Techniques for Performance Measurement of Parallel Programs

Jeffrey K. Hollingsworth[1], James E. Lumpp, Jr.[2] and Barton P. Miller[1]

[1] Computer Sciences Department, University of Wisconsin, Madison WI 53706, USA
[2] Department of Electrical Engineering, University of Kentucky, Lexington, KY 40506-0046, USA

Abstract. Programmers of parallel systems require high-level tools to aid in analyzing the performance of applications. Performance tuning of parallel programs differs substantially from the analogous processes on sequential architectures for two main reasons: the inherent complexity of concurrent systems is greater, and the observability of concurrent systems is complicated by the effects instrumentation can have on the behavior of the system. The complexity of parallel architectures combined with non-determinism can make performance difficult to predict and analyze. Many approaches to help users to understand parallel programs have been proposed. This paper summarizes the problems associated with creating parallel performance measurement tools and describes some of the systems that have been built to solve these problems.

1 Introduction

The primary reason for writing parallel programs is speed. Once a parallel program has been written, and the errors have been eliminated, programmers generally turn their attention to the performance of their program. Most application programmers gauge the performance of their program by turnaround time not throughput. Performance measurement tools exist to provide insight to programmers to help them understand why their programs do not run fast enough. For these tools to be effective, they need to collect data about the application, the operating system, and the hardware, and synthesize it in a way to let programmers concentrate on getting their work done.

As the number and computational power of processors in parallel computers increase, the volume and complexity of performance data that must be gathered can explode. This wealth of information is a problem for the programmer who is forced to navigate through it, and for the tools which must store, process, and present it. This large volume of the performance data also has a high cost associated with collecting it. We divide the problem of measuring the performance of parallel programs into two parts: performance analysis and instrumentation. Performance analysis tries to sift through the huge volume of statistics available about a parallel program's execution and provide useful information to the programmer. The performance instrumentation problem focuses on how to efficiently collect enough information about a computation. A variety of different approaches have been designed to address these problems. We present an overview of some these approaches and try to compare and contrast their relative advantages and disadvantages.

2 Performance Analysis Tools

The range of potential bottlenecks possible in parallel programming dramatically increases the need for performance analysis tools compared to sequential machines. Parallel machines not only have many copies of the resources (CPU, registers, IO systems) that can cause bottlenecks in sequential programs, but they also include unique features such as interconnection networks and coherency protocols that can contribute to performance problems. In this section, we concentrate on techniques designed to help a programmer to improve the execution time of their program on a particular machine. We will not address the related problem of system tuning (tuning a parallel machine and operating system for a particular workload) that is also dramatically more complicated on parallel machines. However, it is worth noting that many of the tools described below are also appropriate for this problem.

The complexity and diversity of the hardware being used in today's parallel computers has led to a variety of different approaches in parallel performance tools. We divide these approaches into three categories: performance metrics, search based tools, and performance visualization. However, in practice, most complete systems incorporate different approaches from several of these categories.

2.1 Performance Metrics

We define performance metrics as any statistic about a parallel computation designed to help programmers reduce the running time of their programs. Profile metrics are performance metrics that associate a single value with components of a parallel application (often procedures). Metrics are often presented as sorted tables so that the most important items are at the top of the display. Performance metrics originated in sequential programming as a simple profile of the CPU time consumed by each procedure in a program. The UNIX utility CPU time profiler Prof [23] is an example of such a tool. A logical extension of this technique to parallel programs is to aggregate the CPU time profiles from each process (or thread) to provide a single profile for a parallel program. The profiling environment on the Connection Machine [68] provides this type of metric.

Simply extending sequential metrics to parallel programs is not sufficient because, in a parallel program, improving the procedure that consumes the largest amount of time may not improve the program's execution time. Inter-process dependencies in a parallel program influence which procedures are important to a program's execution time. The differences between parallel program performance metrics can be seen in the way they account for these interactions. We describe the various metrics in terms of a graph of the application's execution history that incorporates both these inter-process dependencies as well the sequential time. This graph is called a Program Activity Graph (or PAG). Nodes in the graph represent significant events in the program's execution (e.g. lock and unlock operations, procedure calls and returns). Arcs represent the ordering of events within a process or the synchronization dependencies between processes. Each arc is labeled with the amount of process and elapsed time between events. Figure 1 shows a simple PAG for a parallel program with three processes. The solid arcs represent useful CPU time intervals. The dashed lines indicate non-useful CPU time from activities such as spinning on a lock.

One of the first metrics specifically designed for parallel programs was Critical Path Analysis [72]. The goal of this metric is to identify the procedures in a parallel program that are responsible for its execution time. The Critical Path of a parallel program is the longest CPU time weighted path through the PAG. Non-productive CPU time, such as spinning on a lock, is assigned a weight of zero. The Critical Path Profile is a list of the procedures along the Critical Path and the time each procedure contributed to the length of the path. The time assigned to these procedures determines the execution time of the program. Unless one of these procedures is improved, the execution time of application will not improve.

Although Critical Path provides more accurate information than prof, it does not consider the effect of sub-critical paths (i.e. second and third longest paths) in limiting the improvement possible by fixing a component on the Critical Path. An extension to Critical Path called Logical Zeroing [51] addresses this problem. This metric sets the weight of all instances of a procedure in the PAG to zero, and computes the length of the new Critical Path. The difference between the old and new Critical Paths is an indication of how much the execution time of the program can be improved by fixing the selected procedure. Critical Path provides detailed information about how to improve a parallel program, but building the PAG and calculating the metric requires significant space and time.

The Quartz NPT [1] metric also addresses the inter-process dependencies that limits Prof's usefulness. NPT normalizes CPU time for each procedure based on the effective parallelism while that procedure is executing. No time is accumulated for procedures that are blocked due to synchronization. As a result, a procedure that executes serially is assigned a higher value (indicating it is more important) than a procedure with the same CPU time that executes during a parallel section of the code. This metric provides more accurate information than Prof. Unlike Critical Path, NPT does not use synchronization dependencies directly. This makes it easier to compute NPT while the program is executing. However, because NPT only looks at instantaneous information it does not consider the effect of procedures prior to a bottleneck on that bottleneck. For example, if a process that is load im-balanced contains several procedures, NPT would give a higher value to the latter ones even though improving any procedure would reduce the load imbalance.

