PARALLEL PROGRAMMING TOOLS

Innovation in the field of digital computers has been spurred by an ever-increasing demand for computing power. Driven by the desire for speed, processor designs are incorporating more and more complicated features. While these features result in increased performance, the price-to-performance ratio of such uniprocessor systems is fairly high. This diminishing return of performance has motivated the development of parallel computer systems, which consist of a number of uniprocessors connected together by an interconnection network (for more information, see INTERCONNECTION NETWORKS FOR PARALLEL COMPUTERS).

High-performance parallel computers can often be built for a fraction of the cost of a comparably powerful uniprocessor system, especially if commodity (off-the-shelf) components are used. However, the relatively lower cost of such systems comes at a premium. In order to make use of the available concurrency, the programs that run on these systems must have multiple threads of control. These *parallel* programs are highly complex and may require significant programmer effort before they can be made to run correctly and efficiently.

Debuggers and performance analysis tools assist programmers in writing correct, efficient programs. A debugger is a tool that helps programmers pinpoint mistakes in the program that lead to incorrect behavior. Debugging is more of an art than a science, since identifying the erroneous behavior is only the first step. The programmer must then go backward to find the flaw in the program that gives rise to the incorrect behavior at a later point. A performance analysis tool helps identify the reasons for a program to perform below expecta-

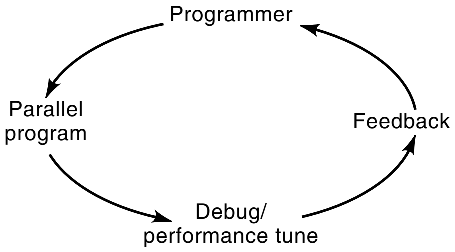

Figure 1. Debugging and performance tuning.

tions. Since there may be any number of reasons for poor performance, performance analysis is also an inexact science.

Debugging and performance tuning are generally cyclic processes, as illustrated in Fig. 1. Debugging typically involves repeating program executions, collecting more and more information from each run, until the program flaw has been identified. In performance tuning, the program is modified based on information gathered when executing the program. Typically, the execute–modify cycle has to be repeated several times before all the performance problems can be addressed.

All of the difficulties of debugging and performance tuning sequential programs are present in parallel programs as well. Debugging and tuning a parallel program is complicated by the interaction between the multiple concurrent components (threads or processes) of a parallel program execution. In particular, interprocess interactions often cause the program behavior to be nonrepeatable, which makes both correctness and performance problems hard to pinpoint. Thus, while it may be possible to build cheap, high-performance parallel computers, it takes sophisticated tools to make parallel programming truly effective on such systems.

A wide variety of programming languages and models can be used for writing parallel programs. Programming paradigms such as logic, data flow, and functional programming are considered more natural vehicles for expressing parallelism (for more information on these topics, see LOGIC PROGRAMMING AND LANGUAGES, DATA-FLOW AND MULTITHREADED ARCHITECTURES, and FUNCTIONAL PROGRAMMING). However, imperative programs (i.e., programs written in languages such as Fortran, Pascal, and C, which explicitly specify the sequence of steps that must be followed by the computer to solve the problem at hand) are currently the best when it comes to extracting performance out of parallel systems. Since performance is one of the motivating factors for using a parallel system, this article concentrates on the debugging and performance analysis of imperative parallel programs.

Tools for debugging and performance tuning parallel programs share the need to monitor program executions. This monitoring must be done without disturbing the phenomenon being observed. These tools differ in other capabilities they must support. Debuggers must provide the ability to interact with the program execution. Performance analysis tools must manage the large quantities of data that are collected from program executions.

The rest of this article is organized as follows. In the section entitled "Debugging Parallel Programs" we explain two mechanisms that must be in place for debugging parallel programs. We also describe how to detect race conditions in shared-memory parallel programs. Then, in the section entitled "Performance Tuning Parallel Programs" we describe how to performance tune parallel programs, a process that involves (1) monitoring program executions and (2) collecting and analyzing the run-time information.

