FINDING BOTTLENECKS
IN LARGE SCALE PARALLEL PROGRAMS

by

JEFFREY KENNETH HOLLINGSWORTH

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1994

ACKNOWLEDGEMENTS

Pursuing a PhD in computer sciences is a long and sometimes bumpy journey; however, the people you meet along the way make the process bearable and often pleasurable. I thank my advisor, Bart Miller, for his years of advice, encouragement, and mentoring. I thank the members of my thesis committee. To the readers, Jim Larus and Miron Livny, I am grateful for their many helpful suggestions that have improved the content and clarity of this thesis. I thank David Wood and Ernest Hanson of the Business School for their probing questions at my defense and the resulting improvements to this thesis.

I thank the staff of the Paradyn Performance Tools project: Mark Callaghan, Bruce Irvin, Karen Karavanic, Krishna Kunchithapadam, and Tia Newhall. A special thanks goes to Jon Cargille for listening to my ranting and spending more time working with low level CM-5 software than anyone should have to endure. I am also grateful to the people who provided application programs for measurement: Mark Hill, Spyros Kontogiorgis, Gary Lewandowski, Tia Newhall, Joann Ordille, Gary Schultz, and Madhusudhan Talluri.

To my parents, John and Sandra, thank you for nurturing and encouraging me. I thank my friends for their support and comradery especially: Ted Faber, Scott Lewandowski, Karen Lohman, and Rich Maclin. Finally, I thank Dane County Habitat for Humanity for providing a constructive outlet for my need to tinker.

This research supported in part by Department of Energy grant DE-FG02-93-ER25176, Office of Naval Research grant N00014-89-J-1222, National Science Foundation grants CCR-9100968 and CDA-9024618, an ARPA Graduate Fellowship in High Performance Computing, and a grant from Sequent Computer Systems Inc.

ABSTRACT

This thesis addresses the problem of trying to locate the source of performance bottlenecks in large-scale parallel and distributed applications. Performance monitoring creates a dilemma: identifying a bottleneck necessitates collecting detailed information, yet collecting all this data can introduce serious data collection bottlenecks. At the same time, users are being inundated with volumes of complex graphs and tables that require a performance expert to interpret. I have developed a new approach that addresses both these problems by combining dynamic on-the-fly selection of what performance data to collect with decision support to assist users with the selection and presentation of performance data. The approach is called the W^3 Search Model. To make it possible to implement the W^3 Search Model, I have developed a new monitoring technique for parallel programs called *Dynamic Instrumentation*. The premise of my work is that not only is it possible to do on-line performance debugging, but for large scale parallelism it is mandatory.

The W^3 Search Model closes the loop between data collection and analysis. Searching for a performance problem is an iterative process of refining the answers to three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the *why* question, tests are conducted to identify the type of bottleneck (e.g., synchronization, I/O, computation). Answering the *where* question isolates a performance bottleneck to a specific resource used by the program (e.g., a disk system, a synchronization variable, or a procedure). Answering *when* a problem occurs, tries to isolate a bottleneck to a specific phase of the program's execution.

Dynamic Instrumentation differs from traditional data collection because it defers selecting what data to collect until the program is running. This permits insertion and alteration of the instrumentation during program execution. It also features a new type of data collection that combines the low data volume of sampling with the accuracy of tracing. Instrumentation to precisely count and time events is inserted by dynamically modifying the binary program. These counters and timers are then periodically sampled to provide intermediate values to the W^3 Search Model. Based on this intermediate data, changes are made in the instrumentation to collect more information to further isolate the bottleneck.

I have built a prototype implementation of the W^3 Search Model and of Dynamic Instrumentation. The prototype runs on Thinking Machine's CM-5 and a network of workstations running PVM. In one study, the tools identified the bottlenecks in several real programs using two to three orders of magnitude less data than traditional techniques. In another study, Dynamic Instrumentation was used to monitor long running programs and introduced less than 10% perturbation. While the W^3 Search Model and Dynamic Instrumentation complement each other, they are also useful individually. The W^3 Search Model can be applied to existing post-mortem performance tools or even to simulated machines and environments. Dynamic Instrumentation has be used to collect performance data for other uses including visualization. The W^3 Search Model and Dynamic Instrumentation are being incorporated into the Paradyn Performance Tools.

Chapter 1

INTRODUCTION

1.1. Motivation

Parallel computers are used because often they are the only way to provide the required computational power. Since performance is important, tools that help users get better performance from their programs are essential to parallel computing. As the number and computational power of processors in parallel computers increases, the volume and complexity of performance data gathered from these machines explodes. First, this wealth of information is a problem for the programmer who is forced to navigate through it, and for the tools that must store and process it. What makes this situation worse is that most of the time, most of the data is irrelevant to understanding the performance of the application. A second problem with the volume of the performance data available is the cost of collecting it. To minimize perturbation, a performance tool should collect only the data necessary to explain the performance of the program. However, deciding which data is relevant during a program's execution is difficult.

Information overload is a serious problem facing parallel programmers. Existing performance tools incorporate several different techniques (e.g., visualization, auralization, and metrics) to address this problem. Visualization and auralization provide a way for programmers to perceive the way their program is performing via graphics and sound. Metrics use measurements to quantify the location of a bottleneck by computing numeric values for different parts (e.g. procedures, processors) of the program being measured. By using these techniques, it is possible to provide programmers with insight into the performance of their program with a relatively small amount of data.

Each of these techniques assumes the programmer knows what picture to draw or which metric is appropriate. Unfortunately, no single approach applies to all parallel programs. In one study we conducted [40], four performance metrics were compared head-to-head for a number of real parallel programs. In addition, we employed a technique to calibrate the accuracy of the guidance supplied by the metrics. That study showed that for different programs, different metrics provided the best guidance. Since no metric or visualization provides optimal guidance, tools tend to include several different types of output. As a result, the user is forced to sort through a sea of potentially conflicting pictures and tables trying to find one that will explain the performance of their program. An excess of pictures is only marginally better than having to search through a sea of raw numbers. To address the information overload problem, we provide *decision support* that helps users to find the right data to understand their program's performance and to select appropriate ways to display and analyze it. Our goal is to give programmers specific advice about where the performance bottlenecks lie in a program.

Before it is possible to provide a meaningful methodology for decision support and advice, it is vital to understand the process of performance debugging parallel programs. Based on our informal studies of how programmers used a previous performance tool, IPS-2[65], several trends can be inferred. Programmers generally start by looking at a high level view of the performance of their application, and iteratively isolate the source of their program's poor performance. This process continues until they understand their program's performance well enough to start tuning it. This thesis develops a performance environment that mimics this manual refinement process and helps to automate what can be a tedious task for the programmer.

Of equal importance to the information overload problem is the problem of how to collect the required performance information efficiently. The potential volume of performance data generated by even moderate size parallel machines is a significant problem for tool designers. We estimate that current RISC processors can generate about two megabytes per second of performance data when collecting basic performance information at the granularity of procedure and system calls. This estimate is based on generating traces of these events. For a large scale parallel computer, say 1000 nodes, this amount of data would be impractical to collect for all but the shortest programs.

An alternative to tracing is to use counters and timers to summarize performance data. Unfortunately when different combinations of procedures, processes, and time varying information are considered, this approach produces as much data as full tracing. Another option to manage data volume is to use short executions of parallel programs to try to understand their performance. However, since the nature of bottlenecks changes when programs are scaled up, parallel programs need to be run at production size to understand their performance. The obvious solution, since most data is irrelevant, is to collect less data. However, this creates a dilemma. Until we isolate the bottleneck, we do not know what data we need to collect. But to find the bottleneck, we need to collect the data!

1.2. Summary of Results

This dissertation provides a two pronged solution to the problems of information overload and data collection. First, we provide a structured methodology called the *W^3 Search Model* to automate the search for performance bottlenecks. Second, a new model of data collection, called *Dynamic Instrumentation*, is introduced. Dynamic Instrumentation permits instrumentation decisions to be made and changed during execution. The two prongs of this work combine to form a integrated system that provides parallel programmers with an efficient system to answer their performance questions.

The W^3 Search Model tries to answer three questions: why is my application performing poorly, where is the bottleneck, and when does it occur? To answer the ''why'' question, we have developed a library of potential bottlenecks, called *hypotheses*, that cover virtually all types of problems in a parallel application. Answering the ''where'' question yields a detailed description of the particular processor, synchronization object, procedure, or disk that is limiting the performance of the application. To answer the ''when'' question, we take advantage of the fact that most parallel programs go through several distinct phases of execution and that different bottlenecks affect different phases. The reason that the W^3 Search Model can express most bottlenecks with a fixed set of hypotheses is that each type of bottleneck can be parameterized by the components of the 'where' and 'when' axes. This gives programmers the precise information they need to understand and tune their programs. The W^3 Search Model is an application and machine independent framework for the isolation of performance bottlenecks. This framework can be used on a variety of parallel machines and makes it is easy for programmers to tune their programs on different platforms. Also, platform independence means that our ideas apply equally well to a heterogeneous combination of machines or to a single homogeneous machine.

The second major contribution of this thesis is Dynamic Instrumentation. Dynamic Instrumentation provides demand-driven performance monitoring. The key idea is to collect data only when we need it, and thus greatly reduce the volume of data collected. By only collecting data when necessary, we can provide a richer palette of performance data than other data collection approaches. In addition, we can monitor larger and longer applications than previously possible. Dynamic Instrumentation also makes it possible to analyze the data being collected during an execution, and change what data is being collected. This collapses the long instrumentation, compilation, execution, and analysis cycles into a single execution of the program. This is a significant advantage for monitoring long running programs such as database servers and multi-hour scientific applications.

An integral part of the dynamic control of program instrumentation is a data collection *cost model* to regulate the perturbation caused by the instrumentation. The model associates a cost with the data being collected and its analysis. Collecting performance data affects many different resources: processors, interconnection networks, disks, and data analysis workstations. Our cost model incorporates the impact of data collection on each of these resources. The cost model provides feedback to the W^3 Search Model which uses it to control how many potential bottlenecks to consider at once. Cost information can also be displayed to the user to show how the performance environment perturbed the application program.

This thesis also calls for a significant shift in the way in which parallel program performance debugging tools are designed, built, and used. To make it possible to dynamically select performance data, performance debugging must move from a post-mortem activity to an iterative, interactive execution time activity. This shift also affords secondary benefits such as the option to integrate performance debugging with correctness debugging and application steering.

The two prongs of this work, the W^3 Search Model and Dynamic Instrumentation, separately provide solid solutions to the problems of information overload and data collection. However, bringing the two ideas together in a single system creates significant synergistic affects. The W^3 Search Model provides a natural way to drive Dynamic Instrumentation and to relieve the programmer from the tedious task of selecting what data to collect. Dynamic Instrumentation provides an efficient mechanism to collect data for the rich palette of different performance problems that can be described using the W^3 Search Model.

We have built an initial implementation of Dynamic Instrumentation and the W^3 Search Model. Currently, the implementation runs on a Thinking Machines CM-5, and a network of workstations running PVM. The system has been used to study several real applications with execution times ranging from a few minutes to over one hour. In each case, significant bottlenecks were identified by the tool.

1.3. Organization of thesis

The remainder of this thesis is divided into eight chapters. Chapter 2 discuses previous work in the field. The middle of the thesis describes the W^3 Search Model, Dynamic instrumentation, and the cost model. After introducing each idea, we present a short example or study to illustrate the effectiveness of the concept. The final part of the thesis brings together the separate ideas and describes a prototype implementation of them. It includes both a commentary of how the components interact and a study of several applications using the prototype implementation.

Chapter 3 defines the W^3 Search Model. We describe the process of independently refining the ‘‘why’’, ‘‘where’’, and ‘‘when’’ of a program’s performance as a search in a three dimensional space where each axis is a dimension. We explain how hypotheses can be evaluated for different combinations of resources along the ‘‘where’’ axis and phases along the ‘‘when’’ axis. This chapter also develops a process to automate searching for performance bottlenecks.

In Chapter 4, a proof of concept implementation of the W^3 Search Model is described. In addition, we present a small case study to demonstrate the effectiveness of the model at finding real bottlenecks.

Chapter 5 describes the data collection approach and Dynamic Instrumentation. We introduce a new way to collect performance data that is both efficient and permits a potentially rich variety of different metrics to be described. We also present a technique to permit performance instrumentation to be inserted into a running program.

Chapter 6 reports on an initial implementation of Dynamic Instrumentation. We also include a series of measurements both at the micro (primitive) level and macro (application) level to quantify the efficiency of Dynamic Instrumentation.

Chapter 7 describes the data collection and analysis cost model. It also shows how this model can be used to throttle the interactions between the W^3 Search Model and Dynamic Instrumentation.

Chapter 8 combines the W^3 Search Model and Dynamic Instrumentation and includes performance case studies for several real parallel programs.

Finally, Chapter 9 summarizes the contributions of this thesis and outlines future extensions to the work.

Chapter 2

RELATED WORK

Performance debugging tools consist of three parts: data collection, analysis, and presentation. Data collection involves deciding what data to collect and how to collect it. Collected data can be restricted to the application being measured or it might encompass statistics from other sources such as the operating system. Measurements may be made by software inside the application itself or by dedicated hardware. Performance data may be gathered either as statistical samples or logs of interesting events. Data analysis reduces a large quantity of raw data to a small amount that explains an application's performance. Analysis can be accomplished by filtering the data or transforming it to numerical metrics. Data presentation includes not only deciding the visual representation of the data, but also managing its presentation to prevent the user from being overwhelmed with too much information. In the past, techniques to avoid information overload have focused on providing effective ways to visualize performance data. This thesis concentrates on two aspects of performance debugging: the problems of data collection and information overload. In this chapter we summarize previous performance debugging environments with an eye to how they handle these two issues. We also note the strengths and weaknesses of the different approaches.

2.1. Information Overload

Previous work to manage the volume of performance data concentrated in two areas: performance metrics and performance visualization. Performance metrics reduce large amounts of raw data into a few numbers that characterize bottlenecks in a parallel application. Performance visualization presents the user with a visual (or aural) abstraction of the parallel computation to help identify the bottleneck in an application.

2.1.1. Performance Metrics

Performance metrics are a tool to help programmers reduce the running time of their programs. Profile metrics are performance metrics that associate a value with each component of a parallel application (frequently procedures). Metrics often are presented as sorted tables so that the most important components are at the top of the display. Profile metrics originated in sequential programming as a simple list of the CPU time consumed by each procedure in an application. The standard UNIX CPU time profiler, Prof, is an example of such a tool. A logical extension of this technique to parallel applications is to total the CPU time profiles from each process (or thread). The profiling environment on the Connection Machine[93] provides this type of metric.

Adding up CPU time from each process ignores the fact that not all CPU time is of equal importance to a parallel program's performance. For example, time spent in a sequential routine has a greater impact on the execution time of a program than the same total time split among three simultaneous instances of a parallel procedure. Anderson and Lazowska developed the Quartz Normalized Process Time (NPT) metric[1] to compensate for the inequities of treating all CPU time the same. NPT normalizes CPU time for each procedure based on the effective parallelism while that procedure is executing. Normalization assigns relatively higher values (indicating they are more important) to sequential procedures than to parallel ones. NPT provides more accurate information than prof, but requires all threads of execution to frequently update a single shared data structure.

Simply using sequential metrics for parallel applications is not sufficient because in a parallel application improving the procedure that consumes the largest amount of time will not necessarily improve the execution time. Inter-process dependencies in a parallel application influence which procedures are important to an application's execution time. Yang and Miller developed Critical Path Analysis[102, 103] to provide more accurate guidance. Critical path provides an estimate of the improvement in the runtime possible if a single procedure is optimized. This estimate is an upper bound, and an extension to Critical Path called Logical Zeroing[65] provides a better estimate. These metrics provide detailed information about how to improve a parallel application, but building the data structures and calculating the metrics require significant space and time.

Prof, NPT, and Critical Path Analysis 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. MTOOL[33] is a metric based tool that includes information about the memory hierarchy. MTOOL compares the observed execution time for an application to its predicted execution time to characterize the influence of the memory hierarchy on the computation. Differences between predicted and observed execution times are attributed to memory contention and extra time to access non-local memory. This is a useful metric, but it is only applicable to memory bottlenecks.

Rather than trying to find a single metric to characterize the entire application, many tools provide sorted profiles for several different resources. Some commonly profiled resources are: CPU utilization, synchronization time, disk operations, vectorization, and cache performance. INCAS[35], JEWEL[51], and ANALYZER/SX[48] are examples of this approach. These tools provide a wealth of information to the programmer, but force them to select the appropriate resources to profile their application.

Building a tool that includes all potentially useful metrics is difficult. An alternative is provide a library of metrics, and make the tool extensible to permit users to create their own metrics. One such tool is IPS-2[39] 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 fraction 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. Defining new metrics as expressions of intrinsic metrics provides an easy way to extend a performance tool. However, the type 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[27] provides a toolkit for building new metrics and analysis modules built on top of a basic event stream. Extensible systems provide a powerful tool for sophisticated users, 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. In a previous paper[40], we compared several different metrics (including Critical Path, Logical Zeroing, and NPT) and concluded that there was no single best metric. However, we were able to characterize the types of applications where each metric would be useful. This information is valuable to the programmer and could help them select appropriate metrics. Different metrics have different costs of computation. Sometimes a cheaper metric (e.g., prof) is adequate, and so there is no need to calculate complex metrics. Unfortunately, no current tools incorporate these meta-metrics.

2.1.2. 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 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 (describe) and to give advice on how to correct it (prescribe).

ATExpert[49] from Cray Research uses rules to recognize performance problems in Fortran programs. Information about the achieved speedup, overhead of parallel routines, and sequential time are used to select appropriate suggestions about how the program's execution time might be improved. Data can been presented at the granularity of subroutines, loops, parallel blocks, or case statements. Recently, Cray Research has released a new tool, the MPP Apprentice[98], that recognizes performance problems for message passing programs on the Cray T3D[19]. The MPP Apprentice's approach is similar to ATExpert, but it works for a more general programming model.

Crovella and LeBlanc's predicate profiling[20] provides a search system to compare different algorithms for the same problem and allow them to evaluate the scalability of a particular algorithm. They define a set of rules that test for possible losses in performance of a parallel program. Losses are classified into four categories: load imbalance, starvation, synchronization, or 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 due to useful work and various sources of loss). Information is presented at the granularity of the entire application; this provides high level information about the type of bottleneck. However, they do not include any way to isolate the bottleneck to program components such as procedures or processes.

Another 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[73] uses this approach. It is a post-mortem tool that checks for assertion failures by scanning log files created during program execution. 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.

Another tool that employs user defined predicates is PMPL[26]. PMPL can be used with existing trace monitors such as AIMS[101] and PICL[32]. PMPL uses user defined predicates to control performance data logging in large scale parallel computers. When a performance predicate is detected, event logs from a circular buffer are written to disk for latter analysis.

The major limitation of these search-based tools is that they do not fully address the data collection problem. Each tool either requires the user to manually specify predicates, or provides only a limited granularity of data collected (i.e., for the entire application).

2.1.3. Visualization

A different approach to finding performance problems in parallel programs is to provide pictures to help programmers visualize problems. 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 visualization of low-level activity on a parallel machine, algorithm animation, and auralization. We divide the existing performance visualization tools into three categories: single visualizations for one type of bottleneck, a fixed collection of visualizations, and toolkits for building user defined visualizations.

A basic visualization is a representation of the physical machine. For example, MAP[13] presents memory access patterns for sequential FORTRAN programs running on vector supercomputers. MAP shows memory as a two dimensional grid. Whenever a memory location is referenced its location in the grid is highlighted, and then 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. This visualization is simple, but effective for identifying memory hot spots. A limitation of MAP is that it does not relate memory accesses to the source code.

Another tool that provides a single visualization of the resources in a parallel computation is PIE[55, 82]. PIE provides a color coded time-line tracking CPU state (e.g., useful work, blocked, spinning) through time. The tool can also show which procedures are executing by assigning a different color to each procedure in the program. 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 are difficult to interpret. For example, at the process level of abstraction it is difficult to see results, such as effective parallelism, that depend on multiple processes‡. At the procedure level, there are so many colors that it is impossible to tell what is happening.