The first three metrics focused on finding CPU time bottlenecks. However, another source of bottlenecks in parallel programs is the memory hierarchy. By necessity, the memory hierarchy in parallel computers is more complex than in sequential machines. Shared memory parallel computers typically have several levels of caches and some form of cache coherency protocol. Effective use of this memory hierarchy is essential for high performance applications. One way to try to understand the impact of the memory system on a parallel program was used in MTOOL [21]. The technique compares the observed

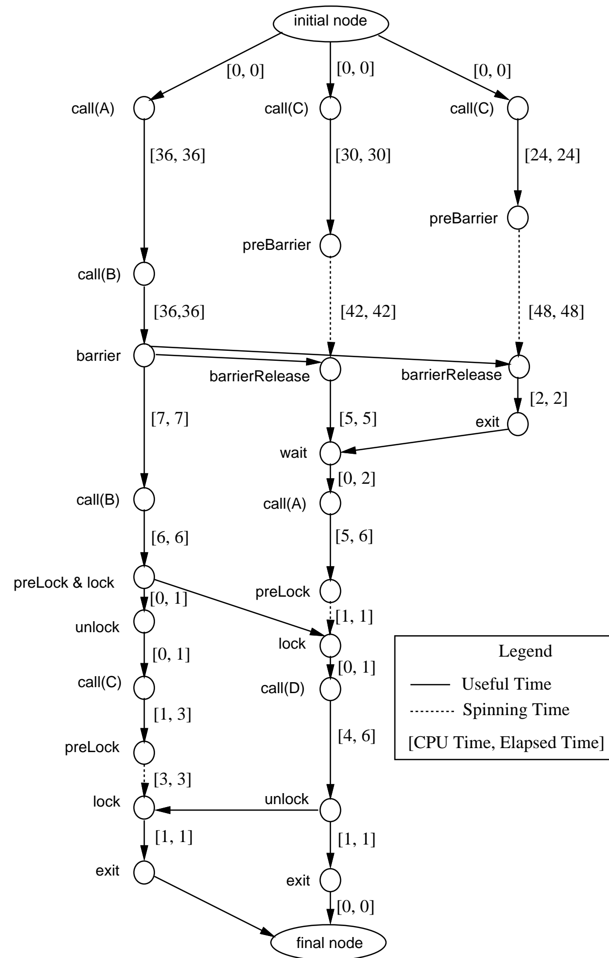

Fig. 1. A sample Program Activity Graph. Nodes represent interesting events in the application (e.g. procedure calls and message passing). The ordered pair indicates the CPU and elapsed times between events.

execution time for each sequential region of the PAG to the ideal execution time if all memory references were serviced by the fastest part of the memory hierarchy. Differences between the observed and predicted execution times are due to instruction stalls that result from memory contention and extra time to access non-local memory. Information for each procedure can be combined to produce a table showing which procedures had the greatest impact on the memory system. An advantage of this approach is that users are able to get information that is not available directly from other systems. Unfortunately, the metric requires an accurate model of instruction costs to compute the predicated performance of each basic block.

Rather than trying to find a single metric to characterize the entire program, many tools provide sorted profiles of the utilization of several different resources. Some commonly profiled resources are: CPU utilization, synchronization time, disk operations, vectorization, and cache performance. INCAS [24], ANALYZER/SX [35], and JEWEL [37] are examples of this approach. An advantage of multiple metric tools is that they can provide information about several types of bottlenecks. In addition, the particular resource that is being overused can be isolated. However, this forces the user to select the appropriate resources to profile. For large programs running on massively parallel systems, there will be thousands (if not millions) of combinations of resources and program components to consider.

Building a tool that includes all possible metrics a user might want is difficult. One approach to this problem is to make a tool extensible and permit users to create their own metrics. An example of a tool that provides this feature is IPS-2 [29] which permits users to create new metrics as algebraic expressions

of previously defined metrics. For example, IPS-2 does not have a built-in metric to indicate what share of the actual CPU time used went to each process. However, it is possible to define this metric in terms of a process' useful CPU time plus its spinning time divided by the total user CPU time on the system. This simple expression language provides an easy way to extend a performance tool. However, the types of metrics that can be defined by the user is limited by the base metrics provided by the tool developer.

Another way to make performance tools extensible is to provide a toolkit for building metrics. PPUTT [16] uses this approach by providing a set of analysis modules and a well defined interface to an event stream. User's create new metrics by combining predefined analysis modules or writing their own. Extensible systems provide a powerful tool for a sophisticated user, and a convenient way for tool developers to explore new metrics. However, many users do not know what additional metrics might be useful and are unable to use this functionality.

The major problem with having many metrics is knowing which one to use. Hollingsworth and Miller [31] compared several different metrics (including Critical Path, Logical Zeroing, and NPT) and concluded that there was no single best metric. In fact, the guidance from several of the metrics conflicted. However, they were able to characterize the types of programs where each metric would be useful. This information is valuable to the programmer, and could help them select appropriate metrics. In addition, different metrics have different costs of computation. Sometimes a cheaper metric (e.g. prof) is sufficient, and so there is no need to calculate complex metrics. Unfortunately, no current tools incorporate these meta-metrics.

2.2 Search Based Tools

Performance metrics provide useful guidance for some types of bottlenecks; however, since different metrics are required for different types of bottlenecks the user is left to select the one to use. To provide better guidance to the user, rather than providing an abundance of statistics, several tools have been developed that treat the problem of finding a performance bottleneck as a search problem. These systems attempt to both identify the problem (descriptive) and to give advice on how to correct it (prescriptive).

AtExpert [36] from Cray Research uses a set of rules to help users improve FORTRAN programs written with the Cray auto-tasking library. The auto-tasking library provides automatic parallelism for FORTRAN programs; however, there are a number of directives that can that greatly affect performance. AtExpert measures a program that has been auto-tasked and attempts to suggest directives that would improve the performance of the program. Since, the tool works on a very specific programming model, FORTRAN programs on small scale shared-memory multi-processors, it is able to provide precise prescriptive advise to the user.

Crovella and LeBlanc's predicate profiling [15] provides a search system to compare different algorithms for the same problem as well as the scalability of a particular algorithm. They define a set of rules that test for possible losses in performance of a parallel program. They classify losses due to: load imbalance, starvation, synchronization, or the memory hierarchy. All values are calibrated in terms of cycle times. The information is displayed as a bar chart showing where the available cycles went (both to useful work and various sources of loss). Information is presented for the entire application, which provides descriptive information about the type of bottleneck. However, they do not include prescriptive information about how to fix the problem or an indication of which procedure contains the bottleneck.

One way to search for performance bottlenecks is to have the programmer specify assertions about the expected performance of their program. For example, the programmer can express the expected CPU utilization or a cache miss tolerance. The PSpec language [56] uses this approach. Using the tool is a three step process. First the programmer specifies their performance assertions. Second, the application is run with monitoring code enabled. Third, after the application has executed, a post-mortem tool called the checker processes the logs and looks for assertion failures. Because performance assertions come from programmers, they can contain precise descriptions of the expected performance. However, writing assertions does create additional work for the programmer. Also, they provide no prescriptive information to the user.