DEBUGGING PARALLEL PROGRAMS

Debugging a program involves finding mistakes or flaws in the program. In general, determining program flaws statically (i.e., without running the program) is not possible. Hence, debugging involves executing the program, determining the point at which the program behavior departs from the expected, and working backward to find the flaw that causes the incorrect behavior. Debugging thus requires the programmer to determine the root cause of incorrect behavior and then correct it.

Erroneous program behavior can be determined by halting the program execution and examining the state of its components (threads or processes). When the state of the components do not match with what is expected, the program behavior is incorrect. For example, a property of the program could be that after reaching a particular point in the execution, the variable x is always positive. If the execution is halted after this point and x is negative, there is a flaw in the program.

Detecting incorrect program behavior is only the first step in the debugging process. This is because after determining a point in the execution at which the program behavior is incorrect, the programmer must work backwards to determine the flaw in the program that gives rise to the erroneous behavior. For instance, in the previous example, x may be negative because an earlier assignment to y was made incorrectly. This is the program flaw, and it leads to x being negative at a later point.

Typically, a cyclic debugging technique is used to determine the root cause of the flawed behavior (1). Once the presence of a bug has been established, the programmer reexecutes the application and halts it at successively earlier points. At each point, the programmer postulates properties about the program and verifies whether the program satisfies these properties by examining the internal state. The intent is to find the earliest point in the program execution where the state of the components deviates from the norm, thereby determining the root cause of the problem.

Thus, two mechanisms are required to debug a program. First, we need a mechanism to halt or stop a program execution. Second, we need a mechanism to reexecute—that is, replay a program execution. A sequential program can be easily halted by stopping the single component that is executing. Since sequential programs are usually deterministic (i.e., for a given input, the program execution always follows the same path and produces the same results), reexecution is also easily accomplished. For a parallel program, the presence of concurrently executing components complicates halting and replaying executions.

The mechanisms for halting and replaying parallel programs form the basis of parallel program debuggers. Using these two mechanisms, programmers can set breakpoints, halt the program execution, and examine the state of the components.

Halting a Parallel Program Execution

A *halt* command has a well-defined meaning for a sequential program. The single, running process can be halted and its state examined. In the case of a parallel program, it may not be feasible to ensure that the halt command will reach all components simultaneously. Consequently, the components may be halted in a different state from that requested by the programmer.

For a parallel program execution, the challenge is in halting the concurrent components in a consistent state. This is best illustrated by the following example. Let a process P reach a point A in the parallel program execution. If we want to halt P at this point, what are the constraints on halting the other processes? Ideally, we would like to halt the other processes at the same instant that P reaches A. However, instantaneously halting all the processes may not be possible. What can we do? If P were to be halted at point A, all of the events that have occurred on P prior to its reaching A must be propagated eventually to the other processes. Let no other process be allowed to proceed beyond a point that requires P to proceed beyond A. The global state of the parallel program execution that obeys this constraint is known as a *meaningful global state,* a concept introduced by Chandy and Lamport (2). Such states suffice for purposes of debugging, because from the viewpoint of process P, any computation on the other processes that takes place after process P reaches point A is dependent only on events that have already happened on process P.

When debugging a sequential program, programmers typically specify predicates about the program state. The debugger monitors the execution and halts it when the predicates are satisfied. The points where the program is halted are called *breakpoints*. When a programmer specifies a breakpoint for a parallel program, at least one of the executing processes will halt at the breakpoint. The remaining processes will eventually block when they reach a point where they require the halted processes to proceed before they can. The global state of the processes at this point will be consistent, and it can be examined by the programmer.

Replaying Parallel Program Executions

Cyclic debugging depends on the repeatability of program executions. When the program behavior is detected to be erroneous, cyclic debugging involves reexecuting the program so that it can be halted at an earlier point. Implicit in this technique is the requirement that the second execution be identical to the first.

Consider two executions of the same program for the same set of inputs. Informally, if the computation sequence and the sequence of values assigned to memory locations are the same in both cases, then the two executions are identical. For a deterministic sequential program, any two executions for the same input will be identical. In general, the presence of concurrency in a parallel program makes any two executions nonidentical.