‡ This limitation is so acute that the authors added a hand drawn plot of parallelism vs. time to their paper to explain one of their visualizations.

Another common visualization is an illustration of the synchronization patterns in a parallel computation. For example, the MAD debugger[104] provides a "causality graph", showing each thread as a vertical line, and the inter-thread communication and synchronization as diagonal lines. Although this provides great detail, the programmer must scan a large and 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. Providing a single visualization limits the types of bottlenecks that can be discovered. Several tools have been developed that incorporate multiple visualizations. An example of this type of tool is Paragraph[37], which supports over twenty different types of displays. Many of the displays can be configured to plot values for different resources (e.g., CPU and disk utilization), greatly increasing the number of available performance displays. Unfortunately not every visualization is useful for every program (some are only useful for a few programs), and the user is left with the formidable task of selecting appropriate visualizations and resources to display. Moviola[54] and the performance tool for PREFACE[10] also provide large libraries of pre-defined visualizations.

No matter how many visualizations a tool provides, users will eventually want more. A solution to this problem is being explored in two systems, Pablo[75] and JEWEL[51]. Both of these systems contain toolkits for building visualization modules. The user constructs visualizations from building blocks. This method permits an almost unlimited number of visualization modules to be built. Each tool also includes a collection of pre-defined visualizations so that the programmer can start using the tool without having to create their own visualizations. However, a major limitation of these systems is that they provide no assistance to the user in creating appropriate visualizations to find their performance bottleneck.

The visualizations discussed so far have been designed to work for any program being studied. A different approach is algorithm animation. Algorithm animation tries to graphically represent an application's execution at a level of abstraction similar to the programmer's mental model of the program. According to Pancake and Utter[71], 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 understand a program's performance. The difficult part is relating events on the physical machine back to an appropriate abstract model. Two approaches have been tried to express these relationships. IVE[29], Voyeur[88], POLKA[90], and Zeus[14] require the programmer to make calls to animation routines from their program. A second approach, used in Belvedere[42, 43], provides an event stream and the user defines reduction and animation operations over it. A problem with algorithm animation is that it adds the burden of creating and debugging animations to the process of writing a parallel application.

A variation on algorithm animation is auralization[28, 87, 89]. Instead of representing a program graphically, auralization animates it using sound, or a combination of sound and pictures. This technique provides an extra dimension to represent the application, but is still limited because it requires the programmer to select appropriate operations for animation.

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 many applications.

2.2. Data Collection

Before performance data can be analyzed or presented, it must first be collected. Where the data gets collected, what data gets collected, and how it gets collected varies greatly with different systems. Performance data can be measured for applications, operating systems, or the hardware itself. The data gathered ranges from summary counters to detailed event traces. The instrumentation to measure programs can be either software, custom monitoring hardware, or a hybrid combination. No matter the approach, effective data collection requires gathering detailed information from a parallel program yet minimizing perturbation so that the data is representative of the un-instrumented program. Many approaches have been tried to develop efficient data collection. In this section, we review a number of different data collection techniques.

2.2.1. Program Instrumentation

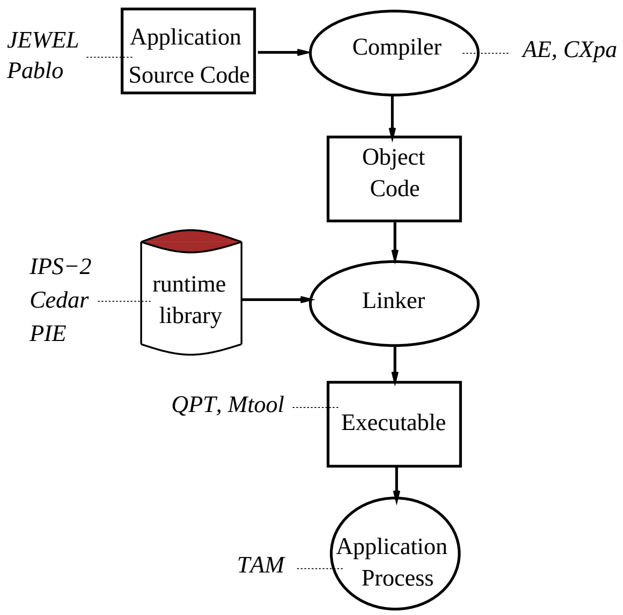

Figure 2.1. Inserting Instrumentation into a Program.

Performance instrumentation can be inserted into an application program at any stage during compilation. Dashed lines indicate how different tools (shown in italic) insert their 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 manually by editing the source code, automatically by the compiler, by linking instrumented libraries, or by modifying the linked executable. Each of these approaches has advantages and disadvantages. Figure 2.1 shows the stages of a compilation, and where 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[65] and Cedar tracing[58] use this approach. Runtime library instrumentation does not require the user to modify their application; it just requires re-linking the program with a modified version of the library. A similar approach is to provide a set of instrumented primitives and require the user to write their applications using these primitives. For example, Instant Replay[53] provides primitives to operate on events, queues, shared memory objects, and processes on the BBN Butterfly[3]. Likewise, PIE[82] works with a programming model called MPC[94] which provides parallelism via operations on shared memory. Basing instrumentation on a particular set of primitives is appropriate when the performance monitoring system is being developed in conjunction with the programming model.

To collect performance information about individual program constructs, such as loops or statements, instrumenting runtime libraries or primitives is not sufficient. Collecting data at this granularity requires instrumentation be interspersed with the statements of the user's program. CXpa[36], AE[52], Prism[84], and MPP Apprentice[98] use a modified compiler to insert instrumentation at the desired location. Compiler based instrumentation affords access to the wealth of information that is available during compilation. For example, information about data and loop dependencies is difficult to gather without compiler information. However, to use compiler based instrumentation, the program must be re-compiled. This forces decisions about what data to collect to be made early in the edit, compile, run cycle. For a large program, re-compiling the source code to gather performance information is a significant barrier to using the tool. In addition, to gather information about libraries their source must be available, but for many commercial products source code is not readily available.

Alternatively the instrumentation can be inserted after the executable image has been built. This approach, called binary re-writing, inserts instrumentation into an object file after it has been compiled and assembled. QPT[4], Mtool[33], jprof[77], and the Stardent Postloader[45] all use this style of instrumentation. Binary re-writing has the advantage that it is language independent, and that compilers and libraries do not need to be modified.

It is also possible to hand modify an application to insert calls to collect performance data. JEWEL[51] uses this approach. Hand instrumentation provides the user a great degree of selectivity, but it 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 and eliminates the need to modify the compiler. The instrumentation system is also portable across platforms, but is tied to a single language. Also, the data dependencies are not available, and of course it requires that the program be re-compiled to collect the data. Pablo[75] and AIMS[101] use source to source translation.

The techniques described so far all require the instrumentation to be inserted prior to program execution. One system that defers instrumentation until the program has started to execute is the TAM facility[78] provided by Intel for the Paragon. TAM uses a fixed set of performance instrumentation profiles (i.e., prof style sampling, or full event tracing) to insert instrumentation into a program after it has been loaded into memory but prior to execution. TAM provides a way to instrument a program at the start of execution, but requires the data collection to remain static throughout the program's execution.

All of the data collection approaches described in this section, require that the instrumentation remain fixed throughout the application's execution. However, when monitoring long running programs, it is desirable to change instrumentation while the program is running.

2.2.2. 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 2.2 shows some of the ways in which programs can be externally monitored.

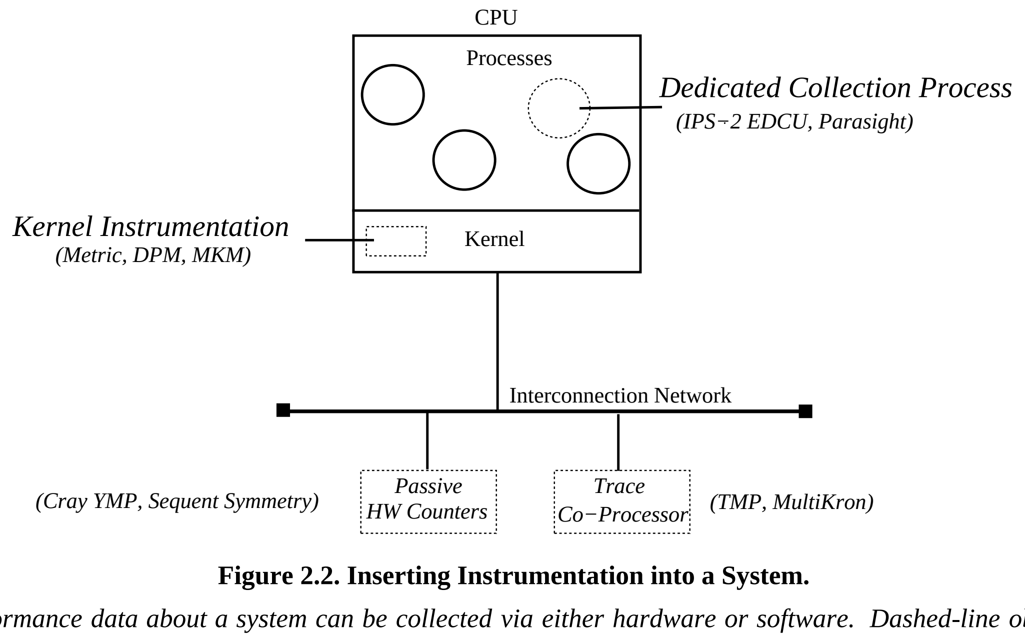

Figure 2.2. Inserting Instrumentation into a System.

Performance data about a system can be collected via either hardware or software. Dashed-line objects show the different ways to collect performance data; and the systems that use each approach are shown in italics.

A common approach to collect performance data is to modify the operating system to collect traces of kernel events and forward them to a user process. Kernel based instrumentation has two advantages. First, it is easy to instrument an application; 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 to collect from a user process or via hardware. However, kernel based approaches suffer several limitations: modifying a kernel is time consuming, error prone, and users need to run a modified kernel to use the tool. METRIC[63] is an early example of using a modified kernel for data collection. Smith's Spider debugger[86] used kernel based instrumentation to track all inter-process communication, but it only worked on a single machine. The Distributed Program Monitor[64] traces inter-process message passing in a distributed environment. The Mach Kernel Monitor[56] 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[39] 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. By using these system calls, data that could only be collected by directly instrumenting the operating system is available to external data collection processes. Special collection processes can also be used to gather information about specific applications. For example, parasight[2] uses dedicated collection processes to gather both operating system and application statistics via shared memory.

2.2.3. Filtering

Using software based instrumentation makes it possible to track a vast number of metrics and events during an application's execution. However, tools are often limited by the large amount of performance data they can generate. One approach to reduce the amount of data is to filter it and store only interesting events. EDL[5], Segall et. al's *receptacles*[81], and Jade[46] use predicates and actions to recognize desired events (or sequences of events) and then generate synthetic events. ISSOS[80] and Meta[61] also use predicates to detect interesting states in an application, but require the user program to periodically call the filtering system. 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, 7], based on EDL, permits dynamic insertion of predicates. BEE[16] allows the dynamic control of the filtering of events based on both event type and by insertion of more complex predicates called "event interpreters". TOPSYS[9] and JEWEL[51] require that predicates be defined before the program starts execution, but permit dynamic control of which ones are enabled. XAB[8] provides dynamic selection of tracing of messages by message type for PVM[25] applications. A limitation of dynamically configured filters is that they necessitate runtime overhead even when no data is being collected. Also, none of these systems provide any assistance to the user in selecting what events to collect and when to collect them. With Dynamic Instrumentation we take a different approach. Instead of collecting large amounts of data and providing dynamic filters, we only collect the data we are interested in gathering.

2.2.4. Hardware Data Collection

One approach to reduce the resource contention between the data collection system and the system being measured is to build dedicated hardware for data collection. Data collection hardware ranges from passive monitors to hybrid software and hardware for trace generation.

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 that record events on the bus, in the memory hierarchy, and on the processor. The performance monitors built into the Sequent Symmetry[92], RP3[12], Power2 (RS-6000)[97] and the PEM Monitor[17] use programmable hardware counters. Advantages of passive monitoring include no perturbation of the observed system, and a simple hardware design. 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[18] is passive, but it is able to track performance data on a per process basis, albeit with additional overhead during context switching. Per process passive monitoring is useful for application performance debugging in a multi-user environment. However, passive monitors have a number of limitations. First, not all interesting events in a program's execution are visible to the hardware monitor. Second, since there are more event signals than counters, the user must select the appropriate events to monitor.

An approach to solve the visibility limitations of passive hardware monitors is a hybrid software and hardware monitor. One example of a hybrid monitor, used in the ZAHLMONITOR[38], is to have the application signal an event occurrence to the hardware by writing to a special range of memory locations. The hardware keeps a count of the event occurrences. 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[68, 69], TMP[35, 99], M31[76] and HYPERMON[60] are examples of this type of monitor. An advantage of hybrid instrumentation is that it eliminates most perturbation of the CPU and inter-connection network, and permits information visible only to the application to be collected. However, the major limitation is that hybrid monitoring can generate so much data that it swamps any file system or data reduction station.

The tools and techniques presented in this chapter show the wide variety of approaches taken to try to improve the performance of parallel programs. Although each system made different design tradeoffs, a common area where most tools could be improved is in the type of guidance and decision support they supply the user. Also, there is more data available than can be efficiently collected. As a result, a subset of this data can be gathered at once. Unfortunately, current approaches to data collection require that instrumentation decisions remain fixed during program execution.

Chapter 3

THE W^3 SEARCH MODEL

Parallel program performance debuggers exist to answer programmers' questions about why their application does not run as expected. For a given parallel program, the amount of performance data that might be useful to answer these questions can be huge. However, in practice a small amount of information is often sufficient to reveal the bottleneck. The goal of performance debuggers should be to help the programmer find the gems of understanding among the large space of available performance data. To do this, a well-defined, logical search model is required. Such a model should provide a structured way for programmers to quickly and precisely isolate a performance problem without having to examine a large amount of extraneous information. This chapter describes the W^3 Search Model, a system that provides these features. It is based on answering three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To deliver answers, rather than simply posing the why, where, and when questions, we need to automate the search process. In this chapter we also describe how to achieve this automation.

3.1. Goals

Before it is possible to develop a methodology for performance debugging, it is important to understand how programmers seek out the answers to their performance problems using current tools. In the course of developing the IPS-2 performance tools, we had an opportunity to observe how several users approached the tool. Several trends emerged. A common starting point was to display a graph of CPU utilization vs. time for their entire program. After digesting this information, they either selected new metrics (e.g., synchronization or I/O), or isolated CPU utilization to specific parts of the program (e.g., processes or procedures). Programmers considered more and more specific information until they understood the problem well enough to starting changing their program. Sometimes they were unable to isolate a performance problem using one approach, but by looking at slightly different data, the relevant phenomena could be explained. For example, although IPS-2 was unable to relate performance of operating system metrics (e.g., page faults) to specific user functions, programmers were able to infer which procedures were responsible by restricting the page fault metric to a specific interval of time. We also discovered that programmers often asked ‘‘which metric should I look at first (or next)?’’ IPS-2 made viewing data easy, but programmers were overwhelmed by their choices. The lessons we drew from this experience were that a performance tool should provide programmers guidance about what to look at and help them conduct an iterative refinement process. However, the search process should permit different ways to refine the problem based on the programmer's mental model and the available data.

Although watching programmers use existing tools provides insight into how to develop a methodology of performance debugging, it is not sufficient. Much to the dismay of tool builders, it has been observed that even on platforms with fairly sophisticated performance tools, most parallel programers do not use the available performance tools[72]. Instead, programmers often manually insert code into their program to understand its performance. Looking at the way in which programmers manually search for bottlenecks provides hints about how to automate the process. For example, a common way to look at programs performance is to insert calls to timer primitives to measure the time spent in a particular region of the program. This provides an idea of where the program spends its time. Additional timers and counters are then inserted and the program re-executed to refine the source of the bottleneck. This step is iterated several times to isolate the cause of the program's poor performance. The methodology of performance debugging with hand instrumentation is similar to our observations about how programmers use IPS-2; programmers start with a global view of their program, and iteratively isolate the source of their performance problem. These observations form the basis for the organization of the W^3 Search Model.

The goal of the W^3 Search Model is to help programmers find the bottlenecks in their programs. We define a bottleneck as a component of the program's execution that contributes a significant amount of time to the application's execution time. By our definition, a single execution of a program may contain several bottlenecks. Also, we do not judge how difficult a particular bottleneck might be to improve. The programmer must decide what, if anything, can be done to reduce the time spent at the bottleneck.

The W^3 Search Model is designed to automate the search for performance bottlenecks, but let the user control the degree that they are involved. It can respond directly to commands from the user, or it can be run in automated mode. In the automated mode, the source of a performance problem is iteratively refined to a specific cause and location in the program. The user can also make a manual refinement of the problem, and then have the system continue trying to further refine the problem. No matter how good the tools get, we feel programmers will continue to play an integral part in performance tuning.

In addition to solving the information overload problem, the W^3 Search Model helps to manage the data collection problem. Dynamic instrumentation affords us an opportunity to greatly reduce the volume of performance data collected, but requires that numerous choices be made about what data to collect and when to collect it. Providing direct control of instrumentation to programmers complicates the already formable task of tuning a parallel program. A goal of the W^3 Search Model is to manage dynamic instrumentation so programmers can enjoy low data collection overhead without incurring the complexity of making instrumentation choices.

Two additional goals for our model are usability and portability. Programmers should not have to write their application so that it works with a specific performance tool. They should able to write their application in whatever style they wish, and not be restricted to a specific programming model. They also should not be required to manually add code to their application to collect performance data. Instrumentation should be inserted transparently and automatically when needed. Finally, the search model should work for a variety of machine architectures and programming styles. This is a particularly difficult goal due to the diversity of machines and styles, and our desire to provide relevant guidance for a specific program on a specific machine.

We have described three major goals for our search system. First, it should provide a structured approach to help programmers understand their programs. Second, it should permit searching to be automated. Third, it should provide a way to drive dynamic instrumentation. To achieve these goals, we developed the W^3 Search Model, which permits users to independently refine the ‘‘why’’, ‘‘where’’, and ‘‘when’’ of a program’s performance. Looking for a performance bottleneck is simply a matter of refining the answers to these three questions. The search process can be thought of as traveling from one point to another in a multi-dimensional space, and at each step we can move in any direction. From each point, several refinements can be tried and the most promising ones selected as candidates for further refinement.

3.2. ‘‘Why’’ Axis

The first performance question most programmers ask is ‘‘why is my application running so slowly?’’ To answer this question we need to consider what types of problems can cause a bottleneck in a parallel program. We represent these potential bottlenecks with *hypotheses* and *tests*. Hypotheses represent the fundamental types of bottlenecks that occur in parallel programs independent of the program being studied. For example, a hypothesis might be that a program is synchronization bound. Since hypotheses represent the universal activities in any parallel computation, a small fixed set of hypotheses, provided by the tool builder, can cover most performance problems.

Tests are boolean functions that indicate if a program exhibits a specific performance behavior related to the hypothesis. They are expressed in terms of thresholds that indicate when a test should return true (e.g., more than 20% of the execution time is spent waiting for synchronization). Tests are provided by the tool builder rather than the user. However the user can (but generally does not have to) modify values of the thresholds. Generally the only time thresholds needs to be changed is when the tool is ported to a model of a new parallel machine.

Hypotheses can have other hypotheses as pre-conditions. The dependence relationships between hypotheses define the search hierarchy for the ‘‘why’’ axis. These dependencies form a directed acyclic graph, and searching the ‘‘why’’ axis involves traversing this graph. Figure 3.1 shows a partial ‘‘why’’ axis hierarchy with the current hypothesis being that the application has a HighSyncBlockingTime bottleneck. This hypothesis was reached by first concluding that an SyncBottleneck exists in the program.