Another approach is to provide a search system that is independent of the programming model and machine architecture. The Performance Consultant [30] uses a hierarchical three axis search model (the "why", "where", and "when" of a performance bottleneck). The "why" axis represents hypotheses about potential bottlenecks in a parallel program (i.e. message passing, computation, IO). The "where" axis

defines a collection of resource hierarchies (CPU, interconnect, disk) that could cause a bottleneck. The "when" axis isolates the bottleneck to a specific phase of the program's execution. A unique feature of the Performance Consultant is that it searches for bottlenecks while the program is executing. This requires an adaptive style of instrumentation, but it can greatly reduce the volume of performance data that needs to be collected. Only the performance data required to test the current hypothesis for the currently selected resources need be collected. In addition to finding bottlenecks, the Performance Consultant includes an explanation system to both textually and graphically relate the performance bottleneck back to the user.

2.3 Visualization

A different approach to finding performance problems in parallel programs is to provide pictures to help programmers visualize the problem. We consider parallel program visualization to be any visual or aural tool that tries to provide the user feedback about how a parallel program is running. This definition includes tools to visualize low level activity on a parallel machine, algorithm animation, and auralization.

A basic visualization is a representation of the physical machine. For example, MAP [11], represents the memory (really FORTRAN arrays) in a parallel computation as a two dimensional grid. Whenever a memory location is referenced, its location in the grid is highlighted, and then slowly decays back to its original color (i.e. similar to the phosphor in a CRT screen). Frequently accessed memory locations get refreshed frequently and appear as a constant glow, while transient memory references are only flashes of light. This visualization is simple, but effective for identifying memory contention.

Another tool that lets users view a parallel computation is PIE [42]. PIE provides a color graph consisting of coded bars representing a time-line. There is one time-line per CPU and the color of the time-line indicates what the CPU was doing at that point in time (e.g. useful work, blocked, spinning). The tool can also show which procedures are executing by assigning a different color to each procedure in the program. In addition, time-lines for other levels of abstraction (e.g. individual processes) can also be displayed. These displays are colorful and provide a feel for a program's execution, but it is difficult to interpret the pictures.

Similar to PIE, PF-View [70] illustrates the status of a parallel computation at multiple levels of abstraction. However, rather than presenting information as a time-line, it uses animation and graphical representations of program abstractions. The tool also provides a text window which shows the source code for the program being animated. This approach provides a display that is easier to understand and relate to the original code than PIE. A limitation of this tool is that the user is forced to watch a graphical execution of their program looking for the bottlenecks, rather than having the tool show them the problem.

Another common visualization is an illustration of the synchronization patterns in a parallel computation. This is essentially drawing the Program Activity Graph shown in Figure 1. The MAD debugger [73] provides what the authors call a "causality graph". This is a graph with each thread indicated by a vertical line, and the inter-thread communication and synchronization shown as diagonal lines. This picture shows a parallel computation in great detail. Unfortunately the programmer must scan a complex graph trying to find the relevant events. In addition, scaling this visualization to massively parallel machines would be difficult.

The visualization tools discussed so far use a single visualization. Several tools have been developed that incorporate multiple visualizations. A good example of this type of tool is Paragraph [27] developed by Mike Heath and Jennifer Etheridge. Paragraph supports over twenty different types of displays. Many of these displays can be configured to plot values for different resources (e.g. messages and CPU time). This approach provides a vast number of different ways to display performance data. Unfortunately not every visualization is useful for every program, and the user is left with the formidable task of selecting appropriate displays and resources.

One of the most difficult parts of creating visualization tools is selecting what to visualize. A solution to this problem is being explored in two systems, Pablo [58] and JEWEL [37]. Both of these systems contain toolkits for building visualization modules. The toolkits provide drawing primitives, event streams, and filtering modules. The user constructs visualizations from these building blocks. This method permits an almost unlimited number of visualization modules to be built. However, these systems provide no assistance to the user in creating appropriate visualizations to find their performance bottleneck.

Another type of visualization is algorithm animation. Algorithm animation tries to graphically represent a program's execution at a level of abstraction similar to the programmer's mental model of the

program. According to Pancake and Utter [55], if the programmer can see the program the way they think about it, rather than how it is represented on the physical machine, it will be easier to find the problem. The difficult part is mapping events on the physical machine back to an abstract model. Two approaches have been tried to this problem.

The first approach is to provide the programmer a library of graphics primitives and let them make calls to animation routines from their program. IVE [18] and Voyeur [65] use this approach. This provides great flexibility because users are able to create a precise animation that illustrates what their program is doing. However, it places the additional burden on the user of creating and debugging their animation while they are trying to create their program.

A second approach, used in Belvedere [32], is to provide an event stream and let the user define reduction and animation operations over it. Belvedere provides a higher level interface to the programmer since they don't need to identify the events in their program directly. In addition, it is possible to build up a library of standard visualizations to make the process of creating animations easier. However, it still requires that the programmer define their animation and maintain it along with their source program.

A variation on algorithm animation is auralization [17, 63, 66]. Instead of representing a program graphically, auralization animates it using sound, or a combination of sound and pictures. For example, message sends could be mapped to tones on the left channel and receives to the right channel. This permits programmers to get information about the frequency of messages, message flight time, and unmatched send & receive pairs. This technique provides an extra dimension to represent the program, but is still limited by the work required by the programmer to select appropriate operations for auralization.

A major problem with visualization systems is that they are constrained by a tradeoff between relevance and re-usability. To provide information at a sufficiently high level to be useful, many visualization modules are specific to a single programming paradigm or worse yet a single application. To be useful for a wide variety of applications, visualization modules need to be based on low level events (e.g. message passing) that are common to a large number of programs. The single visualization tools try to be useful for a wide variety of applications. The multiple visualization and toolkit systems try to provide more specific information but are forced to incorporate large number of displays.

3 Instrumentation

For parallel computing tools to be useful, they need to collect sufficiently detailed data from a parallel program without perturbing it such that the data collected is not representative of the un-instrumented program. Performance tuning, debugging, and testing all require instrumentation to provide information about the execution. However, for each of these usages the type of information required varies. Performance tuning requires information about utilization of architectural features while debugging (and testing) requires detailed information concerning the global state of the computation. Many approaches have been tried to solve this problem: software traces, software event filters, and even custom hardware.

3.1 Program Instrumentation

One way to collect data about an application is to instrument the application executable so that when it runs it generates the desired information as a side effect. There are many ways to do this type of instrumentation. It can be inserted into the application source code directly, automatically by the compiler, placed into runtime libraries, or even by modifying the linked executable. Each of these approaches has advantages and disadvantages. Figure 2 shows the stages of a compilation, and a sample of the places that different tools insert their instrumentation.

Collecting data via the runtime library (e.g. the C library) provides an easy way to collect data about the interactions between processes. IPS-2 [51] and Cedar tracing [46] use this approach. An advantage of runtime library instrumentation is that it does not require the user to modify their program, just re-link it with special flags. However, application specific events are not visible at this level. For example, while tuning a database system the user might wish to trace commits and aborts.