The basic technique for replaying a parallel program execution relies on tracing the order of accesses to shared objects. First, a program execution is monitored, and the order of accesses to various shared objects is recorded. Then, when the execution needs to be replayed, the new execution is constrained to follow the recorded ordering on accesses to shared objects. A central theorem in execution replay is that if a second execution is constrained to follow the same access order as the first, the two executions will be indistinguishable. This *instant replay* technique was developed by John Mellor-Crummey and Tom LeBlanc (3,4). Note that any concurrent program can be modeled as a set of components (threads or processes) that interact or communicate through shared objects. For example, in message passing programs, the sending and receiving of messages can be modeled as accesses to a shared buffer.

Detecting Race Conditions

When the different components of a parallel program communicate through shared memory, the accesses to shared memory locations must be ordered. This ordering is achieved by *synchronizing* the different components. If the synchronization is not present or is defective, the access ordering may not be enforced, causing the program to contain time-dependent bugs known as race conditions (5). Race conditions are especially pernicious because of their dependence on time: The parallel program may behave perfectly normal for 99 executions, yet misbehave on the hundredth one.

In parallel programs that are designed to be deterministic, race conditions can cause the program to exhibit nondeterminism. For example, consider a sequential program that consists of a single loop. This program can be parallelized by assigning different iterations of the loop to different processors. If the synchronization required to perform this assignment is not present, the program may behave nondeterministically. This race condition is termed a general race.

Race conditions can also arise in parallel programs that are designed to be nondeterministic. As an example, consider a task-queue-based program where a set of worker processes picks tasks off a queue. This task-queue-based computation paradigm could be used in an automated chess-playing program. Now if two concurrent tasks can potentially access a shared location simultaneously and one of the accesses is a write, the outcome of the two accesses is time-dependent. The absence of a specific order between these two accesses can cause erroneous behavior, and it is termed a data race.

Race conditions are very hard to detect. Even in simple programs with no loops, race detection is NP-hard when certain synchronization constructs are allowed (for more information on NP-hardness, see COMPUTATIONAL COMPLEXITY THEORY). Most of the research in this area focuses on providing programmers with approximate information. For example, there are several techniques that will report a *nonempty* subset of the races in a program (6). This means that while some of the races that exist may be missed, a report that a program is race-free will be accurate.

From a practical viewpoint, there are several approaches that execute the parallel program and inform the programmer of potential race conditions for that particular data set. In other words, the information provided by these approaches is valid only for the particular data set with which the program is run. An exciting algorithmic development in this field is the research by Feng and Leiserson (7), who show that data races in deterministic Cilk programs can be detected in time that is essentially linear in the time it takes to run the Cilk program serially.

Tools for Debugging Parallel Programs

P2D2 is a portable parallel/distributed debugger developed at NASA Ames Research Center (8). P2D2 uses gdb, the Gnu debugger for sequential programs (9), as the underlying debugger. A front end manages the individual gdb sessions and communicates with them. Debugger commands are qualified with the set of processes where they should be applied. P2D2 provides the ability to set breakpoints, examine the state of the processes in detail, and single step the program execution. P2D2 has been ported to several target architectures, including the IBM SP-2, and parallel machines made up of networked SGI and Sun workstations.

A number of vendors are working on providing source-level debuggers for high-performance Fortran (HPF) programs (for more information on HPF, see PARALLEL AND VECTOR PROGRAMMING LANGUAGES). Typically, an HPF program is compiled to a message passing or shared memory parallel program by a data parallel compiler. In order for debug information from the resultant parallel program execution to be useful, information gathered from the execution must be correlated with the original source program, which is just an annotated sequential program. TotalView from Dolphin Interconnect Solutions Inc. is one commercial debugger that has some support for debugging HPF programs (10). The Pablo project at the University of Illinois is also working on this problem (11).