To make the relationship between hypotheses and tests more concrete, consider the hypothesis and test shown in Figure 3.2. This example shows a simple hypothesis, SyncBottleneck, that defines the source of a bottleneck due to excessive time spent waiting for synchronization. Associated with it is a test, HighSyncTimeTest, which returns true when the average time spent waiting for synchronization operations is greater than HighSyncThreshold, defined as 20%. The test uses two metrics in its boolean expression. SyncTime is the time spent waiting for synchronization operations divided by wall time, ActiveProcesses is the number of processes in this application. (The details of the metric definition are omitted here.) This example shows how a hypothesis represents an abstract type of bottleneck and how tests define the specific numeric criteria for a bottleneck.

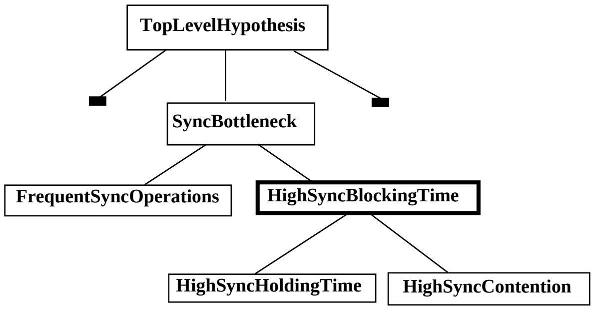

Figure 3.1. Sample ‘‘Why’’ (hypothesis) Hierarchy.

```
constant HighSyncThreshold = 0.20;

metric SyncTime { ... }

metric ActiveProcesses { ... }

hypothesis SyncBottleneck {
  precondition none;
  test HighSyncTimeTest;
  description "The program is synchronization limited.";
}

test HighSyncTimeTest {
  return (SyncTime / ActiveProcesses > HighSyncThreshold);
}
```

Figure 3.2. Pseudo code for a hypothesis and test.

Searching the ‘‘why’’ axis is an iterative process. First we select a hypothesis and then we conduct the test to see if it is true. If it is, we consider possible refinements of this hypothesis, and then test them. For example, a hypothesis might be that an application has a synchronization bottleneck. If the hypothesis is true, then we consider possible refinements: frequent synchronization operations, or too much time waiting for synchronization operations to complete. We then test each of these hypotheses to see if any of them are true.

We define a hypothesis to be true when the tests associated with it return true for the data collected. We continue to verify that the tests conditions associated with a hypothesis remain true as we collect more performance data. Our definition of a true hypothesis is really a *working hypothesis* that meets the test criteria; it should not be confused with a statistical hypothesis or a scientific hypothesis, which have different connotations.

The hierarchical ‘‘why’’ axis capable of searching for many different types of bottlenecks is one of the main contributions of this thesis. Many tools look only at one possible type of problem (e.g., memory bottlenecks, or synchronization). Other tools can display multiple types of bottlenecks, but do not guide the user’s search process. Our structured model helps to focus the user on the problem, not bury them with details.

3.3. ‘‘Where’’ Axis

By searching along the ‘‘why’’ axis we classify the type of problem in a parallel application; to fix the problem, more specific information is required. For example, knowing that a program is synchronization bound suggests we look at the synchronization operations, but a large application might contain hundreds or thousands of these operations. For our search model to be useful, we must find which synchronization operation is causing the problem. We must not only classify the bottleneck, but also identify the resources that are responsible for it. To isolate a bottleneck to a specific resource, we search along the ‘‘where’’ axis.

The ‘‘where’’ axis is formed by a collection of logically independent resource hierarchies. Typical resource types include: synchronization objects, source code, threads and processes, processors, and disks. There are multiple levels in each hierarchy, and the leaf nodes are the instances of the resources used by the application. Searching the ‘‘where’’ axis is iterative and consists of traveling down each of the individual resource hierarchies. Each resource hierarchy can be refined independently. The ‘‘where axis’’ can be viewed as an n-dimensional space where n is the number of resource hierarchies.

The left tree in Figure 3.3 shows a sample resource hierarchy. The root of the hierarchy is Synchronization Objects. The next level contains four types of synchronization (Semaphore, Message Recv., Spin Lock, and Barrier). Below the Spin Lock and Barrier nodes are the individual locks and barriers used by the application. The children of the Message Recv. node are the types of messages used. The children of the Semaphore node are the semaphore groups used in the application. Below each semaphore group are the individual semaphores.

Different components of the ‘‘where’’ axis are created at different times. Some nodes are defined statically, some when the application starts, and others during the application’s execution. The static components are at the top of each hierarchy, and the more dynamic nodes are at the lower levels in each hierarchy. The root of each resource hierarchy is statically defined. Depending on the hierarchy, other nodes below the root might also be statically defined. The next levels in each hierarchy are defined when the specific hosts and application are selected. This step creates nodes for the types of resources that the application might use. The nodes that get created depend on the types of hosts used and the style of parallelism used by the application. For example, once the application is specified we can identify the synchronization libraries it included. The lowest levels in each hierarchy, representing specific resource instances, are added during the application’s execution when the resource instance is first used.

The current status of the search along the ‘‘where’’ axis is called the *focus*, and consists of the current state of each resource class hierarchy. Figure 3.3 shows a sample ‘‘where’’ axis containing three resource hierarchies. The highlighted nodes show the current focus component of each hierarchy. The focus shown in the figure is all spin locks on cpu #12 used in any procedure.

Searching for performance problems along the ‘‘where’’ axis is called *magnifying* the focus. It is possible to magnify the focus of each hierarchy independently. Magnifying a focus is a four step process. First, we pick a resource class hierarchy to magnify. Next, we determine the children of the current node. Third, we select the potential focuses to consider (i.e., restricting our magnification to a subset of all possible children of the current

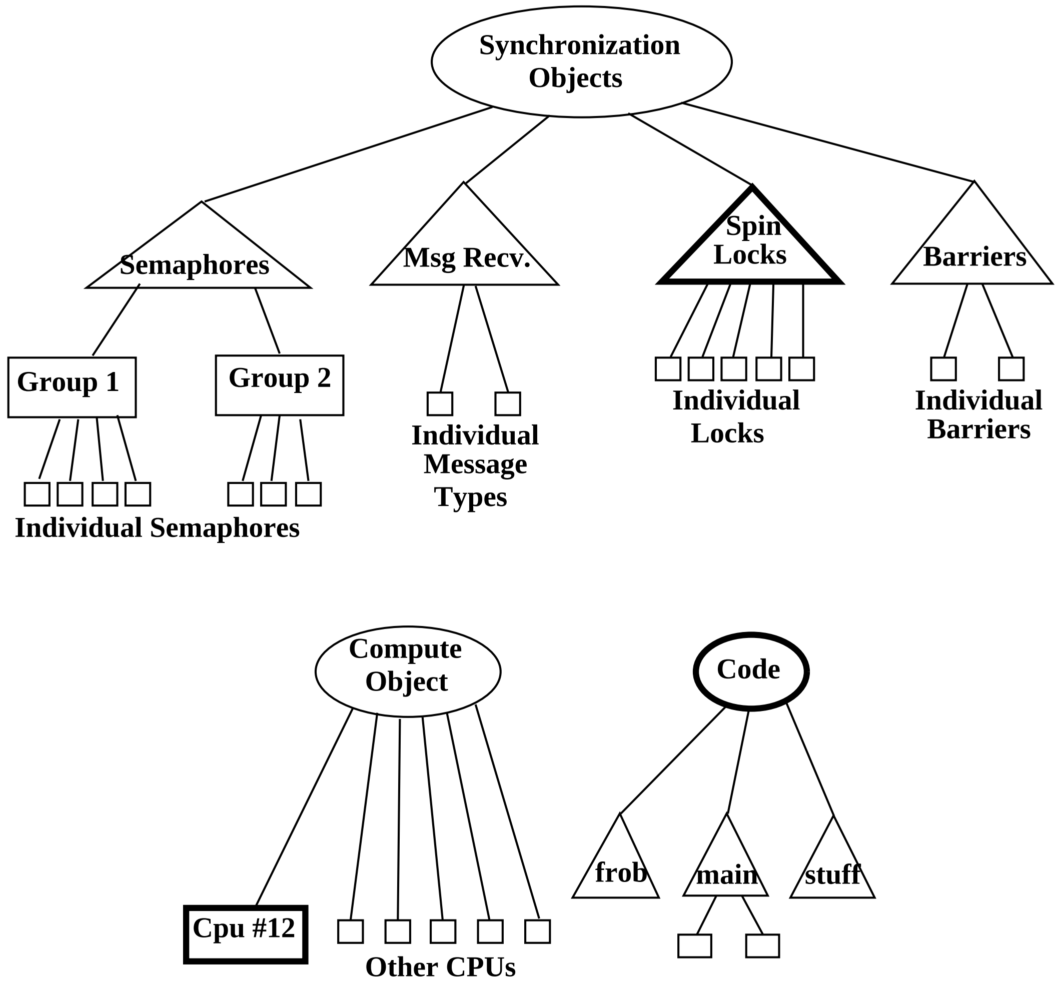

Figure 3.3. A sample ''where'' axis with three class hierarchies.

The highlighted object indicates the current focus. The oval objects are defined in the W^3 Search Model. The triangles are static based on the application, and the rectangles are dynamically (runtime) identified.

node). Last, we test each currently true hypothesis for each potential focus. If the tests indicate that the criteria for hypothesis is met, the potential focus is added to the current focus list; otherwise it is discarded.

Consider a sample magnification starting from the focus shown in Figure 3.3. First, we might select the Code hierarchy. We then get a list of the children of the current node in that hierarchy (the procedures frob, main, and stuff). Then we select which ones to test (we choose to check all of them). Last, we run the tests and conclude that the current hypothesis holds for frob. This results in a new focus for all Spin Locks on Cpu #12 in procedure frob. If the test was true for more than one procedure, they would all be added to the focus.

3.4. ‘‘When’’ Axis

Programs in general, and especially parallel programs, have distinct phases of execution. For example, a simple program might have three phases of execution: initialization, computation, and output. Within a single phase of a program, its performance tends to be similar. However, when it enters a new phase, behavior of the program can change radically. When a program enters a new phase of execution, its performance bottlenecks also change. As a result, decomposing a program’s execution into phases provides a convenient way for programmers to understand the performance of their program.

Phase analysis has been used to understand and tune programs for years. In the 1950’s, programmers drew diagrams by hand of the phase behavior of their programs to manage manual memory overlays[24]. More recently, Denning noted the importance of recognizing phases and phase changes automatically in a memory management system[23]. In parallel computing, tools such as IPS-2[65] provide ways to restrict program measurement to a specific phase of execution. Experience with watching programmers use this feature of IPS-2 convinced us that phase analysis is a major way programmers look for performance bottlenecks.

The third component of the W^3 Search Model is the ‘‘when’’ axis. The ‘‘when’’ is a way for programmers to exploit the phase behavior of their programs to find performance bottlenecks. Searching along the ‘‘when’’ axis involves testing the current hypotheses for the current focus during different intervals of time during the application’s execution. For example, we might choose to consider the interval of time when the program is doing initialization. Figure 3.4 shows the phases of a hypothetical program along with the start and end of each interval. A generalization of the ‘‘when’’ axis is to permit the nesting of time-intervals, and refining bottlenecks to these nested intervals. However, due to the constraints of execution time searching (described in the next section), nested time intervals are not part of the current implementation of the W^3 Search Model.

An important question about the ‘‘when’’ axis is how to define the interval. This is partially a user interface problem and partially a search problem. A simple approach to this problem is to present users with a time histogram (showing one or more metrics and updated in real time), and let them select the start of new time intervals along the histogram. This solution, although simple, has the problem that the user must pay close attention to the program’s

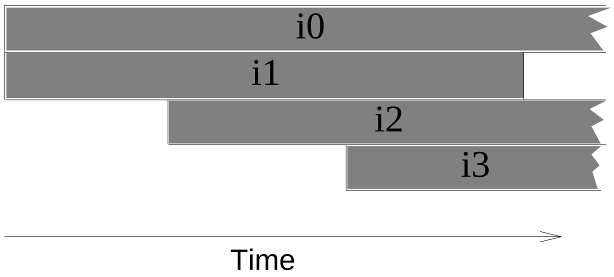

Figure 3.4. Intervals during a program’s execution.

execution. An alternative is to permit the user to specify trigger predicates that start a new time interval when they evaluate to true (or provide a library of these predicates and let the user choose from them). Examples of predicates are the first time a selected procedure is called, or when the synchronization wait time is above a selected threshold. This method requires less direct involvement by the user. Existing correctness debuggers (e.g., Spider[86], and TOPSYS[9]) use similar predicates. Also, PMPL[26] uses similar predicates to trigger data recording. A third approach is to have programmers annotate their programs with calls to library routines to indicate major parts of the computation. This approach can be quite effective, but is not very elegant because it requires the programmer to modify code. Due to the difficulties of automatically detecting phase boundaries, we currently use manual interval specification.

3.5. Searching for Bottlenecks during Execution

The W^3 Search Model is designed for execution time search of performance problems, and to dynamically turn on and off data collection. As a result, we must isolate performance problems with a single pass (made during the execution) over the performance data. Searching using a single pass over the data is difficult because we do not have all of the data we might want. Although the W^3 Search Model is designed for execution time search, it provides the same structured search and automated isolation of bottlenecks when used with post-mortem trace based tools.

Before we can describe how to evaluate based on partial data, we first consider how to search if all the data were available. At each step during the search, we evaluate one or more tests for a focus. In addition, the test evaluation may be constrained to a specific interval of time from the ‘‘when’’ axis. The data required to evaluate a test is the average value of its metric-focus pairs during the desired interval. For example, consider the hypothesis and test shown in Figure 3.2 evaluated for the focus shown in Figure 3.3 during the entire program’s execution. To evaluate this test, we need to know the value of the metrics `SyncTime` and `ActiveProcesses` for all `SpinLocks` used on `CPU #12`. However, during program execution, much of this data is not yet available.

At most, data is available from when the program started until the current time, but for each hypothesis we only have data for the interval from when we started to evaluate the hypothesis to the present time. Consider the sample time interval shown in Figure 3.5. The interval starts at the point marked *Start Interval*, and continues until we decide to start collecting data for this interval (*Start Collection*). We have performance data from *Start Collection* until the point marked *Current Time*, but the interval continues for an unknown length of time until the *End Interval*. To handle execution time searching, several problems need to be overcome:

(1) How long is the current time interval going to last (where is *End Interval*)?

(2) How can we conclude that a hypothesis is true until we have data for the entire interval?

(3) How can we ensure that the partial data we are using is representative of the entire phase?

(4) How do we handle missed data due to selecting a time interval after it has started (i.e., no data is available from *Start Interval* to *Start Collection*)?

(5) What if the user selects a time interval that has already ended (*Start Collection* is after *End Interval*)?

The first question is how long does an interval last? Ideally, we would use data for an entire time interval to determine if a bottleneck exists for that interval. During execution, we do not have complete data for the interval until it is over. We would like to be able to make refinements to sub-intervals without having to re-execute the application, so it is necessary to conclude if a hypothesis is true for a time interval before the interval ends. To do this, we approximate the result of a test by using the cumulative performance data for the current interval from when it is selected until the current time. As the interval continues, additional data is collected, and it is considered when evaluating the hypothesis. We also require a minimum observation time (shown as *minimum observation* in Figure 3.5) during an interval before we conclude that the data is valid. This prevents transient conditions at the start of an interval from causing false conclusions. This approach permits evaluating multiple hypotheses before the interval ends, as well as aggregating over a long enough interval to see meaningful trends in the application's performance.

The second question is how to make decisions based on partial data. An important question about automated refinement is how long to evaluate a hypothesis before concluding the test criteria are not met. We want to try several refinements per execution, so we define a *sufficient observation time* (typically several times the *minimum observation time*), and if we have not concluded a hypothesis is true after waiting this time interval, we consider other hypotheses. Discontinuing testing a hypothesis is different than concluding it is false. To conclude a refinement is false we need to see data for a sufficiently large fraction of the time interval to ensure that no matter what happens in the remainder of the interval, the result of the test associated with that hypothesis will be the same.

The third question is how do we know the data is representative of the entire phase? We ensure that the partial data is representative in two ways. First, the minimum observation time filters out transient phenomena by averaging the data over a minimum window of time. Second, when we conclude that a hypothesis is true, we continue to gather data for the rest of the phase and evaluate the tests associated with that hypothesis. If the test associated with the hypothesis latter becomes false, we report this to the user. For the search to be valid, it is important that data be collected for a significant fraction of the time interval.

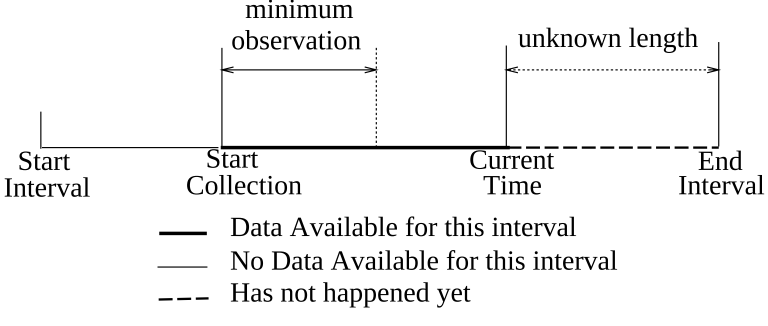

Figure 3.5. Obstacles to changing data collection during execution.

The fourth question is what to do when we start collecting data and testing hypotheses for a time interval after the interval has begun? Since we average performance over the entire interval and we know when the interval started, we can calculate how much time we missed. We use this information as a second constraint on the minimum observation time by requiring data be collected for a minimum percentage of the time interval. This limits the impact of the missed data on our cumulative average. If the time interval ends before we have seen enough additional data, we consider it to be a time interval that has been missed (i.e., the fourth question), and the user has to re-run their application to try hypotheses for this interval.

The fifth question is what happens if we miss an important phenomena during a program's execution because we were not collecting data about the phenomena when it occurred? This is not a problem since the interesting bottlenecks that limit the performance of a long running program by definition last a relatively long time. As a result, we have a long time interval to find and isolate the cause of the problem. If a behavior lasts for only a short interval of time it might get missed, but since it is short, its impact on the total execution-time of the program is also small. For a short running program, the fact that it is short means the time to re-run it is also short.

There are three potential problems with using data from only part of an interval. First, the tests for a hypothesis could be true based on averaging data over a short time at the start of an interval, however they might not be true for the entire interval. This effect can be mitigated by selecting a long enough minimum observation time to permit the system to enter steady state. Even if it happens (as long as the minimum is reasonable), the interval where the hypothesis is true is an indication of a potentially interesting sub-interval. The second problem is that hypotheses, that are true for a long interval of time, build up momentum (due to averaging metric values over a long time interval) and might appear to be true after they become false. However, this is not a problem since we defined the value of a metric for a time interval to be the average over *all* of the interval. If the average is high enough to satisfy our test criteria, its high enough; we do not care about the distribution of time during the interval.

The third problem is what if the running average value of the metric oscillates about the test threshold. These oscillations would cause any test that used the metric to alternate between true and false. To address this problem we include a *hysteresis* parameter. The hysteresis parameter splits the threshold for determining when a test is true into two parts. One threshold determines when a true test is considered to go false, and the other determines when a false test becomes true. The hysteresis parameter is real number between 0 and 1. We multiple the threshold used for a transition from false to true by the hysteresis parameter to get the threshold for the transition from true to false. A value of one effectively disables hysteresis since the thresholds for true to false and false to true are the same. A value of 0 means that once a test becomes true, it can never changes back to false. Appendix A provides a discussion that shows that the hysteresis parameter achieves the desired damping.

In this section, we have developed a way to search for performance bottlenecks using the partial data available during program execution. In Chapter 4 we describe a prototype implementation our methodology and report the result of applying it to several applications.

