } (" \text{temp51 temp54 "}"

"(" \text{temp51 temp55 ")} \text{ to destination}"

98. \$\text{END OF 'EXECUTE ANY' CLAUSE}$

99. \$\text{END QUICKER}$

$\text{DEPENDENCIES}$

1-> [2-4] -> [5, 7, 9, 11, 15, 17, 30, 34] 5-> 6 7-> 8 9-> 10 11-> 14 15-> 16 17-> [23, 28]

30-> 33 34-> 43 1-> 12-> 13-> 14 1-> 31-> 32-> 33 1-> [18, 19, 24, 25, 35, 36, 39, 41, 45, 47, 49]

18-> 23 19-> 20-> 21-> 22-> 23 24-> 28 25-> 26-> 27-> 28 etc etc

$\text{Figure 4: 2\text{nd Stage Contract}}$
Next comes the application program itself. Our goal here is to provide just the basic algorithm without supplying any unnecessary dependencies or extra “tricks” to circumvent the compiler. As a rule, we never reuse any variable on the left hand side of an assignment statement. Also note that each line is numbered for later use in the dependency specification. To aid the compiler in making timing and error estimates, we also require that probability estimates be included for conditional statements. If a probability is not known in advance, this should also be stated. Also, for loops, we recommend that bounds be included in the form of a triple containing: the guaranteed least number of times through the loop, the average expected number of times through the loop, and the guaranteed maximum number of times through the loop. With this information, both average and worst case timing estimates can be calculated. Finally, the application programmer is permitted and *strongly encouraged* to use library subroutines when possible. The idea is to keep a collection of “best known solutions to date” on line to help ensure maximum performance or accuracy while freeing the application programmer from having to continuously “reinvent the wheel”. In some sense, this would be like keeping existing BLAS routines on line but it is also very different in many ways. For example, instead of having one routine to do a particular task, there would be several versions emphasizing difference precision, performance, and architectural requirements. Also, these routines would come with attached error and performance ratings so that the system software could easily match the appropriate version to meet the end user’s needs. Again, these details and the fact that a collection of similar routines exist need not be known by the application programmer. Finally, the subroutines would have to pass strict requirements before being added to the system and would absolutely be forbidden from having extra “tricks” embedded in them. After all, it was these extra tricks that lead to the problem example described in Section 2.2.2.

Then, to make life easier for the compiler, we require that a data dependency specification be included that explicitly states which line of code is dependent on another line of code. However, some dependencies can be omitted due to their transitive nature. For example, if line 1 is dependent on lines 2 and 3 and line 2 is also dependent on line 3, we could denote this as \( (1 \rightarrow 2 \quad 1 \rightarrow 3 \quad 2 \rightarrow 3) \) or alternatively as \( (1 \rightarrow [2, 3] \quad 2 \rightarrow 3) \). However, since the dependency \( (1 \rightarrow 3) \) is implied by the other two dependencies, it is safe to omit it and simply say that \( (1 \rightarrow 2 \quad 2 \rightarrow 3) \). Furthermore, we permit the shorthand notation of combining dependencies into the form: \( (1 \rightarrow 2 \rightarrow 3) \). Finally, we also permit multiple source to destination or source to multiple destinations to be expressed with ranges of lines use combinations of commas, hyphens, and square brackets. Hence, the expression \( (1 \rightarrow [2, 3] \rightarrow [4, 6, 8, 13]) \) is actually shorthand for \( (1 \rightarrow 2 \quad 1 \rightarrow 3 \quad 1 \rightarrow 4 \quad 1 \rightarrow 6 \quad 1 \rightarrow 8 \quad 1 \rightarrow 13 \quad 2 \rightarrow 4 \quad 2 \rightarrow 6 \quad 2 \rightarrow 8 \quad 2 \rightarrow 13 \quad 3 \rightarrow 4 \quad 3 \rightarrow 6 \quad 3 \rightarrow 8 \quad 3 \rightarrow 13) \).