Sun Microsystems has developed a static debugger called LockLint to check for data races and improperly used locks in multithreaded programs (12). LockLint uses programmer-inserted annotations in the program to conduct its analyses. Several race detection tools have also been developed by the research community. RecPlay is a tool that enables the usage of cyclic debugging techniques for shared memory programs (13). For general races, RecPlay uses execution replay to enable the programmer to employ intrusive cyclic debugging techniques. RecPlay detects and reports data races back to the programmer. RecPlay has been implemented for SPARC-based systems. Perkovic and Keleher (14) have also developed an on-the-fly data race detection tool for shared memory parallel programs.

PERFORMANCE TUNING PARALLEL PROGRAMS

Developing a correct, bug-free parallel program requires a substantial amount of effort. Often, after investing this effort, the application performance may only be a small fraction of the peak system performance. Even for sequential programs, the complexity of present-day computer systems dictates the use of performance tuning tools. The additional complexity introduced by the presence of concurrency makes such tools a requirement for understanding and improving the performance of parallel applications.

There are three steps in the performance analysis process: execution monitoring, data collection, and data analysis. These correspond to monitoring the program execution, collecting information about the execution, and analyzing the information to determine the causes for poor performance.

Performance Metrics

The purpose of performance tuning is to reduce the running time of an application. Many indicators or metrics can be used in determining why the running time is higher than expected. These include time, the degree of concurrency, the memory access behavior of the program, and the amount and frequency of intercomponent (threads or processes) interactions.

A parallel program execution can be divided into phases. Each phase corresponds to a portion of the code, with the requirement that all components operate in the same phase at any given time. A natural metric to use is the time spent in different phases. Speeding up the phases of the program where most of the time is being spent can maximize gains. This principle of speeding up the common case is explained by *Amdahl's law* (15) and is illustrated by the following example. Let 75% of the running time of a program be spent in phase A, and let the remaining 25% be spent in phase B. Then, speeding up the execution of the code executed in phase A by a factor of three will speed up the overall execution by a factor of two. In contrast, speeding up the code executed in phase B by a factor of three will speed up the overall execution only by a factor of 1.2.

Another metric that can be used is the degree of concurrency exhibited over the course of the execution (16). Bottlenecks in the program correspond to periods in the execution when not all the processors are busy. By keeping track of the number of processors at work in different parts of the program, these bottlenecks can be pinpointed. This information can be used to change the program and increase the parallelism that can be extracted.

The time spent accessing program data structures is another important metric (17). Many times, a bottleneck may not be attributable to any particular point in the program execution. It may be the memory accesses to a particular data structure, distributed throughout the program, that are causing the performance degradation. A metric that targets the memory access behavior of a program can be used to detect this scenario.

The amount of interprocess interactions, along with the time spent in them, is another useful metric (18). Interprocess interactions such as communication and synchronization are inherent in a parallel program, and they constitute the mechanism by which different program components communicate. These interactions typically cost a few orders of magnitude more than local memory accesses. Very frequent interprocess interactions may indicate that a different algorithm that requires less frequent communication should be used.

Execution Monitoring

Execution monitoring consists of observing the parallel program execution in order to collect data about interesting events. This can be achieved in one of two ways. First, we can use a hardware monitor to collect information about specific events. Alternately, we can add software instrumentation instructions to the parallel program in order to monitor the execution. In both cases, the monitoring process produces data, which must be collected and analyzed.

Hardware Monitoring. Hardware monitoring can be done in two ways. We can use external hardware devices such as logic analyzers or bus analyzers to record specific events. Alternately, we can use processors that have onboard (internal) performance monitoring counters to watch for interesting events.

Programs typically execute millions of instructions in each second. External hardware devices such as bus and logic analyzers can collect information about the events caused by the execution of many of these instructions. For example, the external analyzers can be programmed to watch for the bus transactions that may be caused by load and store instructions. External devices can be programmed in more flexible ways than the onboard event counters. However, external devices suffer from a very significant disadvantage: They require access to the actual hardware on which the program executes.