3.6. Search History Graph

The three axes of the W^3 Search Model help to guide the search process, but they only provide a snapshot of the current state of the search. It is important to know what performance problems have been considered, not just the ones currently being tested. To capture the temporal aspects of the search we include an abstraction called the *Search History Graph* in our model. The Search History Graph records each refinement considered along the ‘‘why’’, ‘‘where’’, and ‘‘when’’ axes, and the result of testing the refinement. For example, isolating a performance problem to a specific barrier might require testing each barrier separately and the Search History Graph indicates which barriers have been tested. Each node in the graph represents a single refinement along the ‘‘why’’, ‘‘where’’, or ‘‘when’’ axes. Nodes are present only for those states that have been tested. The arcs indicate refinements. This graph is useful because it not only represents refinements that were made, but also those that were tried and rejected, and those that were possible, but not tried. Figure 3.6 shows a sample search history tree. The search started at the root of the graph and three refinements have been made (nodes 3, 5, and 6).

3.7. Automated Searching

A key component of the W^3 Search Model is its ability to automatically search for performance bottlenecks. This is accomplished by making refinements across the ‘‘where’’, ‘‘when’’, and ‘‘why’’ axes without requiring the user to be involved. Automated refinement is exactly like manual (user directed) searching, and hybrid

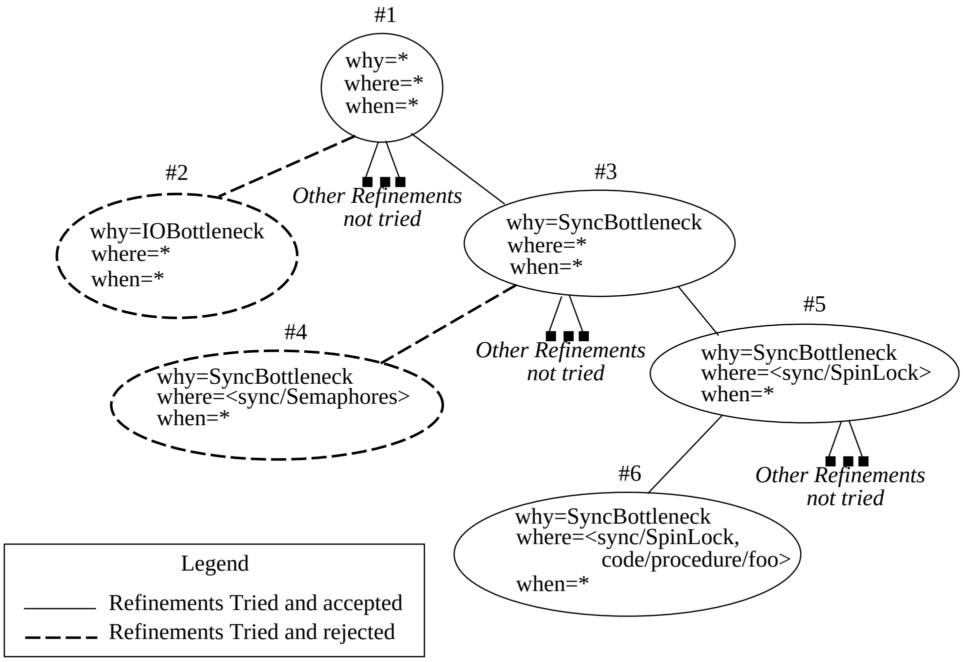

Figure 3.6. Sample Search History Graph.

combinations of manual and automated searching are possible.

Selecting a refinement is a three step process: determining all possible refinements, ordering the refinements, and selecting one to try. We determine the possible refinements by considering the children of the current nodes along each axis. We then use hints (described below) to order the list of possible refinements. Finally, we select one or more refinements to try from our ordered list. If the first one tried is not true, we consider the next item from the ordered refinement list.

The axes of the W^3 Search Model help to direct the user to constrain the search for performance bottlenecks; however at each step there are several choices of possible refinements to consider. Since the number of refinements that can be tried at once is limited, the order in which different refinements are tried can greatly reduce the time required to isolate a performance bottleneck. To provide guidance in selecting good refinements, our search model includes *hints*. Hints are suggestions about future refinements that are generated as a side effect of testing earlier refinements. Hints have two important characteristics. First, they are only hints and exist to order refinements, not alter what refinements are possible. Second, hints provide a mechanism to convey relationships between the axis. For example, a ‘‘why’’ axis hypothesis for a synchronization bottleneck might return a hint to refine along the ‘‘where’’ axis based on synchronization objects.

3.8. Explanation Interface

Once the W^3 Search Model has been used to automatically isolate a performance problem, we need to relate what has been discovered back to the user. No matter how good a performance tool is at finding problems, unless that information can be related back to the programmer in a way that is meaningful to them, the tool is useless.

The Explanation Interface is how the W^3 Search Model conveys to the user what is causing their program to perform poorly. Explanations might involve creating new displays, changing existing ones, performing analyses or even a simple textual output. We have three goals for the explanation interface: provide sufficient context to the user so that the information is meaningful, incorporate existing visualizations and analyses, and provide guidance in selecting visualization and analyses.

Providing a context for the explanation is important because if the W^3 Search Model is run in automated mode, it might make several refinements before reporting results to the user. The Search History Graph contains the information necessary to provide the context for an explanation. For example, assume we found a synchronization problem that was caused by three message tags used in a single procedure. The Search History Graph not only contains the three message tags to profile, but also contains the name of the procedure. A meaningful explanation includes as much information as we can provide about the problem.

Another important aspect of the Explanation Interface is its ability to incorporate multiple displays and analyses. Many useful visualizations of performance data have been created, and if used correctly, they provide quick explanations of complex interactions. However, selecting a fixed set of these visualizations to build into the W^3 Search Model limits its usefulness. Instead we provide a standard interface to visualizations and analyses that permits our system to not only create a display, but also to alter displays the user has already created. For example, if the user has a table displayed, the Explanation Interface could highlight important items or sort the table to illustrate the bottleneck.

A goal of the Explanation Interface is to provide guidance in selecting visualizations and analyses. This is accomplished by using explanations as actions that are associated with hypotheses. Explanations get triggered when a hypothesis evaluates to true for a selected focus. Although the full library of visualizations and analyses is available to the user all the time, the W^3 Search Model will only create displays when they are relevant. In addition, since they are triggered when a specific refinement is true, we can use the state of the ‘‘where’’ and ‘‘when’’ axis as parameters to the visualizations so that the information displayed is relevant and focused to the current bottleneck.

This chapter has described how the W^3 Search Model addresses the problem of programmer information overload. The combination of a systematic, iterative search model, a way to automate the search, and an explanation of the results provide the components necessary for a performance monitoring environment that gives programmers the precise answers they need to tune their programs.

Chapter 4

IMPLEMENTATION OF THE W^3 SEARCH MODEL

Many of the techniques and ideas used in the W^3 Search Model are new, so we implemented a prototype of the model, called the Performance Consultant, to study the ability of our search model to identify performance bottlenecks. We were interested in studying the W^3 Search Model in isolation, so for this study we decided to simulate the Dynamic Instrumentation system on top of IPS-2[65], which uses trace based instrumentation. Using trace data meant we could make multiple runs of the Performance Consultant using the same data to calibrate our search system. Another advantage of using IPS-2 tracing is that we could compare the amount of performance data generated by our method to existing trace based and sampling approaches. We used our prototype to study three applications: two are from the Splash benchmark suite[83] and one is a database application.

To validate the guidance supplied by the Performance Consultant, we also studied each application program using the IPS-2[65] performance tools. This manual check permitted us to verify that the bottlenecks identified by the search model were real, and to check that no obvious bottlenecks were omitted.

Running programs and simulating Dynamic Instrumentation permitted us to study the search model in isolation. However, it is also important to evaluate the W^3 Search Model using a real version of dynamic data selection. A study that combines the W^3 Search Model and Dynamic Instrumentation appears in Chapter 8.

4.1. Experimental Method

Our test implementation included 15 hypotheses (shown in Figure 4.1) including CPU, I/O, synchronization, and virtual memory bottlenecks. To simplify our prototype implementation, we wrote hypotheses and tests as C++ functions compiled into the system. In this experiment, three resource classes were used: code, process, and synchronization object. In a full implementation, we would have additional resources classes but for these experiments we restricted the resources classes to those that could be extracted from the IPS-2 trace data. This means that generally we could select one of 5 to 10 possible refinements (1 to 5 hypotheses, and several ‘‘where’’ axis refinements). In some cases, for example at the procedure level, we had over 50 possible refinements. We also implemented a manual version of the ‘‘when’’ axis.

Since our goal in this study was to evaluate the W^3 Search Model in isolation, we used trace data generated by the IPS-2 performance tool and simulated dynamic data selection. A benefit of this approach is that it permitted us to compare the quality of guidance supplied using dynamic selection to that of full tracing. In addition, IPS-2 runs on a variety of platforms, making a large set of sample data available for the tests. IPS-2 records event traces during a program’s execution. Each event (e.g., procedure call or synchronization operation) contains both wall-clock and process time stamps in addition to some event specific data. In addition to normal IPS-2 instrumentation, we ran the programs with two External Data Collectors[39]. External Data Collectors are dedicated sampling processes that collect additional information not available via tracing. One collector gathered information about the behavior of the operating system (e.g., page faults, context switch rate). The other collected data about the hardware (e.g., cache miss rates and bus utilization).

Once the application had completed execution, we pre-processed the trace files into uniform time histograms and used these time histograms to simulate a real-time execution. Each histogram bucket corresponds to a sample being delivered to our system. We simulated an execution-time tool by evaluating tests only when new performance data was ‘‘delivered’’ to our system. Dynamic Instrumentation was approximated by ‘‘enabling’’ and ‘‘disabling’’ data collection as we refined along the axes of the W^3 Search Model. For example, when we started our simulation, the only data being collected was to evaluate the top level hypotheses for the entire application. When a new hypothesis was considered, or a refinement made along the ‘‘where’’ axis (e.g., looking at data for a specific procedure), we ‘‘enabled’’ the collection of the necessary data. When evaluating tests, only data for the time interval from when the data for that test was ‘‘enabled’’ until the current time was considered. We also simulated the minimum observation time by not evaluating tests until the performance data for that test had been ‘‘enabled’’ for a sufficient time. This technique gave us a good approximation of Dynamic Instrumentation.

Our instrumentation model is based on periodically sampling performance counters and timers from a running program. A counter records an event in the program at a specific focus (e.g., a procedure call counter exists for

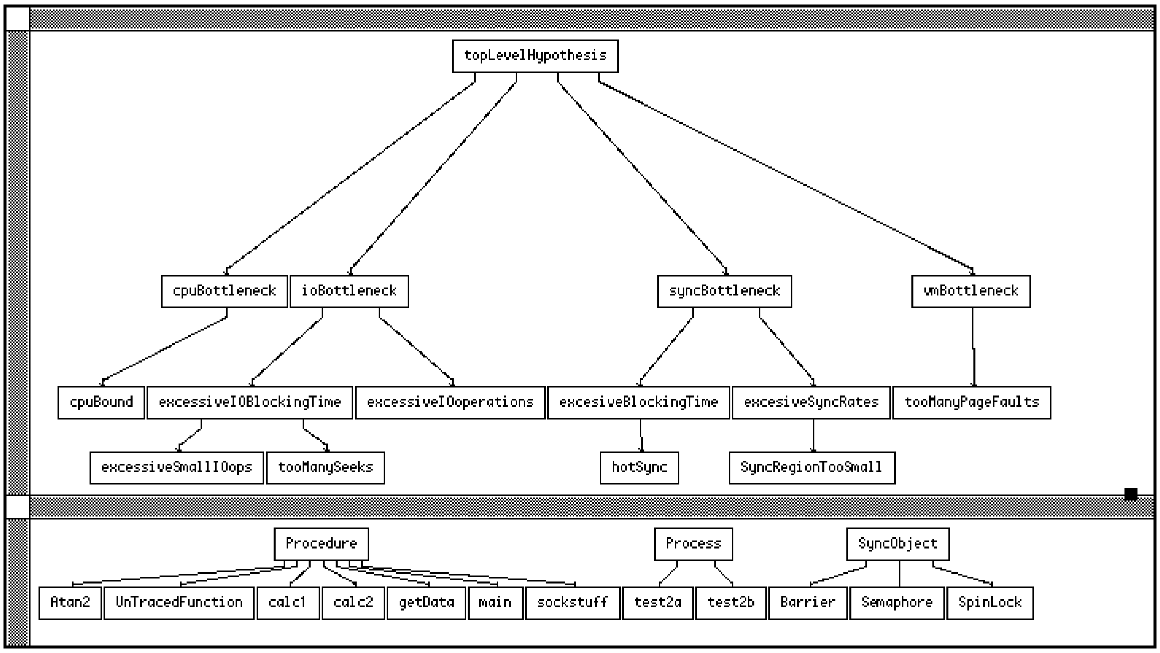

Figure 4.1. A Display showing the ‘‘why’’ and ‘‘where’’ axes.

each procedure in the program). However the focus of a counter can be quite specific. For example, a program that uses spin locks not only has a counter for each lock but also one for each lock in each procedure that used the lock. Since we simulated enabling and disabling data collection, we can calculate the number of samples that would be collected for each counter by multiplying how long the counter was enabled by the sampling frequency. We compare the total number of samples our approach needed to full sampling that requires data for every counter at every sample time. We also compare our results to the full procedure and synchronization tracing done by IPS-2.

A simple explanation option was also included in the prototype. An explanation function is associated with each hypothesis. When a hypothesis evaluates to true, the user can request an explanation, which causes the explanation function to be called. The simplest explanations consist of print statements that describe the type of bottleneck for the hypothesis. More sophisticated explanations report additional information about the program. For example, the explanation for a CPU bottleneck prints a gprof[34] style profile table for the current focus along the ‘‘where’’ axis.

Finally, it is important to understand how we configured the Performance Consultant for our study. Since we were primarily interested in the ability of the tool to automatically find bottlenecks, we ran the system in a mode in which the tool searches for and refines bottlenecks without any user interaction. We somewhat arbitrarily set the minimum observation time to 5 samples (0.1 to 0.5 seconds depending on the sample rate). Likewise the sufficient observation time was set to 10 samples (0.2 to 1.0 seconds).

4.2. Water

The first application we studied was Water, one of the Splash benchmark programs. It is an N-body molecular dynamics application that simulates both the intra- and intermolecular potentials of water molecules in the liquid state. The program primarily uses spin locks for synchronization. We ran this program on a Sequent Symmetry using both 4 processors and 16 processors to see if the performance was different. We also tried small and large input files.

When the program was run on four processors, it ran for 13.8 seconds. The Performance Consultant identified a CPU time bottleneck because 80% of the attempted parallelism was spent in productive CPU utilization. The Performance Consultant found this bottleneck 4.0 seconds into the computation. We were unable to refine this hypothesis to either a specific process, or procedure because no single process or procedure was responsible for most of the CPU time. However, the explanation associated with the CPU bottleneck hypothesis supplied a CPU time profile listing the procedures in the program and the percent time spent in each one. This provided a list of likely procedures to try to improve. To validate this hypothesis, we used the IPS-2 tools and found that the program was indeed CPU bound.

We also ran the program on a larger input file using 16 processors and it ran for 35 seconds. In this case, the Performance Consultant found the program was now synchronization bound (this was also confirmed using IPS-2). The Performance Consultant was able to identify a specific lock variable that was responsible for 43% of the synchronization waiting time and 24% of the total execution time of all of the processes. This was the most specific advice the Performance Consultant gave in our case study. Since the major bottleneck changed when we used more processors and a larger data set, this example showed that it is important to study the performance of parallel programs on real datasets on the desired number of processors.

An important question about our prototype is the volume of performance data needed to find these bottlenecks. Figure 4.2 shows a summary of the performance data collected by our system compared to full sampling. The column for ‘‘Total Counters’’ shows the number of counters required for all of the possible event types and focuses that our search model had at its disposal. ‘‘Counters Used’’ indicates how many of the total possible counters were used to find the program’s bottlenecks. ‘‘Total Samples’’ shows the number of samples required if all of the counters are sampled every 100 msec. ‘‘Samples Used’’ is the number of samples collected for the counters used while they were enabled. Figure 4.3 compares the volume of sample data generated by our approach to the full tracing done by IPS-2. The column labeled ‘‘Samples’’ is the number of samples used (also appears in Figure 4.2). To compare Dynamic Instrumentation to IPS-2 tracing, we needed to figure out how big a single sample from Dynamic Instrumentation would be. Twelve bytes was selected as a reasonable size (4 bytes for a counter identifier, 4 bytes for a time stamp, and 4 bytes for the value). The column ‘‘size’’ shows the number of samples used multiplied by 12 bytes per sample. ‘‘Trace Size’’ is the actual size of the trace file created using the IPS-2 performance tools. ‘‘Ratio’’ shows the ratio of the IPS-2 trace size to the ‘‘Size’’ column.

Our dynamic approach to instrumentation reduced the volume of performance data collected for this program by a factor ranging from 38 to 200 times compared to traditional methods. An interesting aspect of our technique appears in the 16 processor case (shown in Figure 4.2). Dynamic Instrumentation looks at over 4% of the counters, but less than 1% of the total samples. This is because the Performance Consultant was isolating a performance problem to a specific lock variable and needed to measure the lock waiting time for each lock. However, since the Performance Consultant discarded most of the locks after waiting the *sufficient observation time*, we did not need to

CPUs	Total Counters	Counters Used	Total Samples	Samples Used
4	2,022	22 (1.1%)	279,036	1,390 (0.5%)
16	7,783	328 (4.2%)	2,762,965	28,692 (1.0%)

Figure 4.2. Sampling vs. Dynamic Instrumentation for the Water application.

A comparison of the volume of data collected by Dynamic Instrumentation and full periodic sampling for the Water application. Both the number of different counters examined and the number of samples delivered are shown.

CPUs	Dynamic Instrumentation		IPS-2 Trace	Ratio
	Samples	Size (KB)	Size (KB)	
4	1,390	17	1,240	73
16	28,692	343	13,129	38

Figure 4.3. Data used by Dynamic Instrumentation vs. Tracing for Water.

The reduction in data collected possible with Dynamic Instrumentation compared to IPS-2 tracing for the Water program. The sizes are in kilobytes, and assume 12 bytes per sample.

collect the data for each lock for the entire execution.

4.3. LocusRoute

The second application studied, LocusRoute, is also one of the Splash benchmark programs. It is a VLSI tool for doing routing between standard cells, and it computes the area of the resulting layout. We ran the program on a dedicated four processor Sequent Symmetry.

Much to our dismay the Performance Consultant gave us no information about this program. We decided to run IPS-2 on the program to see what was happening. After looking at all of the IPS-2 metrics we were unable to explain the performance of the program either. Finally, we looked at the volume of performance data the IPS-2 system was creating. We discovered that the program makes many procedure calls, and that the instrumentation overhead due to procedure calls was the problem. We confirmed this problem in two ways. First we wrote a test program to see how much overhead tracing a procedure call in IPS-2 involved. Next we used a feature of IPS-2 that permits us to turn off procedure level tracing, and re-ran the program. The Performance Consultant then found a CPU time bottleneck in the resulting program.

We also used the W^3 Search Model model to verify that the observed instrumentation overhead of our system was acceptable. We added an additional hypothesis to identify instrumentation bottlenecks. We defined spending at least 15% of the time in instrumentation as a significant overhead. Using this additional hypothesis we were able to identify procedure call instrumentation overhead as the problem with this program. We were pleased with the ability of our system to test for its own instrumentation bottlenecks.

Figures 4.4 and 4.5 show the amount of performance data collected for LocusRoute both with and without the hypothesis about instrumentation overhead. In the initial version, no bottleneck was found and no refinements were made. Each of the 6 counters required to test the initial hypothesis for the root of the ‘‘where’’ axis are collected at each time interval during the program’s execution. Since we collect more data only when a refinement is made, this shows the minimum amount of data that will be gathered using our approach. In the second case, since we identified an instrumentation bottleneck, more counters were examined as the Performance Consultant tried to isolate the bottleneck to a single procedure. However, even in this case, Dynamic Instrumentation reduces the volume of performance data collected by a factor of 191 compared to full tracing.

