DOE/ER/25176--T1

Performance Measurement and Analysis Techniques for Parallel and Distributed Programs

Technical Progress Report

Department of Energy Grant DE-FG02-93ER25176

August 1, 1996 - May 31, 1997

Principle Investigators:

Barton P. Miller
bart@cs.wisc.edu

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Computer Sciences Department
University of Maryland
College Park, MD 20742

MASTER

This report summarizes our technical progress during the first year of our three year proposal. This research is centered about the Paradyn Parallel Performance Tools. We include a summary of research accomplishments, technical transfers, and a list of papers written under this grant. We have made good progress on our research goals and have a strong outlook for the coming year.

1 SUMMARY OF TECHNICAL PROGRESS

During this past year, we have made progress in several areas of our performance research. The research areas in which we have made progress include profiling memory behavior, feedback mechanisms for controlling instrumentation overhead, experiment management, operating system kernel instrumentation, Paradyn support for Windows/NT, new predictive performance metrics, and integrating visualization tools with Paradyn.

1.1 Profiling Memory Performance

We have developed a new technique for collecting and displaying shared-memory performance information[8]. We detect cache block sharing patterns that indicate potential performance bottlenecks by using Paradyn with a modified cache coherence protocol for the Blizzard fine-grain distributed shared memory system running on the Wisconsin Cluster Of Workstation (COW)¹. Paradyn has been extended to present shared-memory performance data in a data-centric manner, which relates events on cache blocks to program data structures.

The first step in memory profiling is to detect access patterns that indicate performance problems. Our pattern detection mechanism is integrated in a shared memory cache-coherence protocol, so it incurs little overhead and

1. The Wisconsin Cluster of Workstations (COW) is a collection of 40 Sun SPARCStation 20's (with dual 66Mhz Ross HyperSPARC processors) connected by a Myricom Myrinet and 100MB Ethernet.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

May 28, 1997

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.

DISCLAIMER

**Portions of this document may be illegible
in electronic image products. Images are
produced from the best available original
document.**

requires no changes to programs. More importantly, this mechanism associates addresses with events, so a performance tool can present measurements at the program's level of abstraction. Paradyn can use these addresses to relate memory accesses to a program's data structures and use conventional profiling techniques to connect accesses with the statements that execute them. We call this process *shared-memory performance profiling*. Memory profiling by itself cannot find all performance problems, so we built memory profiling into the more extensive facilities of Paradyn.

Although this work exploits Blizzard's custom protocols, other systems can also provide mechanisms to associate shared memory communication with data. Any hardware shared-memory platform can support a fine-grain DSM system, like Blizzard or DEC's Shasta, that exposes a coherence protocol for performance debugging. Since the underlying shared-memory hardware provides fast communication, a DSM system of this sort would incur only moderate overhead. Alternatively, a platform may provide hardware features, such as informing memory operations that trap on cache misses, which can be used to associate a program's memory references with a coherence protocol's actions.

We have illustrated the use of memory profiling through an extended case study of tuning a new shared-memory protein folding code from researchers in the University of Wisconsin Chemical Engineering Department. With the help of memory profiling and Paradyn, we improved this application's performance by more than a factor of 4 to an efficiency of 80% on 16 nodes.

1.2 Controlling the Overhead of Dynamic Instrumentation

Software based monitoring can introduce overhead into the application and can alter its performance. The unique feature of our approach is that we let the programmer see and control the overhead introduced by monitoring rather than simply being subjected to it. To manage the perturbation caused by instrumentation, we have developed an instrumentation cost system to ensure that data collection and analysis does not excessively alter the performance of the application being studied[4,7]. The model associates a cost with each different resource used. Resources include processors, interconnection net works, disks, and data analysis workstations. The cost system is divided into two parts: *predicted cost* and *observed cost*. Predicted cost is computed when an instrumentation request is received, and observed cost while the instrumentation is enabled.

By computing the predicted cost of instrumentation before data collection starts, it is possible to decide if the requested data is worth the cost of collection. For example, if the user requests performance data whose predicted cost of collection is 100% of the application's run-time, they might decide that the impact of collecting the data is too high to warrant collection. This predictive information can be used as feedback to reduce or defer an instrumentation request. Our higher-level performance analysis tools use the cost prediction to control how aggressively they instrument a program in search of performance bottlenecks. In many cases, control of instrumentation overhead can allow our tools to more quickly isolate a performance problem.