Most modern processor architectures (e.g., the SGI R10000, Digital Alpha, and Intel x86) possess the capability to monitor several events through programmable event counters that are internal to the processor (19). As these counters are already integrated with the processor logic, no extra hardware is needed to use them. Typically, these counters can be used to measure events such as cache misses, branch mispredictions, translation lookaside buffer (TLB) misses, and so on. At periodic intervals, the values of these counters can be recorded to obtain information about the program execution.

Hardware monitors can observe execution events at a very fine level of detail (i.e., at the level of individual instructions). However, since the monitoring infrastructure is separate from that used for normal program execution, hardware monitoring does not perturb the program execution.

One drawback of fine-grained monitoring is managing the enormous amount of performance data that gets generated. This information overload problem can be alleviated by sampling the program execution and recording only selected events. Typically, a few thousand samples for every second of program execution are collected. Depending on the kind of performance measurement counters provided by the architecture, the samples can provide information on the time spent in different parts of the program, on cache misses, on branch mispredictions, and so on. Introducing an element of randomness when choosing the sampling interval ensures that the sampling interval does not "beat" with phases in the program execution (19).

Software Monitoring. Software monitoring requires *instrumenting* the program being executed. This is done by adding instructions to the program, which monitor the program execution and gather information about selected events. For example, we can obtain information about the memory accesses of the program by instrumenting the loads and stores in the program.

A clear advantage of software instrumentation is its flexibility. Event monitoring can be much more selective since it is done entirely in software. For example, it is fairly easy to ensure that only memory accesses to a particular data structure get recorded. The disadvantage of software instrumentation is that it can cause the program execution to slow down. Worse still, the instrumentation can slow down the concurrent processes at different rates, perturbing the execution behavior of the application. This perturbation may cause the performance tuning data to be collected from an execution that has very little in common with the typical behavior of the application.

The run-time overheads of software instrumentation can be alleviated by dynamically instrumenting the parallel program while it is executing (20). This involves deferring the decision on what to instrument until execution time, and then periodically modifying the running program. By monitoring the program execution, instrumentation can be added selectively based on perceived bottlenecks. As in the case of hardware monitoring, sampling can also be used here to reduce the perturbation effects. Sampling involves using either hardware cycle counters or the system clock to generate interrupts at periodic intervals (for more information on interrupts, see INTERRUPTS).

Perturbation can also be reduced using execution replay (see section entitled "Replaying Parallel Program Executions"). Instrumentation is first added to the program to record the partial order on all interprocess interactions. This instrumentation is fairly lightweight and perturbs the execution only slightly. The recorded partial order information is then used to replay the execution. The replay run is fully instrumented, but is forced to follow the same path as the first record run.

Data Collection

Once events during execution have been monitored, the next step is to collect the data.

External hardware devices can record this information in external storage. On-chip event counters often use interrupts to collect data related to specific events. When a performance counter overflows, it generates an interrupt which is serviced by the processor. The interrupt software can determine the context in which the interrupt was triggered and records this information in a buffer set aside for this purpose (19). Software instrumentation techniques also record event data into preallocated buffers. When the buffer gets full, it is stored on disk for later processing.

Both strategies use special formats when recording the data in order to reduce the perturbation caused by the data collection process. Carefully designed data formats are used to reduce the amount of data accesses that must be made. Using well-designed data structures can also reduce the number of cache misses that may be encountered when storing the recorded data.

Data Analysis

The most challenging task in the performance tuning process lies in analyzing the collected data. Information gathered from the program execution must also be correlated with the source code for it to be useful.

The data analysis problem is especially difficult. Data collected at run time is often very closely related to the hardware. For example, cache misses and TLB misses represent information at the level of the machine architecture. Unless this raw information is processed, the responsibility of digesting it will lie with the programmer.

Correlating the performance information with the program code and data structures is also not easy. Often the program being performance tuned is very different from the one written by the programmer. For example, a sequential HPF program is compiled to a message passing parallel program before execution (for more information on this, see PARALLEL AND VECTOR PROGRAMMING LANGUAGES). Performance data gathered from the message passing executable must be correlated with the original HPF program (which has no message passing calls), in order for it to be useful (11).