4.4. Shared Memory Join

The third application used in this study is an implementation of a join function for a relational database. It implements a hash-join algorithm using shared memory for inter-process communication[22]. The program was written to study shared-memory and shared-nothing join algorithms. We ran the program on a dedicated four processor Sequent Symmetry.

Our test case ran for 93 seconds. The Performance Consultant identified one bottleneck in the program due to excessive page faults. Since the page fault data is collected via an external sampler process, and is not collected on a

Version	**Total Counters**	**Counters Used**	**Total Samples**	**Samples Used**
initial	1,928	6 (0.3%)	337,400	1,050 (0.3%)
collection hypothesis	1,928	44 (2.2%)	337,400	4,226 (1.2%)

Figure 4.4. Full Sampling vs. Dynamic Instrumentation for LocusRoute.

A comparison of the volume of data collected by Dynamic Instrumentation and full periodic sampling for the LocusRoute application. Both the number of different counters examined and the number of samples delivered are shown. The first row shows the data for the original set of hypotheses, and the second row shows the amount of data collected when we added a hypothesis about data collection being the bottleneck.

Version	**Dynamic Instrumentation**		**IPS-2 Trace**	**Ratio**
	Samples	**Size (KB)**	**Size (KB)**	
initial	1,050	12.6	9,696	770
collection hypothesis	4,226	50.7	9,696	191

Figure 4.5. Dynamic Instrumentation vs. Tracing for LocusRoute.

The reduction in data collected possible with Dynamic Instrumentation compared to IPS-2 tracing for the LocusRoute program. The sizes are in kilobytes, and assumes 12 bytes per sample.

per process or per procedure basis, the system could not directly isolate the bottleneck to a specific procedure. However, we could identify the time during the program's execution in which the page fault bottleneck occurred. This shows that even when the search model is not able to precisely isolate a performance problem along one axis (‘‘where’’ in this case), we are able to use another axis (‘‘when’’) to help isolate the problem. This flexible approach to finding bottlenecks is an important characteristic of this work. To validate this result, we again used the IPS-2 performance tools. Since we had previously studied this program[39], we recognized the page fault problem as one of the problems in this program. The problem was due to the creation of new user data in the program. When a page is first referenced by a process in UNIX, the operating system traps the references and zeros out the previous contents of the page. These pages were previously allocated (but not created) by calls to the dynamic memory allocation routine. A few small changes to the program reduced this page fault behavior and improved the execution time by 10%.

Figures 4.6 and 4.7 show the volume of performance data generated for the shared memory join application. For this program we investigated how changing the simulated sampling interval would change the amount of performance data collected. We sampled at intervals of both 100 msec and 20 msec. At 100 msec, we reduced the volume of performance data collected compared to IPS-2 by a factor of 41, and at 20 msec we reduced it by a factor of 13. The ratio of data collected for full sampling to Dynamic Instrumentation, a factor of 220, did not change significantly when we changed the sampling interval.

4.5. Lessons Learned

By building an early prototype of one of the main parts of the W^3 Search Model, we were able to get an understanding of the interactions between the different components in the search model. Most of the lessons have been incorporated into the full implementation described in chapter 8. First, we realized that some hypotheses from the ‘‘why’’ axis were better represented as refinements along the ‘‘where’’ axis. For example, we wrote a hypothesis to look for highly contested (hot) spin locks, but realized we could trivially re-write it to look for hot synchronization objects. This meant it could be used for other types of synchronization too.

We also used the system to do manual searching for bottlenecks and recorded the search path used by the programmer. Based on an informal analysis of these logs, we were able to better define automated searching. For example, we discovered that refining along the ‘‘why’’ axis, then searching the ‘‘where’’ axis and finally the ‘‘when’’ axis helps to reduce the number of options at each step, and is easier to understand than if refinements are made in a random order. On the other hand, if the system returned a hint, we found it was better to consider it first.

We also confirmed our intuition that we need to be able to find more than one bottleneck in a single program. For example, most of the programs we tried have a relatively uninteresting I/O bottleneck at the start of their execution. It is important to report this fact, and continue looking for other bottlenecks during the rest of the execution.

Sample Interval	Total Counters	Counters Used	Total Samples	Samples Used
100 ms.	1,978	9 (0.5%)	1,845,474	8,379 (0.5%)
20 ms.	1,978	9 (0.5%)	9,227,370	41,958 (0.5%)

Figure 4.6. Full Sampling vs. Dynamic Instrumentation for Shmjoin.

A comparison of the volume of data collected by Dynamic Instrumentation and full periodic sampling for the Shmjoin application. The first row shows the volume of data collected when the sampling rate is one sample (per counter and timer) every 100 mili-seconds. The second row shows the increase in data volume when the sampling rate is changed to every 20 mili-seconds.

Sample	Dynamic Instrumentation		IPS-2 Trace	Ratio
Interval	Samples	Size (KB)	Size (KB)	
100 msec.	8,379	101	4,100	41
20 msec.	41,958	504	6,300	13

Figure 4.7. Dynamic Instrumentation vs. Tracing for Shmjoin.

The reduction in data collected possible with Dynamic Instrumentation compared to IPS-2 tracing for the Shmjoin program. The sizes are in kilobytes, and assumes 12 bytes per sample.

We demonstrated the importance of running programs on full sized input sets rather than toy files. For example, the type of bottleneck found in the Water application changed when we ran it on a larger dataset. Running on production size runs showed the importance of building performance tools that scale up to real applications on large scale parallel machines.

Finally, the prototype showed that it is possible to identify and to isolate performance problems by searching using partial performance data. In addition, the reduction in the amount of performance data collected (typically a factor of 50 to 100) means that the approach should scale well to large machines. In fact, we feel that on larger machines, longer running programs, and with instrumentation down to the loop level, these ratios will be even larger because less and less of the available performance data will be useful at any given time.

Chapter 5

DYNAMIC INSTRUMENTATION

Data collection is a critical problem for any parallel program performance measurement system. To understand the performance of a parallel program, it is necessary to collect data for full-sized data sets running on large numbers of processors. However, collecting large amounts of data can excessively slow down a program's execution, and distort the collected data. As we noted in Chapter 2, a variety of different approaches have been tried to collect efficiently performance data. All of the techniques had limitations in either the volume of data they can generate or granularity of data collected. In this chapter, we describe a new approach to data collection based on software instrumentation of the application program. However, unlike other systems that insert instrumentation directly into an application program, we take a different approach about when to insert the instrumentation and what data to collect. We have developed a new style of performance data collection, called *Dynamic Instrumentation*, that defers instrumenting the program until it is in execution. This approach permits dynamic insertion and alteration of the instrumentation during program execution. We also describe a new data collection model that permits efficient, yet detailed measurements of a program's performance.

5.1. Goals

Monitoring the performance of large-scale programs requires an instrumentation system that is *detailed*, *frugal*, and *scalable*. It must collect information that is detailed enough to permit the programmer to understand the bottlenecks in their program. It must be frugal so that the instrumentation overhead does not obscure or overly distort the bottlenecks in the original program. The instrumentation system must also scale to large, production data set sizes and number of processors.

A detailed instrumentation system needs to be able to collect data about each component of a parallel machine. To correct bottlenecks, programmers need to know as precisely as possible how the utilization of these components is hindering the performance of their program. In addition, they need to understand which part of the program or data is causing the problem. An instrumentation system should be able to collect details about how any combination of components of the parallel machine is interacting.

There are two ways to provide frugal instrumentation: make data collection efficient, or collect less data. All tool builders strive to make their data collection more efficient. To reduce the volume of data collected, tool builders are forced to select a subset of available data to collect. Most existing tools require the decisions about what data to collect be made prior to the program's execution. By deferring data collection decisions until the program is executing, we can customize the instrumentation to a specific execution.

The goals of being frugal and detailed are often in opposition. Collecting detailed information requires a lot of data, but frugality says you cannot afford to collect it. This dichotomy can be seen in the choice of how the data is collected. Currently two styles of collection are used: summary counters (and timers) and tracing (logging) of events during an execution. Summary information optimizes data volume over accuracy, and tracing optimizes accuracy over data volume. We developed a hybrid of sampling and tracing that provides the low data volume of sampling with the accuracy of tracing. We periodically sample detailed information stored in event counters and timers. These intermediate values provide data to make decisions about how to change the instrumentation.

Scalable instrumentation requires that data collection remain detailed yet frugal as the number of resources in the system grows. This means that we should be able to handle increases in the application code size, length of program execution, data set size, and number of processors. Providing scalable instrumentation is important because the nature of bottlenecks in parallel programs change as each dimension is increased. For example, it is common for a program to perform well on a small number of processors, but then fail to scale up to a large number of processors due to increasing contention for synchronization variables. It is also possible for a program to run well on small input sets, but fail to scale to large input sets due to increased memory requirements causing unacceptable paging behavior.

We also wish to develop an instrumentation system that removes obstacles that could deter programmers from using our system. For example, recompiling an application to use a performance tool can be an unreasonable burden. The only step required to use Dynamic Instrumentation is to run a stand-alone pre-processor that appends an instrumentation library to the application's binary. The instrumentation is enabled by modifying the application's binary image while it is executing. This technique has no impact on the application until we insert instrumentation. Also, once we insert the instrumentation, there is no overhead to check if it is enabled. For long-running programs, having to re-execute the program to observe its performance is also undesirable. For example, it might be desirable to monitor the performance of a mission critical database server to understand its performance; however it is not possible to stop such an application to enable monitoring. To combat this problem, at any point during the application's execution, the programmer can choose to start a measurement session for their already running program and start to investigate its performance. Both of these features (no re-compiling and starting measurements during execution) would be difficult to provide in a static instrumentation environment, however they were relatively easy to incorporate into our Dynamic Instrumentation system.

Portability is an interesting goal because of the rich variety of hardware, software and programming models in use on parallel machines. The easiest way to achieve portability is to use only the lowest common denominator of features available on the target platforms. This is undesirable because the resulting tool ignores potentially important features of the specific machine being used. We instead chose to design our instrumentation environment in a modular manner that permits different implementations to share commonality, but to incorporate performance data about unique features of each platform (e.g., virtual memory statistics for machines that have it). In addition, we wanted to be able to take advantage of hardware facilities that make implementation of the instrumentation more efficient. For example, the Thinking Machines CM-5 provides a hardware combining network that is useful for aggregating performance data from each node, although the Intel Paragon has no such hardware. Our strategy makes it possible to build high level performance tools that are independent of the underlying hardware, but can still take advantage of unique features and provide detailed information about applications on specific platforms.

While our dynamic approach makes it possible to collect a wide variety of performance data, it also requires that runtime decisions be made about what data to collect and when to collect it. However, we can use the W^3 Search Model to control Dynamic Instrumentation. It is also possible for programmers or other tools, such as visualizations, to manually control data collection.

5.2. Design

To meet the challenges of our goals of detailed, frugal, and scalable instrumentation we needed to make some radical changes in the way traditional performance data collection has been done. However, since we wanted our instrumentation approach to be usable by a variety of high level tools, we needed a simple interface. We developed an interface that is based on two abstractions: *resources* and *metrics*. Resources are the objects about which we gather performance information. Metrics are quantitative measures of performance. In addition, we developed a data collection model based on the periodic reporting of counts and times of events inside the application.

5.2.1. Dynamic Instrumentation Interface

The first abstraction provided by Dynamic Instrumentation is resources. Resources correspond to the ‘‘where’’ axis of the W^3 Search Model. They are separated into several different hierarchies, each representing a class of objects in a parallel application. For example, there is a resource hierarchy for CPUs, containing each processor. Resources are created at various times before and during an application’s execution. For example, the procedure resources are created when the program to be monitored is specified. However, information about data files used is discovered dynamically when the files are opened.

The second abstraction is metrics. Metrics are time varying functions that characterize some aspect of a parallel program’s performance; examples include CPU utilization, counts of floating point operations, and memory usage. They are defined by higher level performance tools such as the Performance Consultant. However, to assist tool builders, we also have defined a standard set of metrics. Metrics can be computed for any subset of the resources in the system. For example, CPU utilization can be computed for a single procedure executing on one processor or for the entire application.

5.2.2. Data Collection

A key question about any performance data gathering system is what data is collected and how is it collected. To provide an efficient, yet detailed instrumentation system we developed a new data collection model that combines the advantages of tracing and sampling. Tracing inserts instrumentation to generate a log of all of the interesting events during a program’s execution; logged events generally include inter-processor communication and procedure invocations. Each trace record typically includes one or more timestamps. Trace-based systems can collect detailed information about specific events during a program’s execution. However, they can generate vast amounts of data that are difficult to manage. A second approach to collecting performance data is to insert instrumentation to summarize interesting information as counts and times that are reported at the end of program execution. Using end-of-execution summary counters and timers greatly reduces the volume of performance data collected; however summary data loses important temporal information about usage patterns and relationships between different components. For example, it is impractical to collect performance data about how each procedure during different phases of execution uses different synchronization variables on each processor for a large parallel machine. Our approach is to record precise information about relevant state transitions in counter and timer data structures. These structures are then periodically reported to the higher layers of our system. Periodic summaries provides accurate information about the time varying performance of an application without requiring the large amount of data needed by full tracing.

Another approach to data collection is statistical sampling. Statistical sampling involves periodic sampling of program resources, such as the currently active procedure, to gauge program performance. Our periodic summaries can yield more accurate information than can be provided by statistical sampling. For example, if you statistically sample the program counter to compute the amount of time spent in a procedure, the accuracy of the result will depend on the sampling rate. Ponder and Fatemen[74] describe a number of other limitations of statistical sampling approaches. Instead, we insert code to start and stop timers at the procedure entry and exit to accurately record time spent in the procedure. In this case, the frequency of summaries affects only our knowledge of the time, not the accuracy of the time. Figure 5.1 shows a comparison of the tradeoffs between our approach and other data collection systems.

We control the volume of data collected in two ways: first by collecting only the information needed at a given moment, and second by controlling the rate of collecting intermediate results. We periodically report our performance metrics for two reasons. First, we want to isolate performance bottlenecks to specific phases (time intervals) during a program's execution. Second, the W^3 Search Model uses partial data to decide what additional data should be collected to further isolate a performance bottleneck. The Dynamic Instrumentation collects and

Approach	**Volume of data**	**Accuracy**	**Intermediate results (temporal information)**
Event tracing	bad	good	good
Statistical sampling	good	bad	good
End of Execution Summaries	good	good	bad
Periodic Summaries	good	good	good

Figure 5.1. Sampled counter/timers vs. other types of instrumentation.

calculates the performance data; periodically the current data is reported. Varying the reporting rate affects only our rate of decision making and granularity of phase boundaries; it does not affect the accuracy of the underlying performance data.

Collected data is stored in a data structure called a time histogram[39]. A time histogram is a fixed-size array whose elements store values of a performance metric for successive time intervals. Two parameters determine the granularity of the data stored in time histograms: initial bucket width (time interval) and number of buckets. Both parameters are supplied by higher level consumers of our performance data. However, if the program runs longer than the initial bucket width times the number of buckets, we run out of buckets to store new data. In this case, we simply double the bucket width and merge adjacent pairs of samples. This process repeats each time we fill all the buckets.

By using a fixed size time histogram and changing the bucket width, we can also achieve efficiency in the volume of performance data transferred out of the application. When we change the bucket width, we also change sampling rate to correspond to the new bucket width. This decreases the frequency of sampling as the length of program execution increases. As a result, the volume of data transferred grows logarithmically in time, yet facilitates collecting detailed intermediate results.

We chose software based instrumentation because we felt that relying on custom hardware was both impractical and unnecessary. It is impractical due to the cost and time to construct it. More importantly, because of the ability Dynamic Instrumentation to adjust what data is collected, custom trace co-processors such as Multikron[69] or TMP[99] are no longer required. We believe the role of hardware based instrumentation should be to efficiently count those events which are not visible to software based instrumentation. For example, hardware counters are useful for measuring cache misses, floating point operations, and bus utilization.

By using simple counters and timers, we are able to easily integrate performance data from external sources. For example, most operating systems keep a variety of performance data internally; examples include statistics about I/O, virtual memory statistics, and CPU time. Usually, this information can be read by user processes. Also, a number of machines provide hardware based counters that are a source of useful performance information. For example, the Power2[97], Cray Y-MP[18], and Sequent Symmetry[92] systems provide detailed counters of processor events. We can combine external information with direct instrumentation to get precise information to relate the external events back to specific parts of the program. For example, if we have a way to read the cumulative number of page faults taken by a process, we can read this counter before and after a procedure call to compute the number of page faults taken by that procedure.

Using dynamically selected counters and timers provides an efficient data collection system that permits gathering the precise information that programmers need to understand their program's performance. In addition, data can be gathered from many different sources and combined.

5.2.3. Points, Primitives, and Predicates

To collect data, we need to insert instrumentation into the program. We developed a simple well defined set of operations that can used as building blocks to compute metrics for the desired resources. By keeping the instrumentation operations simple, we can optimize their performance for each platform. Recording performance information about the application program is accomplished by *points*, *primitives* and *predicates*. *Points* are well-defined locations in the application's code where instrumentation can be inserted. Currently the available points are procedure entry, procedure exit, and individual call statements. In the future, points will be extended to include basic blocks and individual statements. *Primitives* are simple operations that change the value of a counter or a timer. *Predicates* are boolean expressions that can be associated with primitives to determine if the associated primitive gets executed. By inserting predicates and primitives at the correct points in a program, a wide variety of metrics can be computed.

Our system consists of six primitives: set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. Predicates are simple conditional statements that consist of an expression and an action. If the value of the expression is non-zero, the associated action is taken. Expressions can contain numeric and relational operators. The operands of predicate expressions can be either counters or constants. If the point where the predicate is inserted is a procedure call or entry, the parameters to the procedure can be used as operands in expressions. If the predicate is inserted at a procedure exit, the procedure's return value can be used. Actions are either other predicates or calls to one of the six primitives. Figure 5.2 describes the predicate facility.

Examples of how primitives and predicates can be combined to create metrics is shown in Figure 5.3. This example shows two different metrics. The first example computes the number of times procedure `foo` is called. A single primitive to increment a counter by one has been inserted at the entry point to the procedure `foo`. The second example shows a metric to compute the number of bytes transferred via a message passing procedure. The second parameter to the add counter primitive is a multiplication expression that uses the third and fourth arguments

`if (expr) actn`	If `expr` is non-zero then execute `actn`.
`actn`	An `if` statement or a call to a primitive.
`operand`	A counter, constant, or parameter.
`expr`	`operand operator operand` *or* `operand`
`operator`	`+ - / * < > <= >= == != and or.`

Figure 5.2. Description of the Predicate Language.

Predicates are described in a simple language consisting of expressions and one conditional. All operands and their results are of type integer. The predicate language is a subset of the Metric Description Language described in Appendix B.

to the message passing function to compute the number of bytes transferred.

Figure 5.4 shows a slightly more complex example of Dynamic Instrumentation. Three instrumentation points are used to compute the number of times a message passing routine is called by the procedure `foo`. The top two primitives maintain a counter `fooFlag`, which is non-zero whenever the procedure `foo` is active. The last point increments the `msgsSent` variable. However, the increment only occurs when the counter `fooFlag` is non-zero.