Another approach is to directly instrument the user's application. AE [38] and CONVEX [25] use a modified compiler to insert instrumentation at the desired location. This provides access to information that is only available to compilers (e.g. data and loop dependencies). However, to add instrumentation

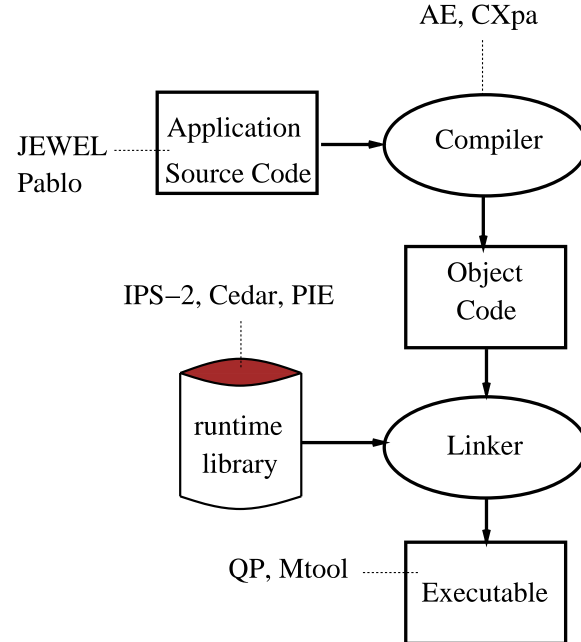

Fig. 2. Inserting Instrumentation into a program. Instrumentation can be inserted at any stage of the compilation and linking process.

into a compiler requires access to the source code for the compiler, and each compiler must be separately instrumented.

Alternatively the instrumentation can be inserted directly into the executable image. QP [4] and MTOOL [21] use this technique. Binary instrumentation permits data to be collected in a language independent way, and makes it easy to collect information about the runtime libraries in the same way that application data is collected. In addition, since re-compilation is not necessary to insert instrumentation, it is easier to instrument large multi-module programs which are time-consuming to re-compile. It is also possible to insert instrumentation into programs and libraries for which the source code is not available. Using direct instrumentation of the application (as compared to kernel or library instrumentation) makes it is possible to collect finer grained information. For example, loop level or basic block data can only be collected with these techniques.

It is also possible to hand modify an application to insert calls to collect performance data. JEWEL [37] uses this approach. This is the easiest way to implement an instrumentation system, and it provides the user a great degree of selectivity. Only what the user desires to instrument gets collected. However, for large systems that require large amounts of instrumentation, this is a tedious process.

Finally, it possible to use source to source translation of the program. This provides a means to automatically insert instrumentation before the program is compiled. An advantage of this approach is that instrumentation can be inserted without needing to modify the compiler. However, the data dependencies are not available, and it of course requires that the program be re-compiled to collect the data. Pablo [58] uses this technique.

Most monitoring systems use some degree of software instrumentation. Because of the difficulty of inferring process level information from the hardware view of the processor state, pure hardware systems are rare. Software solutions are also desirable to avoid the high cost of dedicated hardware. However, the consequence is increased intrusion of the collection activity on the monitored systems.

3.2 System Instrumentation

Sometimes collecting data by creating modified application processes is not desirable. Another software based approach is to collect data via dedicated monitoring processes or via the operating system. This approach decouples the monitored system from the application process at the expense of easy access

to application data structures. Figure 3 shows some of the ways in which programs can be externally monitored. This section describes the software approaches shown in the figure and the next section outlines options for additional hardware to support monitoring.

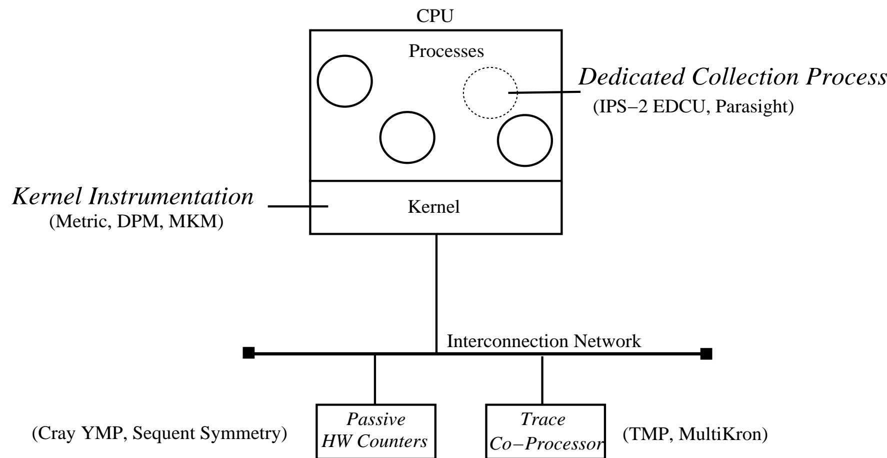

Fig. 3. Inserting Instrumentation into a system. External monitoring can be accomplished either via software or special purpose hardware.

A common approach to collect performance data is to modify the operating system kernel to collect trace data and send it to a file or data reduction node. Kernel based instrumentation has two advantages. First, it is easy to instrument programs, all that is required is a single kernel call to turn on the instrumentation. Second, some types of data (e.g. context switches and page faults) are difficult (if not impossible) to collect from user space or via hardware. However, kernel based approaches suffer two limitations. First, modifying a kernel is time consuming, error prone, and requires users to run a modified kernel to use the tool. Second, some types of information (e.g. procedure calls) is not available from kernel based instrumentation. Several systems have used this style of instrumentation. Smith's Spider debugger [62] used kernel based instrumentation to track all inter-process communication, but it only worked on a single machine. Miller's Distributed Program Monitor [52] traces inter-process message passing in a multi-processor distributed environment. The Mach Kernel Monitor [41] instruments context switches to trace the state of processes through time.

If modifying the operating system is not feasible or desirable, monitoring can be accomplished via dedicated data collection processes. The External Data Collection facility of IPS-2 [29] provides this capability. Information from the operating system is gathered by the collector processes and reported to the user. Most operating systems collect many statistics and make this information available via system calls. Hence much of the data that can be collected by directly instrumenting the operating system is available using external data collection.

If information specific to an application is desired, a hybrid approach, is possible. For example, Parasight [2] uses monitoring threads (parasights) that run on idle processors to maintain monitoring information. "Light-weight" instrumentation in the target nodes is used to write monitoring information to shared memory while the Parasight threads read the data. Threads can be created dynamically and instrumentation can be included or removed dynamically during execution of the program.

3.3 Hardware Instrumentation

Completely non-intrusive detection requires dedicated, stand-alone hardware instrumentation for the recognition of events. Additionally, a non-intrusive hardware monitor for a CPU with no special support for monitoring must passively monitor the CPU busses. Several CPU bus cycles, each representing state changes of the *processor*, will occur per instruction executed in the monitored system. Therefore, a change of state of the active process or an event in the active process will correspond to several processor state changes. A non-intrusive monitoring system must be capable of inferring process level events

from processor level events that may be produced by the target processors at high rates. To monitor distributed multiprocessor systems, the monitoring system must itself be physically distributed and capable of cooperation between various monitoring units in order to recognize events that may be defined across a number of nodes. The design of the monitoring system becomes that of a complete distributed system with real-time inputs; however, the specialization of the system allows its implementation in a very efficient way.