Although predicting the cost of data collection prior to instrumentation execution provides useful data, it is important to make sure that the actual cost of data collection matches the predicted cost. The observed cost tracks the

impact the currently enabled instrumentation has on the application. By computing this value, we verify that the actual impact of instrumentation is held within predefined limits. If the observed cost exceeds these limits, feedback is provided to the user or higher-level tool; this feedback allows us to dynamically maintain (approximately) a fixed level of instrumentation overhead.

1.3 Experiment Management

The development of a high-performance parallel system or application is an evolutionary process. It may begin with models or simulations, followed by an initial implementation of the program. The code is then incrementally modified to tune its performance and continues to evolve throughout the applications's life span. At each step, the key question for developers is: *how and how much did the performance change?* This question arises comparing an implementation to models or simulations; considering versions of an implementation that use a different algorithm, communication or numeric library, or language; studying code behavior by varying number or type of processors, type of network, type of processes, input data set or work load, or scheduling algorithm; and in benchmarking or regression testing. Despite the broad utility of this type of comparison, no existing performance tool provides the necessary functionality to answer it; even state of the art research tools such as Paradyn and Pablo focus instead on measuring the performance of a single program execution.

We have been work on an infrastructure for answering this question at all stages of the life of an application[10]. We view each program run, simulation result, or program model as an *experiment*, and provide this functionality in an *Experiment Management* system. This research has three parts: (1) a representation for the space of executions, (2) techniques for quantitatively and automatically comparing two or more executions, and (3) enhanced performance diagnosis abilities based on historic performance data. In this paper we present initial results on the first two parts. The measure of success of this project is that an activity that was complex and cumbersome to do manually, we can automate.

The first part is a concise representation for the set of executions collected over the life of an application. We store information about each experiment in a *Program Event*, which enumerates the components of the code executed and the execution environment, and stores the performance data collected. The possible combinations of code and execution environment form the multi-dimensional *Program Space*, with one dimension for each axis of variation and one point for each Program Event. We enable exploration of this space with a simple naming mechanism, a selection and query facility, and a set of interactive visualizations. Queries on a Program Space may be made both on the contents of the performance data and on the metadata that describes the multi-dimensional program space. A graphical representation of the Program Space serves as the user interface to the Experiment Management system.

The second part of the project is to develop techniques for automating comparison between experiments. Performance tuning across multiple executions must answer the deceptively simple question: what changed in this run of the program? We have developed techniques for determining the "difference" between two or more program runs, automatically describing both the structural differences (differences in program execution structure and resources used), and the performance variation (how were the resources used and how did this change from one run to the next).

We can apply our technique to compare an actual execution with a predicted or desired performance measure for the application, and to compare distinct time intervals of a single program execution. Uses for this include performance tuning efforts, automated scalability studies, resource allocation for metacomputing, performance model validation studies, and dynamic execution models where processes are created, destroyed, migrated, communication patterns and use of distributed shared memory may be optimized, or data values or code may be changed by steering. The difference information is not necessarily a simple measure such as total execution time, but may be a more complex measure derived from details of the program structure, an analytical performance prediction, an actual previous execution of the code, a set of performance thresholds that the application is required to meet or exceed, or an incomplete set of data from selected intervals of an execution.

The third part of this research is to investigate the use of the predicted, summary, and historical data contained in the Program Events and Program Space for performance diagnosis. We are exploring novel opportunities for exploiting this collection of data to focus data gathering and analysis efforts to the critical sections of a large application, and for isolating spurious effects from interesting performance variations.

While our early results are quite encouraging, this effort is still in its early stages.

1.4 Dynamic Instrumentation in the Operating System Kernel

We have developing initial techniques for using dynamic instrumentation for profiling, debugging, and extending the operating system kernel. Our experimental platform is the standard Solaris 2.5.1 kernel running on an UltraS-PARC. We have been developing techniques for inserting initial control software into the kernel, establishing shared-memory communication regions, and modifying kernel control flow for instrumentation.

A key difference between instrumenting a kernel and instrumenting application code is that you cannot pause the operating system kernel when modifying its code. When we instrument application code, we often replace sequences of multiple instructions to intercept the control flow, and pause the application to insure that we atomically update the sequence. In the kernel, we want to replace single instructions, requiring more sophisticated analysis of the code that we are modifying. We have been developing control and data flow analyses of the binary code to support more sophisticated dynamic code control and generation schemes. This effort is in its early stages, but ultimately will result in lower overhead and greater flexibility on all of our platforms.