Figure 5 illustrates the process that an end user experiences in order to submit a job. Figure 6 illustrates the process that an end user experiences in order to submit a job.
Figure 5: Traditional Compilation of an Application Program
Submit code and contract to compiler for Level 2 analysis

Logic errors?

Yes

Edit program and resubmit for first level compilation

No

Submit code and contract to compiler for Level 2 analysis

Compiler

Library subroutines

Hardware simulator and/or specification file

Are there any optimizations or changes that the compiler would like to recommend?

Yes

Notify the application programmer.

No

Can contract requirements be met?

Yes

Success! Store object code. "Tag" the file(s) with cost, timing, accuracy, and reliability ratings. Report back results.

No

Does programmer want to alter the terms?

Yes

Edit contract and resubmit for second level compilation

No

Failure!

Second level compilation.

Contract must be satisfied.

Priority may be placed on speed, accuracy or cost.

Figure 6: (Second level) Compilation of an Application Program with Contract
4.3 Compiler to Hardware Contract

The third-stage contract is between the compiler writer and the hardware. It is at this stage that specific details (like timing, reliability, and topology) are specified. This information can be specified in a hardware specification file or determined from a microcode simulator. A preliminary example is given in Figure 7. Its contents must include sufficient detail and information to provide the compiler with the resources it needs to estimate wall clock time and output error for the application at hand. In addition, the compiler must know about parallel features in the likelihood that an optimization is needed or requested. Hence, the essential features to be included in the file include, but are not limited to:

- clock cycle time
- time to execute the various arithmetic and logic functions for both INTEGER and FLOATING POINT representations under the different precisions that are available
- precisions and ranges for the INTEGER and FLOATING POINT representations including information on how the values are stored (e.g. biased format for exponents)
- the number of and types of functional units including chaining information and restrictions
- a full description of the memory regime including the cache policies in use
- parallel topology (if any)
- communication network information

The idea is that the front end of a compiler should still be robust enough and adaptable enough to obtain good performance for a variety of platforms.

5 Conclusions and Future Work

In this technical report, we have presented a new programming paradigm involving a three-stage contract between four participants. The participants are: the end user, the application program, the system software, and the hardware specification file. An end user is a person wishing to execute some application but is not assumed to know or care about the architecture, topology, or any machine-specific details. The application programmer engages in a dialog with the compiler to produce an “application program” – a suite of many different, best-known algorithms that cover a wide range situations. Each application program is tagged by the compiler with timing, accuracy, cost, and reliability estimates. This information is used in turn to match the

\[\text{This work is still VERY sketchy and is subject to substantial change in future papers. Suggestions and comments are solicited from the reader on this and any other section of this technical report.}\]
Clock Cycle: 50 ns

Operations:

Logical:
- 1-bit
- 8-bit
- 16-bit
- 32-bit
- 64-bit

Arithmetic:
- Integer (16-bit):
  - add
  - subtract
  - multiply
  - divide
  - square root
  - x^y
  - negation

- Integer (32-bit):
  - add
  - subtract
  - multiply
  - divide
  - square root
  - x^y
  - negation

- Floating Point (32-bit):
  - add
  - subtract
  - multiply
  - divide
  - square root
  - x^y
  - negation

- Floating Point (64-bit):
  <similar to above>

Precision:

Integer (8-bit):
- Unsigned representation
  Range: [0..255]
- Two's complement representation
  Range: [-32768..-32767]

Integer (16-bit):
- Two's complement representation
  Range: [-2147483648..2147483647]

Integer (32-bit):
- Two's complement representation
  Range: [-2147483648..2147483647]

Floating Point (32-bit):
- Mantissa: 15-bit normalized in Two's complement
- Exponent: 7-bit in Biased format with zero= 0111111
- Overflow representation:
- Underflow representation:
- Absolute error: 0.00000005

Continued next page...