5.3. Instrumentation Mechanism

In the previous section, we defined two abstractions: resources and metrics. To collect data, requests to enable metrics for specific resource combinations must be translated into calls to primitives and predicates at the appropriate points in the application processes. The implementation of our instrumentation system is divided into two parts. The first part, the Metric Manager, translates requests for metric and resource combinations into primitives and predicates. The second part, the Instrumentation Manager, modifies code sequences in the application

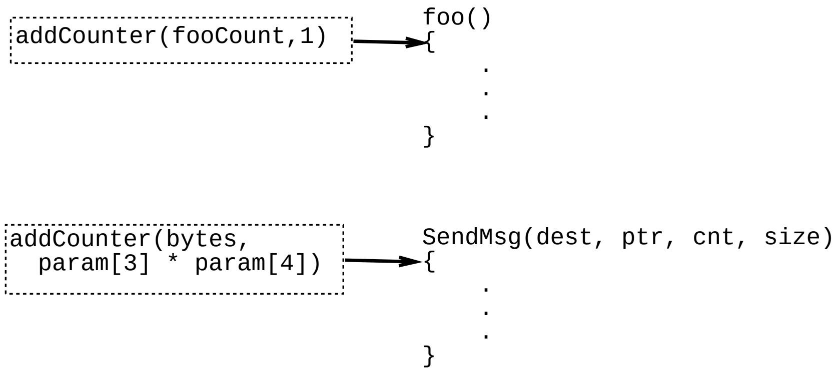

Figure 5.3. Example Showing Two Different Metrics.

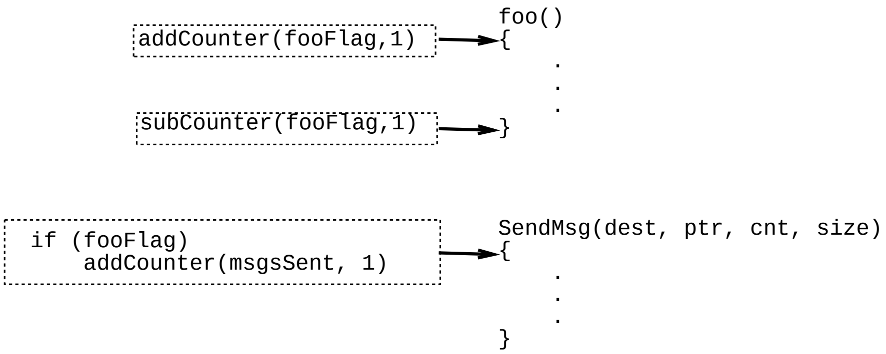

Figure 5.4. Sample Constrained Metric.

being monitored. Figure 5.5 shows the structure of the Dynamic Instrumentation mechanism.

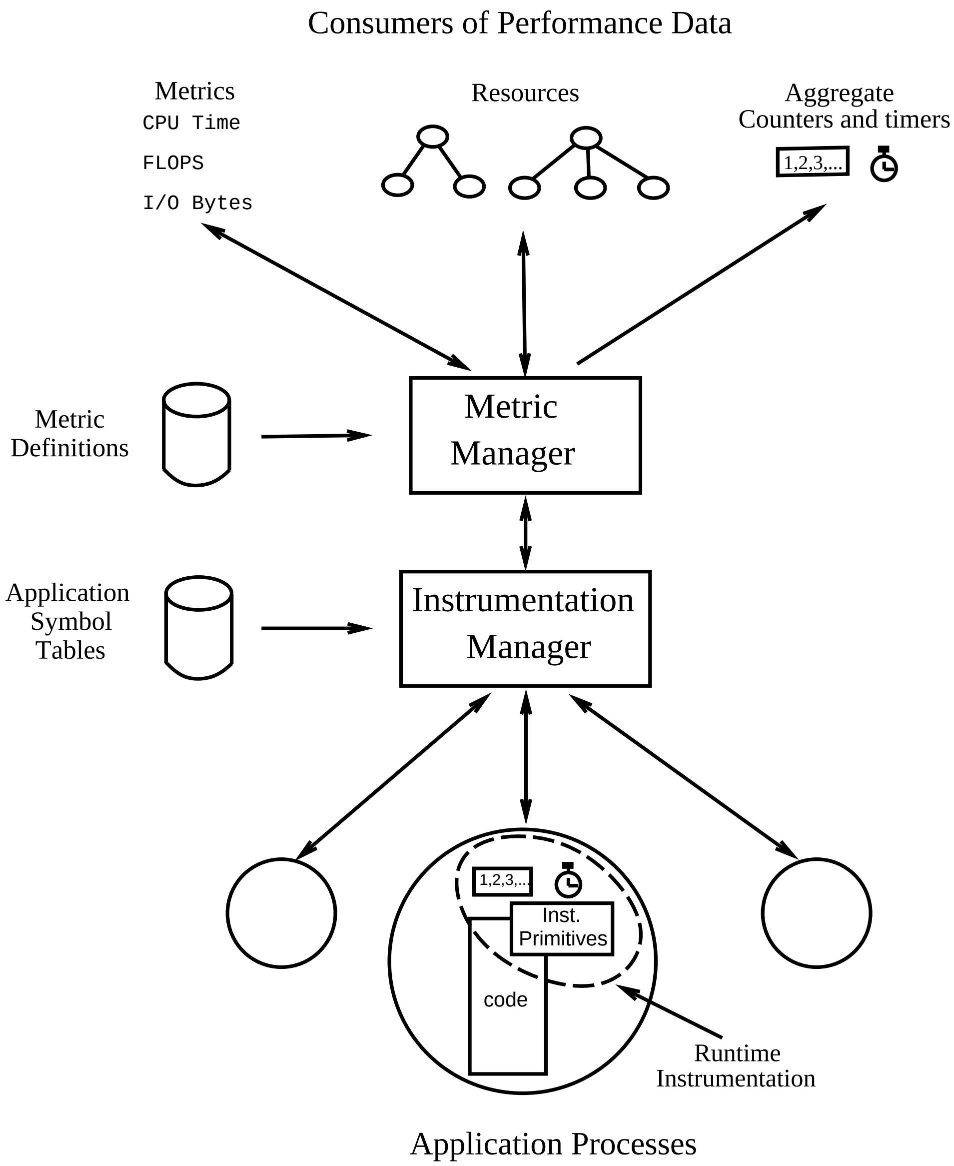

Figure 5.5. The structure of the Dynamic Instrumentation system.

The three items at the top of the figure (metrics, resource, and aggregate counters and timers) are the interface provided to higher level analysis tools. The metric manager uses the metric descriptions to translate requests for metrics and resources into primitives, and predicates at points. The Instrumentation Manager then inserts the instrumentation into the application program (shown at the bottom).

5.3.1. Metric Manager

The Metric Manager has two roles; it translates requests for metrics into primitives and predicates and informs higher level consumers about new resources. It is designed to be application and machine independent. The translation from metrics to primitives is described by *Metric Definitions*.

A metric definition can be viewed as a template that describes how to compute a metric for different resource combinations. It consists of a series of code fragments that create primitives and predicates to compute the desired metric. However, we need to be able to create each metric for any combination of resources. A naive approach would be to create a metric definition for each possible resource combination. However, this would require writing far too many metric definitions. We instead choose to divide metric definition into two parts: a base metric, and a series of resource constraints. The base metric defines how a metric is computed for an entire application (e.g., all procedures, processes, or processors). A resource constraint defines how to restrict the base metric to an instance of a resource in one of the resource hierarchies.

For example, consider what happens when a request is received to compute the number of synchronization operations invoked by a specific procedure in an application. First, we allocate a constraint counter that will be positive only when the desired procedure is active. Second, we insert instrumentation at the entry and exit of the desired procedure to update the counter. Finally, we generate instrumentation at the entry point to each of the synchronization procedures to check that the constraint counter is positive (i.e. the constraint is satisfied); if the constraint counter is positive, then the metric (counter) is incremented. The instrumentation that gets inserted in this case is the same as in the example shown in Figure 5.4.

Metric definitions are expressed in a metric description language that is interpreted when a request to start collecting a metric and resource combination is received from the higher layers. The result of interpreting the metrics description is to generate a series of instrumentation predicate and primitive requests that are inserted into the application program. When the application program continues execution, it will invoke the inserted instrumentation and compute the desired metric.

The metric description language is a simple language consisting of definitions of metrics and constraints. It also contains a list data type. One use of the list data type is to to enumerate possible instrumentation points. A complete description of the syntax of the metric description language appears in Appendix A. Metric definitions can be provided by higher level performance tools, although we provide a substantial library of them. Resource constraints are particular to a specific resource hierarchy, but can be used by different metrics. Some metric definitions and resource constraints are generic and apply to all platforms; others are specific to a platform or programming model.

A metric definition consists of several clauses to define the name, units, fold operator, and aggregation operator for the metric being defined. It includes a series of clauses that refer to constraints (described below). Each metric also includes a definition of a counter or timer that contains the metric value, and a series of requests to insert primitive operations at appropriate points in the program.

```
syncFuncs = { Barrier, Lock, Mutex };

metric syncOps {
    name "Synchronization Operations";
    units operationsPerSecond;
    foldOperator sum;
    aggregationOperator sum;
    constraint MSGTagPredicate;
    constraint processPredicate;
    constraint procedurePredicate;
    base is counter {
        foreach func in syncFuncs {
            append at func->entry constrained [
                addCounter(syncOps, 1);
            ]
        }
    }
}
```

Figure 5.6. A sample metric description.

Figure 5.6 shows a sample metric definition to compute the number of synchronization operations performed by a program. It consists of a list to define the synchronization procedures for this programming model, and the metric definition. The first line of the metric definition declares the name of the metric as `Synchronization Operations`. The second line defines the units for this metric (`operationsPerSecond`). The third line defines the fold operator for the metric (`sum`) which is used to determine how adjacent samples of the metric should be combined when the time histogram folds (The two options are sum and average). The fourth line defines the aggregation operator (`sum`) which is used to aggregate multiple instances of the metric from different processes (or threads). The next three lines define the constraints that can be applied to this metric: in this case, message tag (`MSGTagPredicate`), process (`processPredicate`), and procedure (`procedurePredicate`). The rest of the metric definition is for the base metric. The base definition is a counter (implicitly declared the same name as the metric, `syncOps`). The `foreach` clause iterates through each of the procedures of the list `syncFuncs` and inserts at call to `addCounter` for the counter `syncOps` at the entry point to each procedures. Instrumentation to be inserted into the application is enclosed in pairs of `[` and `]`.

A resource constraint defines a counter (implicitly declared to be the same name as the constraint) whose value is positive only when the desired resource (implicitly declared as `$constraint`) is active. This constraint is implemented by inserting primitive calls to increment and decrement the counter as the status of the resource changes. This counter is used in a predicate that guards the execution each of the primitive operations in the base metric. To compute a metric constrained on two different resource hierarchies (e.g., procedure A in process P), each constraint defines a counter and a conjunctive expression of these counters is used as a predicate on the base metric's primitives.

Figure 5.7 shows a typical constraint clause. This clause defines a counter to be positive if a selected procedure from the procedure resource hierarchy is active. The `foreach` clause inserts statements before and after each subroutine call in the selected procedure to clear or set the constraint counter. The last two statements insert an assignment to the constraint counter at the entry and return points from the selected procedure.

Currently, metric definitions are written in our metric definition language, hand translated into the C programming language, and compiled into the Dynamic Instrumentation controller process. However, we are implementing an interpretor that will read the metric description language, and evaluate metric definitions when instrumentation requests arrive.

The second task of the metric manager is resource discovery. Resource discovery is the process of building the resource hierarchies (i.e., the ‘‘where’’ axis). Much of the resource discovery is done when the application and machine(s) to be used for the measurement session are specified. For example, at this point we know all of the procedures that might be called, and what types of synchronization libraries are linked into the application. However, some aspects of resource discovery must be deferred until the program is executing. For example, building a resource hierarchy of files read or written must be done when the files are first accessed. To collect this runtime resource information, instrumentation is inserted into the application program. For example, we insert instrumentation (using the same technique as normal instrumentation) to record the file names on open requests.

5.3.2. Instrumentation Generation

Our approach of inserting data collection during program execution means that we must be able to generate the instrumentation code while the program is in execution. We then modify the program's execution image to incorporate the desired instrumentation. A number of technical hurdles must be overcome before it is possible to

```
constraint procedurePredicate /Procedure is counter {
    foreach i in $constraint->calls {
        append at i->preCall [
          setCounter(procedurePredicate, 0);
      ]
        prepend at i->postCall [
          setCounter(procedurePredicate, 1);
      ]
    }
    append at $constraint->entry [
      setCounter(procedurePredicate, 1);
    ]
    prepend at $constraint->return [
      setCounter(procedurePredicate, 0);
    ]
}
```

Figure 5.7. A sample constraint clause.

use runtime code generation. First, we must develop a code generation system that is efficient enough to use during program execution; we achieved this by using simple instrumentation primitives. Second, we need to be able to modify a program to insert the instrumentation, yet ensure the correctness of the original program. Inserting the instrumentation into the program is accomplished by a simple runtime instrumentation compiler, and a mechanism called *trampolines*.

Generating code during program execution is not a new idea. A number of correctness debuggers have been built that modify an executing program for assertion checking and conditional breakpoints. Brown[15] developed a debugger for Cray Computers that provided this feature. Kessler at Xerox Parc[47] built a system that used dynamic modification of a program to insert breakpoints. Work has also been done by Wahbe, et. al. on fast data breakpoints[95]. They employ sophisticated program analysis techniques to minimize the overhead of instrumentation for data breakpoints. Massalin and Pu[62] have a novel application of code patch-up in their Synthesis operating system. They modify a program to automatically schedule another thread when it is about to block. Bishop[11] developed a performance monitor that patches a binary program for performance monitoring. However, his patches inserted breakpoint instructions, and required multiple context switches per instrumentation point (with a considerable performance penalty). Our application of runtime code generation to performance instrumentation is new.

Runtime code generation is accomplished by the Instrumentation Manager that performs two functions: it identifies the potential instrumentation points, and inserts primitives and predicates into the application program. Potential instrumentation points are discovered by scanning the application (or applications) binary image. Scanning of the executable is done when the process to monitor is specified. Once the application starts to execute, the Instrumentation Manager waits for requests from the Metric Manager. When a request arrives, the Instrumentation Manager translates it into small code fragments, called *trampolines*, and inserts them into the program. We define two types of trampolines: base trampolines and mini-trampolines (see Figure 5.8). There is one base trampoline per point with active instrumentation. Base trampolines have slots for calling mini-trampolines both before and after the relocated instruction. One slot before and one slot after the instruction are used to call global primitives (ones that are inserted into all processes in an application). The other slots are used to call local primitives (ones that are specific only to this process). This structure makes it easier to use hardware broadcast facilities to install global requests.

Identifying the points where instrumentation can be inserted is accomplished by analyzing the instructions in the application. This is one area where compiler writers could help make our task easier. It would be helpful if additional symbol table information were available that indicated exactly what is code and what is data in a program. Many compilers place read-only data into a program’s text (code) segment. This creates problems for post-linker tools (correctness debuggers and performance tools). Larus and Ball[4] have also noted this problem, and developed heuristics to differentiate code from data. We use many of their heuristics in our implementation. A better solution was used by Sardent[45]. They added additional information to their executable format to facilitate post-linker tools.

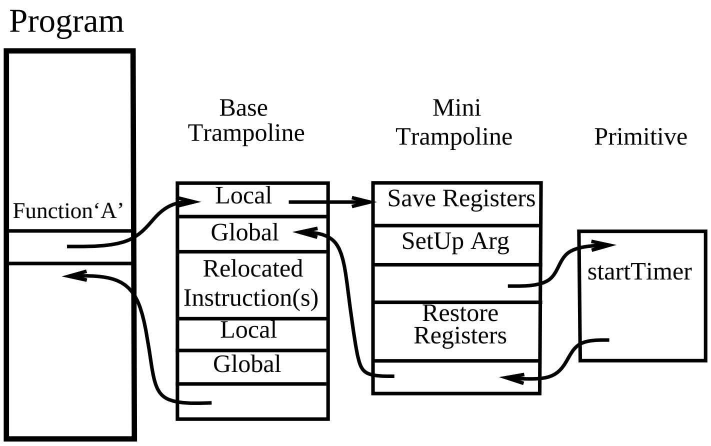

Figure 5.8. Inserting Instrumentation into a Program.

The base trampoline provides slots to call instrumentation before and after the relocated instruction. This example shows one mini-trampoline at the instrumentation point; however there could be multiple mini-trampolines present if multiple instrumentation primitives needed to be called at this point.

Mini-trampolines contain the code to evaluate a specific predicate, or invoke a single primitive. There is one mini-trampoline for each primitive or predicate at each point. A sample instrumentation point with trampolines installed appears in Figure 5.8. Creating a mini-trampoline requires generating appropriate machine instructions for the primitives and predicates requested by the Metric Manager. The predicate language is simple, requiring only a handful of instruction types. The instructions are assembled by the Instrumentation Manager, and then transferred to the application process using a variation of the UNIX ptrace interface. In addition to the code requested by the Metric Manager, code must also be generated to save and restore any registers that the generated code (or called primitives) might overwrite. When we extend the available points to include basic blocks and individual statements, additional processor state (such as condition codes) must also be saved and restored. Figure 5.9 shows two sample mini-trampolines. The first example (a) shows a simple counter increment that has been generated as inline code. This is the code that would be generated for the first primitive in Figure 5.3. The second example (b) shows the instrumentation generated for the predicate and primitive at the entry point to the `SendMsg` procedure in Figure 5.3. The `if` statement has been translated into a branch based on the value of the `fooFlag` counter. If the counter is non-zero, the `stopTimer` predicate is called from the mini-trampoline.

To use the performance data we collect, we must move it off the parallel machine to some other location where it can be analyzed. However, if performance data moves through the parallel machine during program execution, it can compete for resources with the program under study, and thus perturb it. We take advantage of the CM-5 synchronous gang scheduling to avoid this problem. At the end of a global scheduling quanta, all of the nodes context switch to the next job. We arrange to collect our intermediate performance data at this point. Since the program has already been stopped by the operating system and its execution context has been saved, the program execution state is not affected and data transfer does not contend with the program for machine resources.

```
save
load fooFlg, o1
add o1,1,o1
store o1, fooFlg
restore
ba,a baseTramp1+4
```

(a)

```
save
load fooFlg, o1
cmp o1, 0
be +8
load #msgTme, o0
call startTimer
restore
ba,a baseTramp3+4
```

(b)

Figure 5.9. Two Sample Mini-Trampolines.

This technique will cause some dilation of wall-clock execution time, but the data transport itself has virtually no perturbing effect on the program's execution. Waiting to the end of a quanta to transfer data off the nodes works unless the data buffer fills up on the nodes; in this case, it is necessary to either force the timeslice to end early, or wait for it to end. Either option has a perturbing effect. On machines without a synchronous scheduling model, such as the Intel Paragon[44], we use this model of data transport but it is more difficult to factor out the cost.

The Instrumentation Manager may make frequent changes to the instrumentation during the application's execution. To be scalable, the cost of instrumentation operations must not grow with the number of nodes in the parallel computer. When we insert instrumentation into all instances of a procedure on all nodes, the cost should be the same as inserting instrumentation into a single node. To achieve constant time instrumentation, we need a broadcast based version of ptrace to insert global instrumentation into all nodes in unit time. Fortunately the current version of the CM-5 operating system provides a version of ptrace that runs on the nodes, and this permits us to build a broadcast based version of ptrace as a CM-5 application that runs interleaved with the measured application.

On machines that do not provide hardware broadcast, such at the IBM SP/2[91], we can construct a software message spanning tree. The spanning tree technique achieves logarithmic time, instead of unit time cost. However, most new machines come with some form of broadcast facility, including the Intel Paragon (though it is currently not accessible to application software), Cray T3D[19], and Meiko CS-2[41]. Efficient operations such as these are crucial to dealing with the issue of scale in tools for large parallel machines.

5.4. Clock Synchronization

An important question for many parallel tools is how to synchronize the timestamps from different processors. A full solution to this problem requires the use of complex algorithms such as logical clocks[50]. However, since our instrumentation is based on counters and timers that are located in individual processors, global time is not important to dynamic instrumentation. Our main concern is how to combine samples from different sources (processors, processes and threads) to form aggregate values. We need to combine data from different sources whenever a resource combination spans more than one address space.