1.5 Paradyn on Windows/NT

We have ported the Paradyn daemon to Windows/NT 4.0. This port includes dynamic instrumentation for the Intel x86 architecture and application control for NT. We can handle dynamic linked libraries (DLL's) and can instrument unmodified executable (.EXE) files. The port uses the native Microsoft Visual C++ compiler (so that the Paradyn source code is compatible with both the GNU g++ and Visual compile systems). Paradyn can run and monitor application processes on NT, along with the UNIX-based platforms. Heterogeneous parallel applications can be profiled with some processes running on NT and some running on Solaris or AIX.

The Paradyn front-end process, with its graphic user interface has not yet been ported to NT. We are still analyz-

ing the best path to supporting the GUI on both NT and X-window system. We are also investigating the use of the Pentium Pro hardware performance counters with dynamic instrumentation.

1.6 Paradyn Support for Trace Data

The data collection model in Paradyn is based on accurately counting and timing the occurrence of events during a programs execution and then periodically sampling these values and forwarding them for display or analysis. This approach provides a simple, yet powerful model of data collection. However, for some purposes, it is necessary to collect a log of events as they happen (i.e. a trace). We have extended the Paradyn instrumentation system to support the collection of trace data and to forward these traces to analysis and visualization processes via an extension to Paradyn's external visualization interface (visi). To permit dynamic (run time) control of trace data, we have implemented tracing as an extension to Paradyn's Metric Definition Language (MDL).

1.7 Modeling of Paradyn's Scalability

We worked with Prof. Diane Rover and Abdul Waheed of Michigan State University to evaluate the scalability of Paradyn's data collection system[6,9]. These simulations predicted the performance of Paradyn running on large-scale message passing and shared memory computers (with up to 1,000 nodes). The results of these simulations showed that with a couple of changes to Paradyn (batch forwarding of samples and software combining tree for Paradyn demon data) that the system can support very large configurations. The change to batch forwarding has already been incorporated into Paradyn and the measured improvement is similar to the simulation predictions.

1.8 Predictive Performance Metrics

Traditionally performance measurement tools have been used to identify bottlenecks in existing programs. However, once a problem has been located, the user is left with the task of trying out alternatives to fix the bottleneck. This process is often haphazard and may involve wasting time on tuning options that provide little gain in the overall application execution time. Recently, we have been working to develop a family of "what-if" metrics to permit programmers to evaluate the performance implications of potential tuning alternatives before they are implemented. The goal of these metrics is to try to predict the performance of an application after it has been changed. Possible changes that we are able to support include: tuning individual procedures, changes in the latency or bandwidth of the communication network, and changes in process to processor mappings. The technique used is to run the current configuration of the program under Paradyn, and during program execution "simulate" the impact of the proposed change in configuration. The key idea is to use as much information about the execution of the program in its current configuration as possible. Results have shown that it is possible to predict communication and process allocation changes within 15 percent for collection of scientific programs (NAS benchmarks).

1.9 Integrating Visualization Tools with Paradyn

Performance tuning a parallel application involves integrating performance data from many components of the system, including the message passing library, performance monitoring tool, resource manager, operating system, and

the application itself. The current practice of visualizing these data streams using a separate, customized tool for each source is inconvenient from a usability perspective, and there is no easy way to visualize the data in an integrated fashion. We demonstrated a solution to this problem using Devise, a generic visualization tool that is designed to allow an arbitrary number of different but related data streams to be integrated and explored visually in a flexible manner[3]. We displayed data emanating from a variety of sources side by side in three case studies. First we interface the Paradyn Parallel Performance Tool and Devise, using two simple data export modules and Paradyn's simple visualization interface. We showed several Devise/Paradyn visualizations which are useful for performance tuning parallel codes, and which incorporate data from Unix utilities and application output. Next we visualized trace data from a parallel application running in a Condor cluster of workstations. Last, we demonstrated the utility of Devise visualizations in a study of Condor cluster activity

1.10 Port of Paradyn to DEC Alpha and Integration with Multikron Hardware Monitor

We have ported the Paradyn Parallel Performance Tools to the DEC Alpha processor (and Digital UNIX), and integrated support for the NIST developed Multikron Hardware Performance Monitor. Although these activities were supported by NIST, they leverage previous and current funding from DOE.

2 TECHNOLOGY TRANSFERS

This has been an active year for technology transfer activities in the Paradyn project. Release 2.0 of Paradyn occurs on June 1, 1997, and includes substantial functionality and performance improvements. We can attach to already running programs (such as database servers), instrument completely unmodified binary programs, and have support for MPI on the IBM SP platform, and an initial Windows/NT port of Paradyn.

A major technical milestone is the release of the Paradyn DynInst API. This interface will provide researchers in a wide variety of areas access to our core technologies. In addition, we had the first of our Paradyn Week meetings, bringing together Paradyn developers and technology partners, and we have the first commercial licensing of Paradyn (by IBM).