Figure 7: Hardware Specification File
Functional Units:
- 2 16-bit integer ALU (IU1 IU2) may be used simultaneously for a single 32-bit operation
- 3 32-bit floating-point ALU (FPU1 FPU2 FPU3)

Cascading: IU1 -> IU2 FPU1 -> FPU2 -> FPU3 FPU1, FPU2 -> FPU3

Memory Regime:
Registers:
- 32 32-bit registers (or 16 64-bit registers)
  1cc refresh rate

Cache:
- 1024 32-bit 8-way etc etc associate cache
  minimum block size: 128 words
  warm start on swapping
  write back on write policy
  cache hit: 1cc
  cache miss: 12cc

Main Memory:
- 32 MByte in 8 modules with 4 byte stride
  1.75cc refresh rate per module

Figure 7: (Continued) Hardware Specification File
end user with the most appropriate algorithm. It may be the case that no such algorithm exists
that can meet the end user's requirements. In such a case, contract negotiations may ensue to
relax end user requirements. Finally, the hardware specification file contains all of the information
specific to a particular platform. This information is used to help determine timing, cost, and
reliability estimates. The file's topological details also help in the selection of the appropriate
algorithm for a particular machine.

Obviously, we cannot give all the details of our scheme in this summary. Rather, we would
like to reiterate that the main contribution here is the new way in which interactions with the
computer are thought of. Now, the user cares only about obtaining acceptably good answers
in a reasonable amount of time and within a particular (cash or time) budget. The application
program can now focus on the underlying algorithm without presuming compiler and hardware
behavior. Finally, the compiler is given the information it really needs: what can be run in parallel
and what types of hardware are necessary to do the job.

In the short term, we hope to develop a working prototype. The hardest part will be finding
a universal, hardware specification. The compiler is also expected to take a great deal of time.
Then, once the prototype is working, a library of routines for a variety of precisions and timing
can be developed. Gradually, more target hardwares will be incorporated with an emphasis on
parallel platforms. The creation of a robust system capable of handling a wide variety of machines
is likely to take ten years or longer.

If we are successful, application programs will be thought of in an entirely new way. For one
thing, they will not be dependent on any given language. More importantly, though, the application
program (by not specifying any trick code for higher performance) will become immortal as
it locks in the essence of the algorithm while ignoring techniques and technology. Such a program
could, in theory, need never be rewritten except in the case where the underlying algorithm itself
is improved. Then, as these immortal programs are developed, an international clearing house or
repository can be set up providing on line access to the code.

Authors' Note:

This Technical Progress Report summarizes some of the work done during the Summer of 1994 and
in May 1995. In many cases, our thoughts and views have undergone substantial changes making
portions of this report obsolete. In particular, we no longer hold the view that the compiler should
be expected to perform any sort of clever optimization or be expected to find alternate solutions.
In fact, the third participant is now referred to as the system software and not just the compiler.
Many aspects of the contracts have also changed. For the latest version of our work, we refer the
reader to a pending publication [19].

References


Distribution List

INTERNAL DISTRIBUTION
1. J. Corones, 329 Wilhelm Hall, Ames Laboratory
2. S. Elbert, 327 Wilhelm Hall, Ames Laboratory
3. D. Heller, 327 Wilhelm Hall, Ames Laboratory
4. Technical Information Office, Ames Laboratory
5. Chicago Field Office, Patent Counsel, 9800 Cass Avenue
6. Director, ISURF, 315 Beardshear, Iowa State University, Ames, IA 50011-3020
9-20. Scalable Computing Laboratory, 236 Wilhelm Hall, Ames Laboratory