Passive hardware monitors observe activity on a parallel computer and record results for later analysis. They are often implemented as a set of programmable counters which count the occurrences of events on the bus, in the memory hierarchy, and on the processor. The performance monitors built into the Sequent Symmetry [67], and the PEM Monitor [13] are examples of this approach. Advantages of this passive approach include no perturbation of the observed system, and relatively straight forward hardware required. Since these passive monitors record information for all activity on a system, it is difficult to correlate performance information with its source. The hardware monitor on the Cray Y-MP [14] improves this situation by providing per process hardware collector information (by saving and restoring the state of the hardware collector as part of a context switch). However all of these systems have two limitations. First, not all interesting events in a program's execution are visible to the hardware monitor; tracing procedure calls and returns is particularly difficult. Second, since there are more event signals than counters, the user must select appropriate events to monitor.

An example of a passive hardware monitor is the RP3 [10] which includes hardware instrumentation built into almost every subsystem of the computer. Each device recognizes its own events and passes the information through I/O pins to a Performance Monitor Chip (PMC) which counts the events. The PMC also samples memory references in order to provide some statistics on memory usage. The capabilities are limited, however, due to constraints on space and cost imposed on the designs.

A more complex system to allow the non-intrusive monitoring of real-time distributed systems was proposed by Tsai *et al* [69]. The processors are mimicked by an additional, identical processor in the monitoring system so that the state of the processor can be known explicitly when monitoring begins. The state of the mirror processor can be saved in response to events occurring within the monitored processor. All interactions of the processors are assumed to be explicit (e.g., semaphores for shared data) and all significant events are recorded. The system also allows executions to be replayed.

Passive hardware monitors, however, have the disadvantage of having to infer process level state changes from processor level changes, typically processor bus signals. While it is possible for hardware to extract process level events from bus signals, it is unrealistic to expect cost effective hardware to do so. The cost for the monitoring system can easily equal the cost of the nodes of the target system. In addition, many popular high-performance architecture features work against hardware monitoring. Pipelines, register files, and on-chip memory management make many of the internal activities of the processor invisible from the external pins. A pipeline may fetch instructions never executed, variables of interest may be mapped to registers and only the physical addresses output by the memory management unit are visible on the external address lines of the CPU. Also, because standards for monitoring support for microprocessors are not likely to occur in the near future, hardware monitoring systems must contain target specific designs, at least at the lowest levels of physical contact with the processors busses.

An approach to overcome the cost and information visibility limitations of passive hardware monitors is to build a hybrid software and hardware monitor. Reilly did comprehensive work instrumenting the DEC M31 system [59] and compared pure software, hardware and hybrid monitoring. He noted that passive monitors required significantly more hardware than the hybrid approach. In addition, he reported that his hybrid system reduced the intrusion of the measured system by a factor of 10-20 times compared to software approaches. Based on this study, he concluded hybrid approaches were more practical than completely non-intrusive (passive) ones.

One approach to a hybrid monitor, used in the ZAHLMONITOR [28], is to have the program signal an event occurrence to the hardware via writes to special registers. The hardware counts each event occurrence, and either periodically sends the data to a file, or reports it to an analysis process.

DISDEB uses non-intrusive hardware to monitor busses of the shared memory Selenia Mara architecture (a system aimed at real-time process control applications) [39]. Compound event specifications are supported along with predicates for non-local events. The system includes monitoring boards with processors of equal power to the target nodes (8086 and 80286). The distributed hardware monitoring boards communicate via a signal bus. Event-action bindings can change dynamically, with interactive

commands. Event action predicates are necessarily limited by the limited hardware capabilities.

Another approach is to have the hardware collector generate trace data and send it to a data reduction station over a separate bus or interconnection network. MultiKron [54], TMP [24], and HYPERMON [48] are examples of this type of monitor. These are hybrid collectors because the user program initiates event recording by making a request to the hardware collector. This approach provides the flexibility and event visibility of direct instrumentation of an application with the low overhead of dedicated hardware collectors.

MultiKron is a custom VLSI co-processor for performance monitoring. It is designed to look like a memory mapped device to the main processor. To generate performance data, the main processor writes the event data to the co-processor which adds a processor id and time-stamp, and then generates the trace record. Traces are sent via a special purpose monitoring bus to a central data reduction station. The chip also provides several programable counters to record event frequency. MultiKron is designed as a general purpose performance monitoring co-processor and can be used with a variety of different processors.

The TMP system uses software probes with hardware support to minimize intrusion. Each Test and Measurement Processor (TMP) consists of a general purpose micro-processor and custom hardware to monitor the bus of the main processor. Application program's signal events to the TMP by generating references to a well known (and otherwise unused) rang of memory addresses. The system uses concepts of user and system events (called optional and standard). The micro-processor at each node filters events locally (to a degree) and then periodically sent to the central monitoring facility which handles the user interface. The system provides an option to view the state of programs in execution to aid in the debugging of long running programs.

HYPERMON is a software instrumentation system assisted with hardware on iPSC/2 [48]. The system is aimed at providing performance monitoring support. The system includes a hardware board that monitors backplane signals which can be manipulated with special I/O instructions in the processors. Data can be moved from the processes in 4 bit chunks that can prove to be a bottleneck. Unfortunately, the I/O operations are considerably slower than memory accesses.

The monitoring system proposed for PASM is a completely non-intrusive hardware-based system to support both debugging and performance evaluation [44]. The system includes physically distributed event recognition hardware with a central collection facility. It also includes a powerful event specification language based on the Event-Action Paradigm. The system allows intrusive actions to modify the state of the program for debugging and almost all event recognition is done in real-time.

Hardware instrumentation was included from the early stages in the design of Cedar in order to allow evaluation of the architecture and the programs subsequently written [46]. Information is gathered on the interconnection network, memory system, and signals from the backplane of the Alliant clusters. This information includes processor activity, cache requests, and address references. There is some support for "triggers" to support monitoring of software events. In order to use this feature, the processor writes to special memory ranges which enable/disable monitoring. The system supports profiling with actual execution time of code segments, processor utilization. It also permits tracing of some process and program level activities.

Lumpp and Casavant have also designed a system to exploit the similarities between the monitoring requirements of testing and debugging. They employ a non-intrusive monitor of the target CPU buses [45]. Event specifications are divided into primitive (or run-time) events and compound events. The primitive events are monitored during the execution while compound events are identified in a post-execution analysis phase. The complexity of the primitive events is driven by the complexity of the run-time monitoring hardware and the compound events are defined by temporal or logical predicates involving primitive and compound events. An intrusive host based prototype has been developed and is being evaluated [53].