Visualization is a useful technique to cope with the large quantity of data produced by long-running programs (11). The data are usually plotted in different formats to enable the programmer to easily spot trends or patterns that indicate a performance problem. The plots can be as simple as a listing of the time spent in different functions, or as complicated as a graph that shows the amount of interprocess contention on a per-data structure basis. The end goal is to make it easier for the programmer to detect trends in the data.

A new research direction advocates reducing the burden on the programmer by moving away from data presentation to automatically analyzing run-time data (21). This approach focuses on *prescribing* solutions to the performance problems detected in the program. Prescriptive feedback can be provided by casting the problem of improving program performance as an optimization procedure on a model of the system on which the program is being executed. Automatically solving the optimization problem allows the tool to directly prescribe how the program should be changed.

Current Performance Tuning Tools

Paradyn is a performance measurement tool that dynamically instruments the program being traced and searches the execution for bottlenecks (20). The base metric used is the time spent in various operations. A search model, consisting of a set of hypotheses about potential performance problems, is incorporated within Paradyn. During execution, Paradyn refines this set, until a few hypotheses accurately reflect the performance bottleneck in the program. One problem with this approach is that performance bottlenecks cannot often be attributed to any one *point* in the code. For instance, the performance degradation might be caused by accesses to a particular data structure whose references are distributed through the program.

MemSpy presents data-oriented statistics for tuning the memory performance of parallel programs (17). MemSpy simulates the parallel program and measures different types of cache misses. It then correlates this information to code and data structures in the program. The programmer typically uses this information to rearrange the data structures in the program.

Rx is a tool that focuses on the automatic analysis of data gathered from parallel program executions (21). Instead of leaving the programmer with the task of digesting execution data, Rx analyzes the data and *prescribes* solutions to performance problems. It does so by having a model for the underlying system, and solving an optimization procedure on the model.

The Digital Continuous Profiling Infrastructure (DCPI) is a robust performance tuning tool for Digital Alpha platforms (19). DCPI samples the program execution using the Alpha processor's performance counters to collect information about execution events. A suite of auxiliary tools can be used to display and summarize the collected data. Sampling permits an extremely low overhead (around 2%) for the profiling process. Furthermore, DCPI profiles the entire system, including both user level and operating system code.

Intel's Visual Tuning Environment (VTune) is a commercial performance tuning tool for Intel platforms running Windows-NT and Windows95 (22). Using time-based and event-based sampling, it noninvasively monitors both user level and operating system program activity. Currently, VTune targets only time; that is, it only supplies information on the time spent in different parts of the program.