EXTERNAL DISTRIBUTION
22. Daniel F. Anderson, Ford Scientific Research Laboratory, Room E-1135, P.O. Box 2053, Dearborn, MI 48121-2053
23. Clifford Arnold, 2097 Iglehart, St. Paul, MN 55104
24. Mark Arnold, 2456 Brophy Circle #2, Fayetteville, AR 72703
25. Cleve Ashcraft, Boeing Computer Services, P.O. Box 24346, M.S. 7L-21, Seattle, WA 98124-0346
27. Albert Baker, Dept of Comp. Science, 226 Atanasoff, Iowa State University, Ames, IA 50011
28. David Bailey, NASA/Ames Research Center, M.S. T27A-1, Moffett Field, CA 94035-1000
29. Ray Bair, Pacific Northwest Laboratory, P.O. Box 999, M.S. K1-87, Battelle Boulevard, Richland, WA 99352
30. Ben Barker, BBN Advanced Computers, Inc., 10 Moulton Street, Cambridge, MA 02138
31. Hal Barraclough, 3160 Fairview Park Drive, Falls Church, VA 22042
32. Edwin H. Barsis, Division 1400, P.O. Box 5800, M.S. 0321, Sandia National Laboratories, Albuquerque, NM 87185
33. Robert S. Barton, 7350 SW Sharon Lane, Portland, OR 97225
34. C. Gordon Bell, 450 Old Oak Court, Los Altos, CA 94022
35. Donna Bergmark, 745 E&TC Building, Hoy Road, Cornell University, Ithaca, NY 14853
37. Steve Bershaded, MRJ Inc., 10455 White Granite Drive, Suite 104, Oakton, VA 22124
38. Jeff Brooks, Cray Research, 655 F Lone Oak Drive, Eagan, MN. 55121
39. James C. Browne, Department of Computer Sciences, University of Texas, Austin, TX 78712
40. Bill L. Buzbee, Scientific Computing Division, NCAR, P.O. Box 3000, Boulder, CO 80307
41. Duane B. Call, Computer System Architects, 950 North University Avenue, Provo, UT 84604
42. William Camp, Cray Research Inc., Cray Research Park, 55A Lone Oak Drive, Eagan, MN 55121-1580
43. Ralph Carlson, Lawrence Livermore National Laboratory, P.O. Box 808: L-31, Livermore, CA 94550
44. Tom Casavant, University of Iowa, Department of Electrical and Computer Engineering, Iowa City, IA 52242
45. John Cavallini, Scientific Computing Office, ER-30, U.S. DOE, 19901 Germantown Road, Germantown, MD 20874
46. Melvyn Ciment, National Science Foundation, 1800 G Street NW, Washington, DC 20550
47. Denny Cole, Portland Group, 9150 SW Pioneer Court, Suite H, Wilsonville, OR 97070
48. Alva Couch, Department of Computer Science, Tufts University, Medford, MA 02155
49. Frederica Darema, National Science Foundation, 1800 G Street NW, Washington, DC 20550
50. Howard Demuth, University of Idaho, Electrical Engineering Department, Moscow, ID 83844-1023
51. Tice DeYoung, ARPA, 3701 N. Fairfax Drive, Arlington, VA 22203-1714
52. Dennis Duke, SCRI, Florida State University, Tallahassee, FL 32306-4052
53. Hesham El-Rewini, University of Nebraska at Omaha, Department of Computer Science, Omaha, NE 68182
55. Roger Firestone, MRJ Inc., 10455 White Granite Drive, Suite 200, Oakton, VA 22124
56. Joseph E. Flaherty, Department of Computer Science, Rensselaer Polytech Institute, Troy, NY 12180-3590
57. Bob Flynn, Polytechnic Institute, 456 North Street, White Plains, NY 10605
58. Ian Foster, Argonne National Laboratory, Math & Comp. Science Division, 9700 Cass Avenue, Argonne, IL 60439
59. Joan M. Francioni, Computer Science Department, University of Southwestern Louisiana, Lafayette, LA 70504
60. Paul O. Fredrickson, Cray Research Inc., Suite 101, 557 Oppenheimer, Los Alamos, NM 87544
61. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401
62. Jean-Luc Gaudiot, Department of Electrical Engineering-Systems, USC, EEB-336, Los Angeles, CA 90089-2562
64. William Greenley, 516 Webster Street, Webster City, IA 50595
65. Bruce Greer, Intel SSD, 5200 NE Elam Young Parkway, M.S. C06-05, Hillsboro, OR 97124
66. Stuart Hawkinson, Intel SSD, CO6-08 Bldg. Zone 8, 14924 NW Greenbrier Parkway, Beaverton, OR 97006
67. Ann H. Hayes, Computing and Communications Division, Los Alamos National Laboratory, Los Alamos, NM 87545
68. Michael T. Heath, NCSA, 4157 Beckman Institute, University of Illinois, 405 North Mathews Ave., Urbana, IL 61801-2300
69. Richard Hirsh, National Science Foundation, 4201 Wilson Boulevard, Room 1122, Arlington, VA 22230
70. Daniel A. Hitchcock, Scientific Computing Office, ER-30, U.S. DOE, 19901 Germantown Road, Germantown, MD 20874
72. Frederick Howes, Scientific Computing Office, ER-30, U.S. DOE, 19901 Germantown Road, Germantown, MD 20874
73. Gary Johnson, Institute for Computational Sciences and Informatics, George Mason University, M.S. 5C3, 103 Science and Technology I, Fairfax, VA 22030-4444
74. Jan Julianus, Information and Business Division, Elsevier Science Publishers B.V., Sara Burgerhartstraat 25, P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands
75. Alan H. Karp, HP Labs 3U-7, Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304
77. David Krumme, Tufts University, Dept. of Computer Science, Pearson Annex, Medford, MA 02155
78. John Levesque, Applied Parallel Research, Inc., 550 Main Street, Suite I, Placerville, CA 95667
79. Ted Lewis, Computer Science Department, Oregon State University, Corvallis, OR, 97331
80. James McGraw, Lawrence Livermore National Laboratory, P.O. Box 808: L-306, Livermore, CA 94550
81. Gary Montry, Southwest Software, 11812 Persimmon NE, Albuquerque, NM 87111
82. H. Stephen Morse, 3312 Pathway Ct., Falls Church, VA 22042-3325
83. Thomas Nash, Fermilab, P.O. Box 500, M.S. 127, Batavia, IL 60510
84. Naresh Nayar, IBM, 3605 Highway 52 North, Department 44J/015-2, Rochester, MN 55901
86. Ed Oliver, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 4500N, Room K-244, Oak Ridge, TN 37831-6259
87. Cherri Pancake, Oregon State University, Department of Computer Science, Corvallis, OR 97331
88. William J. Pardee, PQM, 798 Lynnmere Drive, Thousand Oaks, CA 91360
89. Roy P. Pargas, Department of Computer Science, Clemson University, Clemson, SC 29634-1906
90. Michael J. Quinn, Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202