We depend on the clocks of each processor to provide timestamps for the samples; however, the granularity of sampling is relatively infrequent compared to clock drift. Typically, we sample metrics at the rate of 1-2 times per second, but clock synchronization algorithms such as NTP[66, 67] can keep clock drift to a few milli-seconds. As a result, clock drift causes a bit of uncertainty about the end of our time intervals, but the fraction of a sample interval affected is small.

Aggregation is complicated because sampling from different sources is (potentially) asynchronous. Not all samples arrive at the same time, nor do they necessarily cover the same time interval. We need to align samples to combine the data from each source for each interval of time. To do this, we developed an algorithm to forward data only when it is received from all component sources for a specific time interval. The algorithm keeps track of the earliest ending time of the most recent sample from any source (called the *current time*). Source we do not yet have data from have an ending time of zero. When a new sample arrives, we re-compute the current time. If the new sample advances the current time, an aggregate sample for the time interval from the previous value of the current time to the new value of the current time is generated. An aggregate sample's boundaries are two consecutive values of the current time, and current time is computed by looking all of the sources. As a result, the time interval of the aggregate sample may not be the same as *any* of the constituent samples' interval.

Figure 5.10 shows a series of four samples arriving from three different sources. In this figure, the dashed box indicates the new sample that just arrived; the value inside each box is the value of that sample. In the first picture (a), a sample arrives for source 1 and since there is no data for the other sources, it is simply stored. Likewise, when the second sample arrives (b), we do not have any data for the third source yet and the sample is stored. When the third sample arrives, we now have data from all of the sources and an aggregate sample is generated. The value of the aggregate sample is the sum of the weighted share of the most recent sample from each source. (We used sum as the aggregation operator in this case, but as noted in section 5.3.1 the aggregation operator is specified as part of the metric description.) The number in bold shown below each sample is the component of that sample included in the aggregate value. The aggregate sample in this case is for the time interval from 2 to 3 and has the value of 13. When the fourth sample arrives (d), the current time is not advanced, so the new sample is combined with previous sample for that source. In this case, we combine the samples for source two to be a single sample with the value of 28 for the interval from 2 to 6.

5.5. Conclusion

In this chapter, we have described a new approach to data collection called Dynamic Instrumentation that permits customized instrumentation to be inserted at runtime. We outlined how to implement Dynamic Instrumentation to ensure the correctness of the original program. We also introduced a new hybrid of sampling and tracing that permits efficient collection of intermediate (temporal) values of counters and timers without having to use full tracing. In the next chapter, we describe a prototype implementation of dynamic instrumentation. We also evaluate the performance of the prototype at the granularity of each primitive, and at the granularity of several applications.

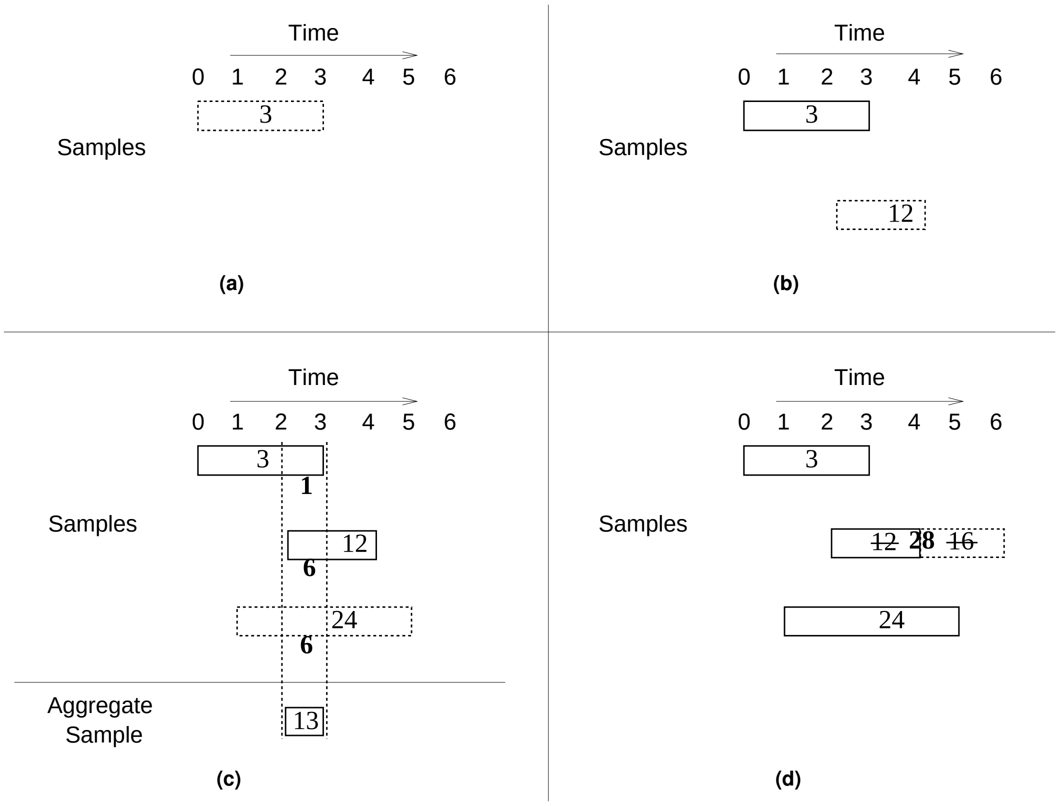

Figure 5.10. Aggregating Samples from Different Sources.

This figure shows the arrival of four successive samples from three different sources. Each picture shows the arrival of one sample with the new sample shown as a dashed rectangle. In the first two cases, no aggregate sample is generated. In the third case, the new sample completes the set of processes, advances the current time, and so an aggregate sample is generated. In the final case, the sample does not advance time, but results in a new coalesced sample for the middle source.

Chapter 6

IMPLEMENTING DYNAMIC INSTRUMENTATION

We decided to study the performance of our Dynamic Instrumentation at two levels. First, we report on the cost of the individual operations, accounting for cost down to exact number of instruction cycles. These costs provide insight into the abilities and limitations of our techniques, and insights to operating system designers about what facilities to provide. Next, we provide macro benchmarks for a few examples of the overall cost of using these facilities. The macro results are based on a few simple workloads and are meant only to be a general guideline (i.e., your mileage may vary).

The initial implementation of Dynamic Instrumentation is on a Thinking Machine CM-5 parallel computer. To explain the performance of our instrumentation, it is necessary to know a bit about the implementation of the CM-5. The CM-5 is a distributed memory multiprocessor containing nodes connected together by a network that provides both point-to-point and broadcast communication. The machine is controlled by a front end processor running a modified version of the SunOS operating system. Processor nodes consist of SPARC processors running at 33Mhz, memory, and a network interface chip. Each node has a virtual 64 bit free running clock.

6.1. Micro Benchmarks

Our micro benchmarks show the cost of the individual components of the Dynamic Instrumentation. We first report the cost of basic instrumentation, including cost of executing the trampolines, cost of the primitives and predicates, and a detailed break down of several of the primitive operations. Next, we report on the cost of the instrumentation support functions, including the instrumentation generation, instrumentation installation and performance data transport. These detailed results provide insights into the limitations of various instrumentation techniques. It also provides guidance to operating system designers about what facilities are needed to support efficient instrumentation.

The first aspect of the performance of our instrumentation system we studied was the overhead of the trampolines that get inserted when a primitive is to be called. Figure 6.1 shows the cost executing the trampolines. The time required for a base trampoline is 4 clock cycles; the machine has a 33Mhz clock so one cycle is about 30 nano-seconds. This time provides four no-op slots to call the mini-trampolines. The second row shows the time required for the smallest mini-trampoline. Two clock cycles are required for the SPARC save and restore register instructions (for case when no register window overflow occurs). Another source of delay in the trampolines (8 clock cycles long) is for the branches into and out of the trampolines. The final component of the trampoline overhead is the time required to setup the parameters to call the predicate. This time ranges from 1 to 5 clock cycles depending on the type of the parameter.

Item	Clock Cycles
Base Trampoline (slots for branches)	4
Mini Trampoline (save/restore registers)	2
Branches	8
Parameters	1-5

Figure 6.1. Time to execute trampolines.

This table shows the time (in clock cycles) to execute both types of trampolines, the time to branch into and out of trampolines, and the time to setup the parameters to a primitive.

The other intrinsic cost of our instrumentation is the time to execute our primitives. Figure 6.2 shows each of our six primitives and their execution time in clock cycles. In addition, the time to execute a simple predicate that checks if a counter is positive is included. The measured times ranged from 7 clock cycles for the predicate to 71 clock cycles to stop a process timer. These results were collected by invoking each primitive 1,000 times in a tight loop, and recording the elapsed time. For each call, the counter or timer passed to the primitive was the same, so the effects of the machine's memory hierarchy has been factored out of these results. To get a feel for the relative cost of these numbers, we compared the cost of our primitives to simple procedure calls on the machine. A one-argument procedure call that returned its argument took 15 clock cycles.

To look for bottlenecks in our timers, we isolated the time required to execute each part of each timer primitive. Figure 6.3 shows each of the four timer primitives and 5 different operations used to implement the timer. The

Primitive	Clock Cycles
addCounter	8
subCounter	8
startTimer(wall)	32
stopTimer(wall)	55
startTimer(process)	37
stopTimer(process)	71
if (counter > 0)	7

Figure 6.2. Cost of Primitive Operations.

The cost (in clock cycles) of executing the instrumentation primitives, and a simple predicate on a TMC CM-5.

first row, ‘‘read clock’’, is the time to get a 64 bit representation of the time (either wall or process time) into two 32 bit registers. This involves reading the 32 bit free running Network Interface clock (7 clock cycles) plus reading the high order 32 bits from memory. Since the low-order word could overflow while we read the high-order word, we need to check for this condition and re-read the clock if it overflowed. As a result, it took 20 clock cycles to read the wall clock, and 25 clock cycles to read process time (Process time requires an additional load and subtraction to factor out time when the process was not running.) The semantics of our timers permit multiple call to `startTimer` to be made and a matching number of `stopTimer` calls are required to stop the timer. The time to provide this abstraction appears in the row ‘‘maintain counter’’. The third component of our timer primitives is ‘‘store value’’ which is the time to record either when the timer started or the elapsed time when it stopped. In addition, stopping a timer involves two extra steps. First, we need to compute the elapsed time since the timer was started (shown in the ‘‘compute elapsed’’ row). Second, since timers can be read by an asynchronous sampling function, we need to ensure that the value of the timer is consistent even when we are stopping it. (This is not a problem when starting the timer because we compute and store the start time before marking the counter as running.) Permitting sampling at any time is accomplished by storing a snapshot of the timer in an extra field while the other fields are being updated. The overhead introduced to maintain this field is shown in the row labeled ‘‘consistency check’’.

Our breakdown of the cost of timer operations shows that the largest time component in any primitive (and the majority of time in two cases) is spent reading the clock. This time could be substantially reduced if processors provided a clock that was readable by user level code at register access speed. Currently, few RISC processors, with the exception of the DEC Alpha[85], provide such a clock. Accurate cheap-to-read clocks are a critical architectural feature to permit building efficient performance tools.

	Time (clock cycles)			
	Wall		**Process**	
Operation	**start**	**stop**	**start**	**stop**
read clock	20	20	25	25
maintain counter	8	10	8	11
store value	4	4	4	4
compute elapsed		10		10
consistency check		11		21

Figure 6.3. Primitive times by component.

The numbers presented above reflect the time to execute our primitives after a slight modification to the CM-5 operating system was made. Originally, the operating system required a system call to read the clock, and the overhead of this system call dominated the cost of our timer primitives. Figure 6.4 shows the time to execute our four timer primitives both with and without the mapped clock. Using these system calls slowed our timer primitives by a factor ranging from 2 to 9. We overcame this bottleneck by adding a new system call to the operating system that maps the kernel's clock data structure into user address space so that we could read the clock at memory speed. Another option to implement our timers would be to use the timer library routines supplied by Thinking Machines. The results of using their timer calls also appears in Figure 6.4. Our timers are a factor of 6 to 12 times faster than the vendor-supplied ones. However, the Thinking Machine timers provide both a process and a wall timer in one. While this may be useful for some applications, for Dynamic Instrumentation we only need one timer type at a time. It is vitally important when building performance tools that accessing clocks be as fast as possible. Requiring a kernel call to read a clock is unacceptably slow. Hardware support alone is not sufficient. Operating system designers also need to provide access to the hardware clocks without requiring a system call.

The overhead of trampolines is one place where we could improve our implementation. For example, if we generated the base and mini-trampolines as a straight line code sequence we could reduce the overhead by 7 clock cycles (4 for the no-op slots, and 3 for not having to branch to the mini-trampoline). This would require additional complexity in the instrumentation manager (especially when one primitive is removed from a point with several primitives). However, 7 clock cycles is a significant fraction of the time for many of our primitives.

A potential source of overhead in Dynamic Instrumentation is the mechanism to process instrumentation requests. Unlike traditional compilers and code generators that run before the program executes, in Dynamic Instrumentation code generation is interleaved with program execution. Instrumentation request processing consists of generating the required instrumentation instructions, transporting them from the instrumentation manager to the application, and installing them. Instrumentation generation is done by the metric and instrumentation controllers described in Chapter 5. To transport the instrumentation, we developed a broadcast based version of the UNIX

Primitive	**Default TMC Timers**	**Dyninst**	
		OS trap	**Mapped Clock**
start(Wall)	410	109	32
stop(Wall)	410	133	55
start(Process)	410	320	37
stop(Process)	410	358	71

Figure 6.4. Timer overheads on a TMC CM-5.

ptrace command. Figure 6.5 shows the steps in moving an block of instrumentation from the instrumentation manager to the nodes on a CM-5.

We now report the overheads associated with the components of instrumentation generation. The runtime code generator can generate about 800 mini-trampoline per second. The number of mini-trampolines per instrumentation request varies greatly with the type of request, and so there is no typical cost per instrumentation request. The top half of Figure 6.6 shows the time required to insert the generated instrumentation into an application. The implementation of a scalable ptrace includes two components. First, data is copied via the CM-5 broadcast network to the nodes. There is a fixed per operation cost of 31 microseconds associated with this copy to synchronize and setup the broadcast network. There is also a per byte cost of 0.49 microseconds to copy data to the nodes over the broadcast network. Second, the data is transfered via a variation of the UNIX ptrace command from the "controlling" process to the application process. For the ptrace step, the fixed cost is 34 microseconds which is due to the time to make a kernel call on the node, and check the ptrace parameters. The ptrace per-byte cost is 4.3 microseconds; the reason this value is so high is that each word copied generates a cache flush instruction to ensure the instruction cache is consistent.

We also measured the performance of the data transport layer, which moves performance data from the nodes to the front end processor. The data path for the transport is the reverse of that shown in Figure 6.5. The data transport is implemented using a similar mechanism as the scalable ptrace. The one difference is that rather than using the CM-5 broadcast network to return the data from the nodes, the data network is used. The performance for the data transport facility is shown in the lower half of Figure 6.6. The per operation overhead, 200 microseconds, is much higher for the CM-5 network case due to the more complex protocol employed by CMMD for the data

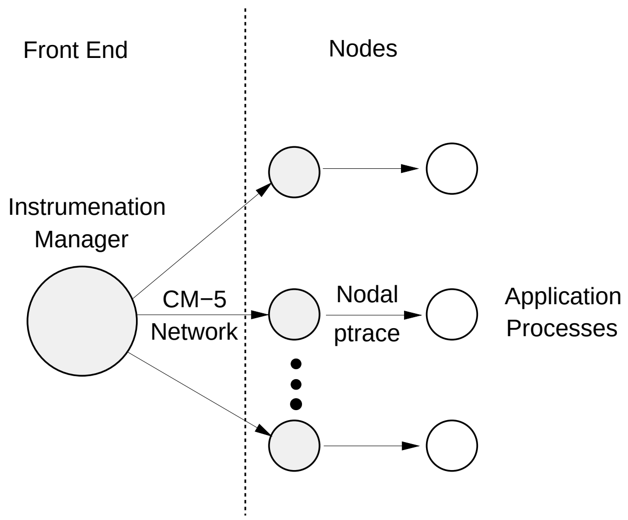

Figure 6.5. Communication Paths for Scalable Ptrace.

Cost Component	Time (in microseconds) CM-5 Network	Nodal Ptrace
Instrumentation Insert		
Per-operation Cost	31	34
Per-byte Cost	0.49	4.30
Data Transport		
Per-operation Cost	200	194
Per-byte Cost	0.34	0.06

Figure 6.6. Overhead of inserting instrumentation and transporting data.

The total time for a single move request, in either direction, is the sum of the fixed network and ptrace costs plus the number of bytes transfered times the the sum of the network and ptrace per-byte costs.

network. However, the per-byte cost of 0.34 microseconds is lower due to the higher bandwidth of the data network. For the ptrace component of the data transport, the per operation cost of 194 microseconds is higher than the instrumentation insertion case because additional requests need to be made to read the base and bounds of the data buffer to be read from the application. However, the per-byte cost is 0.06 microseconds. This is much better than the ptrace write performance since the cache does not need to be flushed on a read operation.

The front end processor or the transport mechanism can become a bottleneck if too much data needs to pass through them. To reduce the amount of data sent through the transport mechanism, we use the hardware combining facilities of the CM-5 to aggregate metrics that have components on each node. This approach ensures that the data required for aggregate metrics remains constant regardless of the machine size.

6.2. Macro Benchmarks

The micro benchmarks were designed to study the performance of the primitive operations in our instrumentation system. To get a feel for the larger picture, we also studied the performance of our instrumentation system for entire applications. The goal of these benchmarks was to study the aggregate impact of our instrumentation system for several real applications. We conducted two type of benchmarks. First, we evaluated how our system performed with a tool that makes multiple requests to change instrumentation. Second, we compared the overhead of Dynamic Instrumentation to existing tools.

For the dynamic tests, we ran two parallel applications on the CM-5. The first application does a domain-decomposition method for optimizing large-scale linear models. The second program is a database simulator that implements a parallel Grace Hash join algorithm in a simulated shared-nothing environment[79]. Both programs are written in C. We compared the execution time of an un-instrumented application to the execution time of an instrumented application. We measured the overall cost of our instrumentation and the overhead of various components of the instrumentation. For this study, we used the Performance Consultant, an implementation of the W^3 Search Model, a system that takes full advantage of Dynamic Instrumentation, to drive our instrumentation.

The results for these tests are shown in Figure 6.7. The second column, Original Execution Time, shows the time to run the application without any instrumentation. One program ran for about an hour, and the other one for a couple of minutes. The third column, Execution Time w/Instrumentation, shows the total execution time with Dynamic Instrumentation. To understand where this time is spent, we have divided the overhead for instrumentation into three categories: Null Handler time, Actual Handler time, and Direct Perturbation. The first two types of overhead represent the time spent in the end-of-timeslice handlers. Null Handler time is the cost of running the end-of-timeslice code without any instrumentation enabled. The Actual Handler time is the time to process ptrace requests and collect data from the nodes. The differences between the two applications for Actual Handler time is due to the larger volume of performance data transferred for the database application. Since the CM-5 uses synchronous gang scheduling, overhead time in the handlers does not directly perturb the performance of the program. Instead, it dilates execution similar to the way that other time-shared jobs impact an application's execution. The third type of overhead represents the direct perturbation of the application due to the instrumentation inserted. It is a modest amount (6 and 8%) for these two applications, since the Performance Consultant requests only the instrumentation it requires.

For the second set of tests, we selected several sequential applications and compared the overhead of our system to two UNIX profilers: prof, and gprof. This made it possible to benchmark our instrumentation cost compared to existing techniques. Using the Dynamic Instrumentation, we enabled the CPU time metric for each procedure in the program. While this does not take advantage of Dynamic Instrumentation's ability to adapt data collection, it still provides the benefits of an attachable performance tool and does not require the application to be recompiled. In addition, since Dynamic Instrumentation collects intermediate values, it is possible to study the time varying performance of an application which is not possible with traditional prof. We also compiled and ran each program with

Program	**Execution Time**		**Components of Perturbation**		
	Orig.	**with Inst.**	**Null Handlers**	**Actual Handlers**	**Direct**
Decomp	59:22	65:29 (10%)	01:26 (2%)	00:06 (< 1%)	04:34 (8%)
Hash-join	01:22	02:09 (57%)	00:10 (12%)	00:32 (39%)	00:05 (6%)

Figure 6.7. Cost of ptrace operations.