BIBLIOGRAPHY

1. M. Garcia and W. Berman, An approach to concurrent systems debugging, *Proc. 5th Int. Conf. Distributed Comput. Syst.,* 1985, pp. 507–514.
2. K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *IEEE Trans. Comput.,* **3**: 63–75, 1985.
3. T. J. LeBlanc and J. M. Mellor-Crummey, Debugging parallel programs with Instant Replay, *IEEE Trans. Comput.,* **C-36**: 471–482, 1987.
4. J. M. Mellor-Crummey, Debugging and analysis of large-scale parallel programs, PhD thesis, Univ. of Rochester, 1989. This dissertation has an excellent introduction on the various issues that must be addressed when debugging parallel programs.
5. R. H. B. Netzer, Race condition detection for debugging shared-memory parallel programs, PhD thesis, Univ. of Wisconsin, Madison, 1991. This dissertation provides an introduction to race detection in shared-memory parallel programs. It is also available as technical report CS-TR-91-1039 from the University of Wisconsin.
6. D. P. Helmbold and C. E. McDowell, Race detection—Ten years later, in M. L. Simmons et al. (eds.), *Debugging and Performance Tuning for Parallel Computing Systems,* New York: IEEE Computer Society Press, 1996. This book has a collection of papers that form an excellent starting point for several of the topics addressed in this article.
7. M. Feng and C. E. Leiserson, Efficient detection of determinacy races in Cilk programs, *Proc. 9th ACM Symp. Parallel Algorithms Architectures,* 1997, pp. 1–11. More information on race detection work in Cilk is available from http://theory.lcs.mit.edu/~cilk/papers.html.
8. R. Hood, The *p2d2* Project: Building a Portable Distributed Debugger, *Proc. 1996 ACM SIGMETRICS Symp. Parallel Distributed Tools,* 1996. For more information on *p2d2*, see http://science.nas.nasa.gov/Groups/Tools/Projects/P2D2.
9. Free Software Foundation Inc., *Debugging with gdb: The Gnu Source-Level Debugger,* 1997. This manual, gdb software, and more information on other Gnu projects can be found at ftp://prep.ai.mit.edu/pub/gnu.
10. Dolphin Interconnect Solutions Inc., *TotalView Multiprocess Debugger User's Guide,* 1997. For more information on TotalView, see http://www.dolphinics.com/tw/tvover.htm.
11. V. S. Adve et al., An integrated compilation and performance analysis environment for data parallel programs, *Proc. Supercomput. 1995.* pp. 1370–1404. This paper is online at the Pablo project website: http://www-pablo.cs.uiuc.edu.
12. Sun Microsystems Inc., *SPARCworks/iMPact: Tools for Multithreaded Programming,* 1995. Catalog number: 802-3542-10.
13. M. Ronsse and K. De Bosschere, An on-the-fly data race detector for REC-PLAY, a record/replay system for parallel programs, *Poster session 16th ACM Symp. Operating Systems Principles,* 1997. Also available at http://www.cs.washington.edu/sosp16/wipWWW.html.
14. D. Perkovic and P. Keleher, Online data-race detection via coherency guarantees. *Proc. Second USENIX Symp. on Operating System Design and Implementation,* 1996.
15. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* San Mateo, CA: Morgan Kaufmann Publishers, 1990.
16. T. E. Anderson and E. D. Lazowska, Quartz: A tool for tuning parallel program performance, *Proc. Int. Conf. Measurement Modeling Comput. Syst.,* 1990.

17. M. Martonosi, A. Gupta, and T. E. Anderson, Tuning memory performance of sequential and parallel programs, *IEEE Comput.,* **28** (4): 32–40, 1995.
18. R. Rajamony and A. L. Cox, Performance debugging shared memory parallel programs using run-time dependence analysis, *Proc. 1997 ACM SIGMETRICS Int. Conf. Measurement Modeling Comput. Syst.,* 1997.
19. J. M. Anderson et al., Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.,* **15** (4): 357–390, 1997. For more information on DCPI, see http://www.research.digital.com/SRC/dcpi.
20. B. P. Miller et al., The Paradyn parallel performance measurement tools, *IEEE Comput.,* **28** (11): 37–46, 1995. More information on the Paradyn project including software distributions can be found at http://www.cs.wisconsin.edu/paradyn.
21. R. Rajamony, *Prescriptive Performance Tuning: The Rx approach,* PhD thesis, Rice Univ., Houston, TX, 1998. For more information on Rx, see http://www.cs.rice.edu/CS/Systems/Rx.
22. Intel Inc., *VTune: Developers Manual,* 1997. More information on VTune, including demonstration versions of the software, can be obtained from http://support.intel.com/support/performancetools/vtune.

Reading List

Cherri M. Pancake, *Parallel Debugger Bibliography.* This is an online bibliography, available at http://www.cs.orst.edu/~pancake/papers/biblio.html, and is an excellent place to find papers and dissertations on the subject of parallel and distributed debuggers.

The Parallel Tools Consortium, This consortium has a Web site: http://www.ptools.org, which is a wonderful resource for getting information about the latest parallel tools projects. In particular, they have a Web page that lists a set of parallel tools projects around the world.

RAMAKRISHNAN RAJAMONY
IBM Austin Research Laboratory

ALAN L. COX
Rice University