28
91. John Riganati, David Sarnoff Research Center, CN 5300, Princeton, NJ 08543-5300
92. Diane T. Rover, 155 Engineering, Dept. of Electrical Engineering, Michigan State University, East Lansing, MI 48824
93. Robert Schreiber, RIACS, M.S. 230-5, NASA Ames Research Center, Moffett Field, CA 94035
94. Margaret L. Simmons, Computing and Communications Division, LANL, Los Alamos, NM 97545
95. Anthony Skjellum, Department of Computer Science, Drawer CS, 300 Butler Hall, Mississippi State, MS 39762-5623
96. Burton Smith, Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103
97. Norris Parker Smith, HPC Wire, 4678 Whiting Street, Pleasanton, CA 94566
98. Guy Steele, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214
99. Rick Stevens, Math and Comp. Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL
100. Steve Stevenson, Department of Computer Science, Clemson University, Clemson, SC 29634-1906
101. Xian-He Sun, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803-4020
102. John Toole, ARPA/CSTO, 3701 N. Fairfax Drive, Arlington, VA 22203-1714
103. Robert G. Voigt, National Science Foundation, Room 417, 1800 G Street NW, Washington, DC 20550
104. Dan Warner, Department of Mathematics, Clemson University, Clemson, SC 29634-1906
105. Gil Weigand, Sandia National Laboratories, P.O. Box 5800, Org 1090, M.S. 0170, Albuquerque, NM 87185
106. James M. Winget, Silicon Graphics, 2011 N. Shoreline Boulevard, Mountain View, CA 94043-1389
107. Michael Wolfe, OGI, Dept. of Comp. Sci. and Engineering, 19600 NW von Neumann Drive, Beaverton, OR 97006-1999
108. Jack Worlton, 2231 East 3980 South, Salt Lake City, UT 84124