All of these hardware monitors can reduce the intrusion of collecting data for parallel tools. However, a potential problem with hardware based approaches, is that they make it possible to cheaply collect so much data that is swamps the file system storing the data or workstation trying to process it. Hardware based instrumentation alone is not able to solve the data collection problem; tools must also be developed to assist with the intelligent collection of relevant data.

3.4 Event Specification

One of the key aspects of any monitoring system is how events are specified. The specification can have profound effect on the monitoring in both the types of information which can be gathered and the volume of data which must be handled. Monitoring cannot be separated from the specification of events. Many systems have included some type of specification language for events and almost all of them have included the ability to make temporal assertions on the events being monitored.

Snodgrass suggests storing monitoring information in a relational database called a "historical database" which contains a history of objects [64]. The user can also generate all queries before execution and selectively monitor only events useful for identifying the relational queries. The performance penalty and intrusion would be much higher if the historical database approach is taken because the number of primitive events to be recognized would greatly increase. Then complex events can be identified by queries on the database in TQuel and primitive events are gathered with software instrumentation in the users code or OS.

MPD applies path expressions and predecessor automata to event specification (regular expressions with history and concurrency constraints) [57]. The system uses the GDB debugger developed at CMU for Mach threads programs. Users set breakpoints corresponding to the terms of the path expressions defined previously. As breakpoints are reached the occurrence is recorded and used to evaluate the path predicates. The system assumes reproducibility of executions, which is odd for a debugger.

Many specification languages use temporal logic predicates to specify constraints (or assertions) on the monitored system. Goldszmidt proposed the division of debugging tools into two parts, a language dependent part and language independent part [22]. They implemented a prototype for Occam and Nil in which user code is automatically instrumented to communicate information to debugger process through standard communication channels. In a post-execution phase the traces were traversed to verify assertions made with a temporal specification language. The user could also make data-base like queries on the traces. In IDD (Interactive Distributed Debugger) the user specifies a set of temporal logic (interval logic) assertions which are checked during the execution [26]. The prototype system was aimed at a distributed workstation environment. The system included run-time identification of assertion violations which were flagged for the user. The ECSP debugger (a debugger for ECSP, a CSP like language) on the muTEAM system includes an event specification language based on assertions on the run-time behavior of the system [3]. The system attempts to compensate for intrusion by delaying for an equal amount of time any threads which synchronize with a thread delayed by instrumentation. They do not suggest how such a system could be implemented. However, it would rely on hardware for tightly coupled systems.

Another specification language is Peter Bates' Event Definition Language (EDL) [7]. EDL is a language permitting the user to create complex predicates for event classes composed of primitive events and logical and temporal operators. This provides the capability to abstract the activities of the system to higher level events and states specified by the user. The specification and identification of primitive events is left open, as are the problems of global time.

Several debugging systems have used EDL for event specification. Belvedere uses EDL to define events to be animated [33]. The system assumes most bugs in parallel programs are communication errors. The system uses postmortem animation of communications only; there is no support for events in sequential segments of code. Garcia and Berman also use EDL in a debugger which maps Path Pascal expressions to petri-net representations for debugging [20]. Path Pascal is based on monitor like objects for concurrency control. The nets are used to represent mutual exclusion scheme (objects in path pascal) on a multi-programmed single CPU machine. Monitoring threads compete for system resources with the monitored program and no effort is made to limit bandwidth loss due to debugging information.

3.5 Event Filtering

The instrumentation techniques described above make it possible to track a vast number of metrics and events during a application's execution. However, they are limited by the large amount of performance data they can generate. One approach to reduce the amount of trace data generated is to filter it so that only interesting events are recorded. EDL, Segall's *receptacles* [61], and Jade [34] use predicates and actions to recognize desired events (or sequences of events) and then generate synthetic events based on recognized events. ISSOS [60] and Meta [50] also use predicates to detect interesting states in a program, but rather than use an event stream abstraction, the user adds calls to the event recognizer into their

program. Filtering greatly reduces the amount of performance data collected. However, all of these tools force the user to select the desired events prior to starting their program.

Several tools incorporate dynamic control of the event recognition and filter process. EBBA [6, 5] is based on EDL, but permits dynamic insertion of the predicates. BEE [12] permits the dynamic control of the filtering of events based on both event type and by insertion of more complex predicates called "event interpreters". TOPSYS [8] and JEWEL [37] require that predicates be defined before the program starts execution, but permit dynamic control of which ones are enabled. Dynamic control of the information collected can greatly reduce the volume of performance data generated, but it creates that additional problem of selecting what data to collect. None of these tools directly address this problem.

3.6 Perturbation Compensation

In all of the instrumentation techniques described above (except purely passive) some perturbation of the application program will occur. While most techniques attempt to minimize the impact of these delays, a different approach is to compensate for this effect.

The concept of intrusion has been studied in several contexts and is also known as the "probe effect" [19]. Intrusion can be defined as any use of target system resources for monitoring activities [49]. For example, a print statement being used for debugging to log the contents of a variable constitutes intrusion through the use of CPU cycles and the I/O data paths. Intrusion includes both direct contention for resources (e.g., CPUs, memory, or communication links) and secondary interference with resources (e.g., interactions with cache replacement or virtual memory). Both of these effects must be considered for all resources in the system to quantify the intrusion introduced by the instrumentation.

Approaches to addressing the intrusion problem can be divided roughly into four categories. These are (listed in increasing order of difficulty to implement):

1. Realize that intrusion affects measurement and treat the resulting data as an approximation.
2. Leave the added instrumentation in the final implementation.
3. Try to minimize the intrusion.
4. Quantify the intrusion and compensate for it.

Clearly, approach 1 is not generally desirable. It leads to nondeterminism, eliminates the possibility of cyclic debugging and leaves the user wondering, "what really happened?" Although approach 2 effectively eliminates intrusion, it does so at the expense of the performance of the application. Approach 2 may indeed be feasible for applications which do not depend on performance; however, these problems are in the minority of problems typically targeted for high-performance systems. As a result, approach 3 is by far the most popular approach to date. Even though it does not assure the elimination of intrusion like approach 2, it does not degrade the performance of the final application. This approach is most popular in systems with a sufficient amount of nondeterminism already present in the computation. In this case, the goal is to assure the perturbation does not produce an execution flow which was not "likely". These techniques do not actually remove the intrusion but hide them in the inherent asynchrony and delays of the computation. Unfortunately, there is still the chance the intrusion changed the behavior of the application.

Some systems simply subtract the time required for the intrusion. An example is the PREFACE system [9]. In PREFACE, programs are assumed to be composed of parallel and serial portions, the bulk of the intrusion is done in the serial portions where the time perturbations can be subtracted without the possibility of changing any event orderings. In the parallel regions, the time necessary for execution of the instrumentation is also subtracted. However, no attempt is made to identify when intrusion actually introduces changes in the ordering of events and the approach is not useful in the general case.