2.1 Dynamic Instrumentation API

We have defined and implemented an Application Programmer Interface (API) for Paradyn's run time instrumentation system. This API permits programmers to use the unique features of run time code generation and program control that have been developed as part of the Paradyn Project for other purposes besides performance measurement. Potential uses include architectural simulation, fast data breakpoints for debuggers, and application steering. We have produced an early access distribution of the API, and anticipate first public release in summer 1997.

The DynInst API was demonstrated at Supercomputing'96 as part of the University of Maryland's research exhibit booth. We plan on having a separate Paradyn research exhibit booth at the upcoming Supercomputing '97 in San Jose.

2.2 Paradyn Week

In February 1997, we organized the first "Paradyn Week" meeting in Madison. This meeting brought together Paradyn developers, industry and research laboratory collaborators, and other interested technical groups. The first day was dedicated to technical briefings from the Paradyn efforts at Wisconsin, Maryland, and Los Alamos. The remainder of the week included small and large group in-depth technical discussions. The DynInst API design was reviewed, analyzed, and revised over the course of the week. The group involved in this discussion included members from Wisconsin, Maryland, and IBM's Power Parallel Division. Also at this meeting, the Paradyn group worked with Sun's Database Engineering Group to develop an approach to using Paradyn to access the new hardware performance counters in the UltraSPARC processes.

Organizations represented at the Paradyn Week meeting include: University of Wisconsin, University of Maryland, IBM, Sun Microsystems, Microsoft Research, Intel, Los Alamos National Lab, Argonne National Lab, Sandia National Lab, DOE ER, NIST, and USC ISI.

Given the success of this meeting, we plan on making this an annual event.

2.3 Paradyn License to IBM

We have signed a license agreement with IBM giving them rights to use, modify, and distribute Paradyn for commercial uses. IBM's Power Parallel Division is planning a new generation of run time tools based on our dynamic instrumentation technology. This use of Paradyn can have a substantial impact on software development on the ASCI Blue platform.

3 LIST OF PUBLICATIONS AND REPORTS

1. D.C. Burger, R.S. Hyder, B.P. Miller, and D.A. Wood, "Paging Tradeoffs in Distributed Shared-Memory Multiprocessors", *Journal of Supercomputing* 10, 1 (1996). Also appears in *Supercomputing '94*, Washington, DC, November 1994.
2. K. Kunchithapadam and B.P. Miller, "Integrating a Debugger and a Performance Tool for Steering" in **Debugging and Performance Tools for Parallel Computing Systems**, IEEE Computer Society Press, M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, eds., pp. 53-64, 1996.
3. K.L. Karavanic, J. Myllymaki, M. Livny and B.P. Miller, "Integrated Visualization of Parallel Program Performance Data", *Int'l Journal of Parallel Computing* 11, 2 (Summer, 1997). Special issue environments and tools for parallel scientific computing.
4. J.K. Hollingsworth and B.P. Miller, "Using Cost to Control Instrumentation Overhead", *Theoretical Computer Science* (TCS), to appear.
5. J.K. Hollingsworth and E. L. Miller, "Using Content-Derived Names for Configuration Management", *ACM Symposium on Software Reusability*, Boston, MA, May 1997, pp. 104-109.
6. A. Waheed, D. Rover, and J.K. Hollingsworth, "Modeling, Evaluation, and Testing of Paradyn Instrumentation System", *Supercomputing '96*, Pittsburgh, PA, November 1996.
7. J.K. Hollingsworth and B.P. Miller, "An Adaptive Cost Model for Parallel Program Instrumentation", *Euro-Par'96*, Lyon, France, August 1996, pp. 88-98.
8. Z. Xu, J.R. Larus, and B.P. Miller, "Shared-Memory Performance Profiling", *6th ACM Symposium on Principles and Practice of Parallel Programming*, Las Vegas, June 1997.
9. A. Waheed D. T. Rover, and J. K. Hollingsworth, "Modeling and Evaluating Design Alternatives for an On-Line

- Instrumentation System: A Case Study", *submitted to IEEE Transactions on Software Engineering*.
10. K. Karavanic and B.P. Miller, "Experiment Management Support for Performance Tuning", *submitted to Supercomputing'97*.
 11. C. Akinlar, E. Guven, and J.K. Hollingsworth, "Benchmarking a Network of PCs Running Parallel Applications", *submitted to Supercomputing'97*.
 12. J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *submitted to ACM/USENIX Conference on Domain Specific Languages*.

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.