Compensation for intrusion of software instrumentation was studied carefully for the PIE system [40]. The PIE system was designed for tightly coupled shared memory multi-processors, and was intended as a portable performance tuning environment. The software intrusion model includes both probes within processes and monitoring processes which can compete with the monitored processes. Intrusion is assumed to be limited to time delays incurred and is assumed to be constant and measurable. Again, no attempt was made to compensate if the ordering of events changed due to delays. However, they did differentiate between the "orders" of intrusion: order 1 is delays in threads, order 2 is a change of order of events, and order 3 is a change that alters the outcome of the computation.

Maloney and Reed studied the intrusion of software instrumentation for performance monitoring on the Intel iPSC/2, Cray X-MP, and Cray 2 [47]. They studied the timing delays introduced by intrusion and attempted to quantify changes in the ordering of events. They modeled intrusion as delays and constructively recover the compensated timings. Both *time-based* and *event-based* models were developed to compensate for the temporal changes to the application and the order changes possible through differential delays of processes. Because the goal was to approximate the performance of the instrumented applications, precise order information was not necessary. Their approach was to develop approximate models of instrumentation intrusion and then refine them until acceptable predictions of performance were possible.

Lumpp and Casavant, *et al* developed techniques for compensation of perturbations in message passing systems [43]. The approach includes requirements on the parallel system, the instrumentation, and the parallel program in order to assure recovery is possible. Their work differs from previous in that the correctness of the ordering of events is paramount. The concurrent program and the target architecture are viewed as a single complex system and modeled. Control theoretic techniques are employed to analyze the traces [71]. Given sufficient trace information, it is possible to recover the timings that would have occurred in the absence of the instrumentation. In addition, the technique determines when the instrumentation causes event order changes that can affect the subsequent behavior of the computation.

References

1. T. E. Anderson and E. D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, Boston, May 1990.
2. Z. Aral and I. Gertner. A high-level debugger/profiler architecture for shared-memory multiprocessors. In *Proceedings of the 1988 International Conference on Supercomputing.*, pages 131–139, New York, NY, 1988.
3. F. Baiardi, N. D. Francesco, and G. Vaglini. Development of a debugger for a concurrnet language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.
4. T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 19-22, 1992.
5. P. Bates. Distributed Debugging Tools for Heterogeneous Distributed Systems. In *Proceedings of the 8th Int'l Conf. on Distributed Computing Systems*, pages 308–315, San Jose, Calif., June 1988.
6. P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, Madison, WI, May 5-6, 1988. appears as SIGPLAN Notices, January 1989.
7. P. C. Bates and J. C. Wileden. EDL: A Basis For Distributed System Debugging Tools. In *Proceedings of 15th Hawaii International Conference on System Sciences*, pages 86–93, January 1982.
8. T. Bemmerl, A. Bode, P. Braum, O. Hansen, T. Tremi, and R. Wismuller. The Design and Implementation of TOPSYS. Technical Report TUM-INFO-07-71-440, Technische Universitat Munchen, July 1991.
9. D. Bernstein, A. Bolmarcich, and K. So. Performance visualization of parallel programs on a shared memory multiprocessor system. In *International Conference on Parallel Processing (ICPP)*, pages 1–10, University Park, PA, USA, August 1989.
10. W. C. Brantley, K. P. McAuliffe, and T. A. Ngo. RP3 Performance Monitoring Hardware. In R. K. Margaret Simmons and I. Bucker, editors, *Instrumentation for Future Parallel Computer Systems*, pages 35–47. Addison-Wesley, 1989.
11. O. Brewer, J. Dongarra, and D. Sorensen. Tools to aid in the analysis of memory access patterns for FORTRAN Programs. *Parallel Computing*, 9(1):25–35, 1988/89.
12. B. Bruegge. A Portable Platform for Distributed Event Environments. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 184–193, Santa Cruz, CA, May 20-21, 1991. appears as SIGPLAN Notices, December 1991.
13. H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Trans. Computers*, 38(5):725–737, May 1989.
14. Cray Research Inc. *UNICOS File Formats and Special Files Reference Manual*, SR-2014 5.0 edition.
15. M. E. Crovella and T. J. LeBlanc. Performance Debugging Using Parallel Performance Predicates. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, San Diego, CA, May 17-18, 1993.
16. R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. In *Proceedings of the SIGPLAN/SIGOPS*

Workshop on Parallel and Distributed Debugging, pages 163–173, Madison, WI, May 5-6, 1988. appears as SIGPLAN Notices, January 1989.

17. J. M. Francioni, L. Albright, and J. A. Jackson. Debugging Parallel Programs Using Sound. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 68–75, Santa Cruz, CA, May 20-21, 1991. appears as SIGPLAN Notices, December 1991.
18. M. Friedell, M. LaPolla, S. Kochhar, S. Sistare, and J. Juda. Visualizing the Behavior of Massinely Parallel Programs. In *Proceedings of Supercomputing'91*, pages 472–480, Albuquerque, NM, November 18-22, 1991.
19. J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.
20. M. E. Garcia and W. J. Berman. An Approach to Concurrent Systems Debugging. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 507–514, May 1985.
21. A. J. Goldberg and J. L. Hennessy. Performance Debugging Shared Memory Multiprocessor Programs with MTOOL. In *Proceedings of Supercomputing'91* , pages 481–490, Albuquerque, NM, November 18-22, 1991.
22. G. S. Goldszmidt, S. Katz, and S. Yemini. Interactive Blackbox Debugging for Concurrent Languages. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging published in ACM SIGPLAN Notices*, 24(1):271–282, January 1989.
23. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, June 1982.
24. D. Haban and D. Wybranietz. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, Feb 1990.
25. G. J. Hansen, C. A. Linthicum, and G. Brooks. Experience with a Performance Analyzer for Multithreaded Application. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 124–131, Amsterdam, June 11-15, 1990.
26. P. K. Harter, Jr., D. M. Heimbigner, and R. King. IDD: An Interactive Distributed Debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498–506, May 1985.
27. M. T. Heath and J. A. Etheridge. Visualizing Performance of Parallel Programs. *IEEE Software*, 8(5), September 1991.
28. U. Hecksen, R. Klar, W. Kleinoder, and F. Kneibl. Measuring Simultaneous Events in a Multiprocessor System. In *Proceedings of 1982 SIGMETRICS Conference*, pages 77–88, August 1982.
29. J. K. Hollingsworth, R. B. Irvin, and B. P. Miller. The Integration of Application and System Based Metrics in A Parallel Program Performance Tool. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming* , pages 189–200, Williamsburg, VA, April 21-24 1991. appears as SIGPLAN Notices, July 1991.
30. J. K. Hollingsworth and B. P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, July 1993.
31. J. K. Hollingsworth and B. P. Miller. Parallel Program Performance Metrics: A Comparison and Validation. In *Proceedings of Supercomputing 1992*, pages 4–13, Minneapolis, MN, November 1992.
32. A. A. Hough and J. E. Cuny. Perspective Views: A Technique for Enhancing Parallel Program Visualization. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages II.124–132, August 1990.
33. A. A. Hough and J. E. Cuny. Initial Experiences with a Pattern-Oriented Parallel Debugger. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 195–205, Madison, WI, May 5-6, 1988. appears as SIGPLAN Notices, January 1989.
34. J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.
35. K. Kinoshita. An Experience with the ANALYZER/SX Performance Tuning Tool. In R. K. Margaret Simmons and I. Bucker, editors, *Instrumentation for Future Parallel Computer Systems*, pages 223–231. Addison-Wesley, 1989.
36. J. Kohn and W. Wiliams. ATExpert. *Journal of Parallel and Distributed Computing*, 18:205–222, 1993.
37. F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurement System. *to appear in IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–71, November 1992.
38. J. R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *SPE*, 20(12):1241–1258, Dec 1990.
39. B. Lazzerini and C. A. Prete. A Programmable Debugging Aid for Real-Time Software Development. *IEEE Micro*, 6(3):34–42, June 1986.
40. T. Lehr. *Compensating for Perturbation by Software Performance Monitors in Asynchronous Computations*. PhD thesis, Carnegie Mellon University, Dept. of Electrical and Computer Engineering, 1990.
41. T. Lehr, D. Black, Z. Segall, and D. Vrsalovic. MKM: Mach Kernel Monitor Description, Examples and Measurements. Technical Report CMU-CS-89-131, Carnegie-Mellon University, March 1989.
42. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman. Visualizing Performance Debugging. *IEEE Computer*, 21(10):38–51, October 1989.

43. J. E. Lumpp, Jr. *Models for Recovery from Software Instrumentation Intrusion*. PhD thesis, University of Iowa, Dept. of Electrical and Computer Engineering, 1993.
44. J. E. Lumpp, Jr., T. L. Casavant, H. J. Siegel, and D. C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, June 1990.
45. J. E. Lumpp, Jr., R. K. Shultz, and T. L. Casavant. Design of a System for Software Testing and Debugging for Multiprocessor Avionics Systems. In *Proceedings of the 15th Annual International Computer Software and Applications Conference (COMPSAC91)*, pages 261–268, September 1991.
46. A. D. Malony. Multiprocessor instrumentation: Approaches for Cedar. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 1–34, Reading, MA, 1989. Addison Wesley.
47. A. D. Malony. *Performance Observability*. PhD thesis, University of Illinois, 1990.
48. A. D. Malony and D. A. Reed. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 213–226, Amsterdam, June 11-15, 1990.
49. D. C. Marinescu, J. E. Lumpp, Jr., T. L. Casavant, and H. J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, June 1990.
50. K. Marzullo and M. D. Wood. Tools for Constructing Distribued Reactive Systems. Technical Report 91-1187, Cornell University, January 1991.
51. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.
52. B. P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley UNIX. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 43–54, May 1985.
53. S. P. Miller, J. E. Lumpp, Jr., and G. B. Doak. Modified Decision/Condition Testing. Technical Report Technical Report #WP91-2005, Rockwell International Corporation, 1991.
54. A. Mink, R. Carpenter, G. Nacht, and J. Roberts. Multiprocessor Performance Measurement Instrumentation. *IEEE Computer*, 23(9):63–75, September 1990.
55. C. M. Pancake and S. Utter. Models for visualization in parallel debuggers. In *Supercomputing 1989*, pages 627–636, Reno, NV, November 1989.
56. S. E. Perl and W. E. Weihl. Performance Assertion Checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 134–145, Asheville, NC, December 5-8, 1993.
57. M. K. Ponamgi, W. Hseush, and G. E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, 8(3):37–43, May 1991.
58. D. A. Reed, R. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. The Pablo Performance Analysis Environment. Technical report, Dept. of Comp. Sci., University of Illinois, 1992.
59. M. Reilly. Instrumentation for Application Performance Tuning: The M31 System. In R. K. Margaret Simmons and I. Bucker, editors, *Instrumentation for Future Parallel Computer Systems*, pages 143–158. Addison-Wesley, 1989.
60. K. Schwan, R. Ramnath, S. Vasudevan, and D. M. Ogle. A language and system for parallel programming. *IEEE Transactions on Software Engineering*, pages 455–471, April 1988.
61. Z. Segall, A. Singh, R. T. Snodgrass, A. K. Jones, and D. P. Siewiorek. An Integrated Instrumentation Environment for Multiprocessors. *IEEE Transactions on Computers*, C-32:4–14, January 1983.
62. E. T. Smith. *Debugging Techniques for Communicating, Loosely-Coupled Processes*. PhD thesis, University of Rochester, December 1981.
63. S. Smith and M. G. Williams. The Use of Sound in an Exploratory Visualization Environment. Technical Report R-89-002, University of Lowell Computer Science Department, 1989.
64. R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.
65. D. Socha, M. L. Baily, and D. Notkin. Voyeur: Graphical Views of Parallel Programs. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 206–215, Madison, WI, May 5-6, 1988. appears as SIGPLAN Notices, January 1989.
66. D. H. Sonnenwald, B. Gopinaht, G. O. Haberman, W. M. K. III, and J. S. Myers. InfoSound: An Audio Aid to Program Comprehension. In *Proceedings of the Twenty-Third Hawaii International Conferences on System Sciences*, volume 11, pages 541–546, 1990.
67. S. S. Thakkar. Performance of Parallel Applications on a Shared-Memory Multiprocessor System. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, pages 233–256. Addison-Wesley, 1990.

68. R. Title. Connection Machine Debugging and Performance Analysis: Present and Future. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 272–275, Santa Cruz, CA, May 20-21, 1991.
69. J. J. P. Tsai, K.-Y. Fang, and H.-Y. Chen. A Noninvasive Architecture to Monitor Real-Time Distributed Systems. *IEEE Computer*, 23(3):11–23, March 1990.
70. S. Utter-Honig and C. M. Pancake. Graphical Animation of Parallel Fortran Programs. In *Proceedings of Supercomputing'91*, pages 491–500, Albuquerque, NM, November 18-22, 1991.
71. K. J. Williams, J. A. Gannon, M. S. Andersland, J. E. Lumpp, Jr., and T. L. Casavant. Conditions for tracking timing perturbations in timed Petri nets with monitors. In S. Balemi, P. Kozak, and R. Smedinga, editors, *Discrete Event Systems: Modeling and Control*, pages 141–151, Basel, Switzerland, 1993. Berkhauser-Verlag.
72. C.-Q. Yang and B. P. Miller. Performance measurement of parallel and distributed programs: A structured and automatic approach. *IEEE Trans. Software Eng.*, 12:1615–1629, Dec. 1989.
73. D. Zernik and L. Rudolph. Animating Work and Time for Debugging Parallel Programs Foundation and Experience. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 46–56, Santa Cruz, CA, May 20-21, 1991. appears as SIGPLAN Notices, December 1991.