This table shows the overhead of using Dynamic Instrumentation with the Performance Consultant for two applications running on the CM-5. All times in minutes:seconds.

prof and gprof profiling enabled.

The results of running Dynamic Instrumentation, prof, and gprof on the two sequential applications is summarized in Figure 6.8. The first application is MultiComm which is a math programming application that solves a multi-commodity network flow problem with mutual capacity constraints. It is written in a mixture of C and Fortran. For this application, Dynamic Instrumentation has only 4% overhead, but prof and gprof have overheads of 40% and 79% respectively. Prof and gprof instrument the internals of the C and math libraries; the default for Dynamic Instrumentation is to not instrument library routines. Since this application makes heavy use of integer divide which is a library function on most SPARC machines, prof and Gprof have higher overheads.

The second application is Tycho, a cache simulator. It is written in C and spends most of its time repeatedly calling 9 small procedures. For this application, Dynamic Instrumentation has a very high overhead of 70% compared to prof and gprof which have overheads of 14% and 44%. Dynamic Instrumentation has a higher per-procedure overhead because it needs to invoke a mini-trampoline and start and stop a timer for each procedure called. In contrast, prof only increments a counter (using periodic sampling to approximate CPU time per procedure) and gprof needs to identify the caller/callee relationship at each procedure.

The sequential macro benchmarks indicate that Dynamic Instrumentation has a higher per-operation overhead than traditional UNIX profiling. This is not surprising since Dynamic Instrumentation uses direct timing instead of sampling. However, the overhead seen for all three techniques when the procedure call frequency is high indicates that blindly instrumenting procedures is not necessarily the right granularity for data collection. The flexibility afforded by Dynamic Instrumentation to instrument at different levels than just procedures (e.g. modules and loops) makes it possible to collect data at an appropriate granularity for each application.

6.3. Conclusion

We have described the performance of an implementation of Dynamic Instrumentation. The current implementation runs on a Thinking Machine CM-5 using explicit message passing programs written in C, C++, or Fortran. The cost of the primitive operations is only a few instruction times, and overall application perturbation was under 10% for several real applications tested. Dynamic Instrumentation still uses software based probes that can

Application	**Original Time**	**Overhead (minutes:seconds)**		
		Dynamic Inst.	**Prof**	**Gprof**
MultiComm	01:12	00:02 (4%)	00:29 (40%)	00:56 (79%)
Tycho	01:57	01:22 (70%)	00:16 (14%)	00:32 (44%)

Figure 6.8. Overhead of different Sequential CPU Profilers.

perturb the execution of the application program. In the next chapter we describe a cost model to characterize the impact of this perturbation.

Chapter 7

INSTRUMENTATION COST MODEL

Our strategy of enabling instrumentation only when it is needed greatly reduces the amount of data collected, and therefore the perturbation caused by the instrumentation system. However, instrumentation requests still have an impact on the program's performance. To manage the perturbation caused by our instrumentation, we have developed an instrumentation cost system to ensure that data collection and analysis does not excessively alter the performance of the application being studied. The model associates a cost with each different resource used. Resources include processors, interconnection networks, disks, and data analysis workstations. The cost system is divided into two parts: predicted cost and observed cost. Predicted cost is computed when an instrumentation request is received, and observed cost while the instrumentation is enabled.

In this chapter, we describe the cost models and how the W^3 Search Model can use them to control the impact of instrumentation. We also describe an initial implementation of both parts of the cost model, and present results that show how closely it predicts the actual perturbation for several real applications. In essence, the Predicted Cost Model is just another performance metric and the techniques for computing performance metrics described in Chapter 5, and the W^3 Search Model, described in Chapter 3 are used to implement much of the model. In our initial version of the Predicted Cost Model, we only consider the direct CPU cost of instrumentation. This is a reasonable approximation on machines such as the CM-5, since our use of other resources (e.g. the interconnect) is during periods when the application has been stopped.

7.1. Predicted Cost

With Dynamic Instrumentation, data collection overhead no longer remains fixed for the entire program's execution. Each time a new request for instrumentation is received, the instrumentation overhead is (potentially) changed. In addition, different types of instrumentation requests can have decidedly different effects on a program's performance. Knowing the expected effect of an instrumentation request provides performance tools and programmers the opportunity to decide if a particular instrumentation request is worth the expected cost. We have developed the *Predicted Cost Model*, as part of Dynamic Instrumentation, to assess the overhead of each instrumentation request. This information is used by the Performance Consultant to control the amount of instrumentation that is enabled for a particular program. In this section, we describe the Predicted Cost Model, and how performance tools can take advantage of it.

Predicted cost is the expected overhead of collecting the data necessary to compute a metric for a particular focus (combination of resources). We compute the predicted cost when an instrumentation request arrives, but before the instrumentation is inserted into the application. The predicted cost is expressed as the percentage utilization of each resource in the system required to collect the desired data.

An important question is what resources should be included in the model. Should individual copies of the same resource be counted separately? For example, should each CPU be considered a resource, or should we group all processors together? Currently, we track the cost for each instance of a resource, and use the maximum predicted cost for any instance of that resource as the value to compare to the threshold. As we gain experience with the Predicted Cost Model, we will evaluate these choices.

To make the model more concrete, consider how to compute the expected CPU perturbation. In Dynamic Instrumentation, CPU time perturbation is due to the insertion of instrumentation primitives at various points in the program's executable image. To predict CPU time perturbation at a single point in the program, we need several pieces of information. First, we need to know what instrumentation will be inserted at the point. Second, we need to know the cost of executing that instrumentation. Third, we need to know the frequency of execution of that point. Figure 7.1 shows how the predicted instrumentation cost is computed. Given this information, we can multiply the overhead of the predicates and primitives at each point by the point's expected execution frequency to compute the predicted perturbation. The sum of this information for all points is the predicted cost for an instrumentation request. Metric definitions are used to enumerate what instrumentation primitives and predicates need to be inserted and where. Based on the experiments conducted in Chapter 6, we know the precise cost of each instrumentation primitive and trampoline request. The difficult part is estimating the frequency of execution of each point.

Our data about the execution frequency of points comes from a static model of procedure call frequency. We associate with every point in the program an expected frequency. The value of each point is static and based on point's type. Currently, the model has three types of points: system calls, message passing routines, and normal procedure calls. Part of the effort required to port Dynamic Instrumentation is to define the value of each of these three constants. Since the predicted cost is used in conjunction with the observed cost, it is not critical that the values of these estimates be perfect. Effectively, as the application executes, the cost information is updated based on the actual values of the application being monitored.

Computing the predicted cost is only part of the story. Of equal importance is how we use this information. The fundamental question is how much perturbation can an application tolerate. Different applications can tolerate different amounts of perturbation before the instrumented program no longer is representative of the original. In addition, depending on the desired accuracy (e.g., a coarse measurement session vs. final tuning), programmers may be willing to tolerate more or less perturbation of their application. The best way to accommodate these varying needs is to let the programmer control the amount of perturbation the tool inflicts on the application. We have developed a technique that lets the programmer, during a measurement session, set the tolerable perturbation of the application for each system resource. We use these thresholds to moderate how much instrumentation gets inserted into the application.

The goal of the perturbation threshold is to ensure the total cost of all the data being collected does not exceed pre-defined threshold for each resource. When a request for new instrumentation is received, its predicted cost is computed. If the request can be accommodated without exceeding any of the thresholds, it is processed; otherwise, the request is deferred.

We now describe how the predicted cost can be used with W^3 Search Model. In manual search mode, the predicted cost simply acts as a check to see if the request associated with a hypothesis can be evaluated without undue perturbation. In automated search mode, the interaction between the Predicted Cost Model and the search process is more complex. Recall, that in automated searching, we develop an ordered list of possible refinements to test, and then work down that list adding instrumentation and evaluating the results. When the Predicted Cost Model is used, we use this ordered list and request instrumentation for each test. However, when a test request is deferred because the instrumentation overhead is too high, we stop requesting new instrumentation and let the program continue to execute. If we find a refinement that is true (before the sufficient observation time has expired), then we start to consider refinements of that bottleneck. However, if no hypothesis is satisfied after the sufficient observation time has passed, we stop considering that set of hypotheses and move onto the next group from the list of possible refinements. Thus the perturbation threshold regulates the number of hypotheses that can be considered at one time. By raising the threshold, the search system can try more tests at once, but with higher perturbation that could decrease the accuracy of the results. However, changing the threshold does not change what hypotheses get

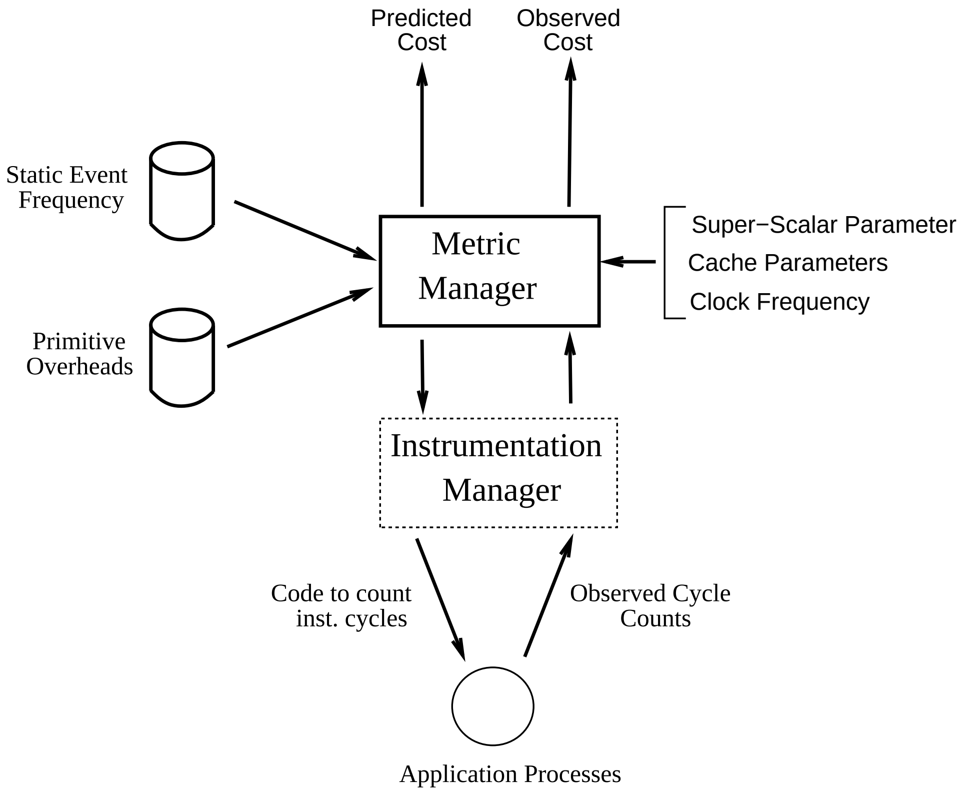

Figure 7.1. Computing Predicted and Observed Costs.

tested; it simply changes when they get tested.

Another (complementary) approach to increase the speed of the search is to change the order that hypotheses are considered to try to decrease the total instrumentation batches required. However, recall that the instrumentation requests from the W^3 Search Model are ordered based on a set of heuristics that have the most promising refinements to the current bottleneck at the top of the list. So, we would not wish to re-order the refinements haphazardly. In addition, finding a better ordering is a variation on the the bin packing problem, which is NP-complete[31]. When we consider multiple resource constraints, we have an instance of the multi-dimensional bin packing problem[21, 30]. This problem is also NP-complete, and the best known heuristics have a divergence from optimal packing that is linear with the number of dimensions (resources). Currently, we do not re-order refinements, but in the future we might make small changes to try to decrease the number of instrumentation batches considered.

7.2. Observed Cost

Predicted cost is based on a model of how instrumentation perturbs a program's execution. However, it is always a good idea to check that the model matches reality. That is where the *Observed Cost Model* is used. The observed cost indicates the affect on the application from collecting data. Its purpose is to check that the overhead of data collection does not exceed pre-defined levels, and if it exceeds these levels, report it to the higher level consumers of the data. Since the observed cost acts as an alarm system to report when instrumentation has gotten out of hand, the value for this threshold should be higher than for predicted cost.

Actual cost might also differ from predicted cost because resource contention between the application and the data collection can affect cost. For example, collecting data might consume a large percentage of a resource (e.g., the interconnection network), but if the application is not using that resource then the impact on the application is small. Alternatively, if the data collection needs a resource that the application is using heavily, it could have a major impact on the application's performance. Perturbation effects are difficult to fully predict a priori, and so measuring instrumentation is vital.

The most uncertain parameter used in the Predicted Cost Model is the execution frequency of each point. As a result, a key goal of the Observed Cost Model is to verify this value. To compute the Observed Cost Model, we add a extra instrumentation to our instrumentation to record its execution time. This information is periodically sampled from the application. Based on these values, we compute the observed cost. If the aggregate perturbation exceeds the user defined threshold, we report this event. Figure 7.1 shows how we use observed event execution frequency to compute the predicted cost.

The Observed Cost Model is just a performance metric to characterize a type of bottleneck in a parallel program. We have already developed a way of isolating bottlenecks in parallel programs, the W^3 Search Model. The only difference is that the bottleneck in which we are interested was created by the data collection system rather than the programmer. We treat this potential bottleneck like an application bottleneck and use the W^3 Search

Model to look for it. At the level of the W^3 Search Model, the Observed Cost Model is expressed as additional hypotheses along the ‘‘why’’ axis, that can be isolated to specific resources along the ‘‘where’’ axis, and characterized temporally along the ‘‘when’’ axis.

A closely related topic to our two part cost model is the work that has been done on perturbation compensation[59, 100]. The goal of perturbation compensation is to reconstruct the performance of an un-perturbed execution from a perturbed one. These techniques generally require a trace based instrumentation system and post-mortem analysis to reconstruct the correct ordering of events. Our approach differs in that we do not try to factor out perturbation; instead we try to avoid it with the Predicted Cost Model, and quantify it using the Observed Cost Model.

7.3. Implementation of Observed Cost

Since the purpose of the Observed Cost Model is to report the time consumed by instrumentation, the simplest way to implement it would be to add additional instrumentation to the system to record the time spent executing the instrumentation code. However, the overhead required to execute this meta-instrumentation would be as expensive, if not more expensive, than the instrumentation we are trying to measure. Instead, we need a cheap but relatively accurate way to measure the cost of our instrumentation. We developed an implementation of the Observed Cost Model that provides the approximate information and requires only one additional instruction in each of the base and mini-trampolines. We conducted several tests to verify the accuracy of this approximate cost model and found that for a wide variety of degrees of instrumentation inserted, the measured program running time with instrumentation was within 5% of the original running time plus the observed cost estimate.

To implement the Observed Cost Model, we added an additional instruction to each trampoline to record the number of machine cycles required to execute that trampoline. There are two components to the instrumentation: the trampolines and the primitives. Since there are a small number of primitives, their performance can be measured once for each platform (as we did in Chapter 6), and used as the measured cycle counts for the observed cost. Since the instructions for each trampoline are generated by the Dynamic Instrumentation system, all that is needed is to keep track of the number of cycles required for a particular trampoline as it gets generated. The cost of a mini-trampoline, in machine cycles, is the cost of the trampoline plus any primitives it calls. The cost of a base trampoline is the fixed instruction sequence it executes. Each trampoline includes code to increment the total observed cost by its cycle count. The total observed cost is stored in a register dedicated to this task†.

† On the SPARC, we were able to use one of the three ABI reserved registers for this purpose. On other platforms, we would need to either use memory or perform register scavenging as part of the pre-pass we make over the program prior to running the system.

The observed cycle counter provides us with a precise count of the number of instructions that are executed for instrumentation. However, we still need to convert instruction counts to time; this is where the observed cost value becomes an approximation. In particular, the impact of super-scalar processors and caches must be approximated. Super-scalar processors can issue more than one instruction per machine cycle. But, dependencies between the operands of instructions mean that rarely can instructions be issued at the maximum rate. Likewise, memory hierarchies add uncertainty to the execution time due cache misses from either the instrumentation instructions or data. For our initial implementation, we used the following approximations to convert from cycle counts to time. First, the processor used in the sequential experiments was a two-way super-scalar processor. We decided to pick a single constant factor to characterize the degree to which our instrumentation was able to use super-scaling. Based on tests of typical instrumentation sequences, we used 1.5 instructions per cycle for our Observed Cost Model. An alternative would be to analyze the instrumentation sequences to develop a more precise estimate of the number of cycles required for each instrumentation block. To do this, we could use Wang's[96] framework of modeling instructions by their functional unit requirements to get a more accurate estimate. To account for cache time, we used measured times for the instrumentation primitives. These measurements resulted in approximately once cache miss per clock primitive. An alternative way to compute cache misses is to model the cache, but this is far more complex than trying to model the super-scalar instruction times. The final step in converting from counted cycles to time is to divide the cycle count by the clock frequency of the machine.

Integrating the implementation of the observed cost with Dynamic Instrumentation was easy. Since our instrumentation system was designed to support reading external sources of performance data (e.g., hardware and OS counters), we simply treat the observed cost as one of these metrics and use the normal Dynamic Instrumentation mechanisms to report this data to the higher level consumers of performance data. By treating this observed cost as a normal metric, we can use the facilities of the metric description language to constrain the metric to particular resource combinations within the application.

Once we had implemented the observed cost metric, we were interested in seeing how well it tracked with the actual perturbation of applications. To investigate this, we ran three sequential programs from the floating point SPEC92 benchmark suite[105], and one parallel application. For each program, we used the UNIX ''time'' command to measure the CPU time required to execute the program without Dynamic Instrumentation. We then ran each program with instrumentation, and measured the new CPU time for the program, and recorded the value of the Observed Cost Model. Any difference between the sum of the un-instrumented CPU time plus the observed cost and the measured CPU time of the instrumented program is the error in the observed cost value.

For each program, we measured its performance with four different levels of instrumentation enabled: base, procedure profiling, Performance Consultant base, and Performance Consultant full. Base is the amount of instrumentation inserted by starting up Dynamic Instrumentation. It consists of instrumentation to record the start and end of the application. It also causes the application to be run as a child process using the UNIX ptrace facility. Procedure profiling consisted of turning on CPU metrics for each user supplied procedure in the application. The Performance Consultant base case enabled the initial instrumentation used by the Performance Consultant to search for a bottleneck in the application. For the Performance Consultant full case, we ran the Performance Consultant in fully automated mode and let it turn on and off instrumentation as needed. Since we were interested in assessing the accuracy of the cost model, we did not want to use the cost model to control the number of refinements being considered. However, we also did not want to completely overwhelm the application with instrumentation by enabling all refinements at once. As a compromise, we configured the Performance Consultant to consider ten refinements refinements at once.

The three sequential applications we measured were: Ear, Fpppp, and Doduc. The applications were selected to reflect a variety of different programming styles. In particular, since instrumentation is currently inserted at procedure boundaries, we wanted a cross section of procedure granularity and procedure call frequency. We also measured another program, Tomcatv, to represent the low end of procedure call frequency but we were never able to measurably perturb it. The programs were run on a idle SPARCstation 10 running at 40Mhz.

The results for the first application, Ear, are shown at the top of Figure 7.2. This program consists of 16 files containing 93 functions. The un-instrumented version of the program runs for just over ten minutes, and averages 11,000 procedure calls per second during its execution. The results shown in the table show that the observed cost metric is within 2% of the CPU time for the application for all cases. The total perturbation ranged from about 1% to just over 10% of the CPU time of the un-instrumented version.

The second program we measured was Fpppp, a quantum chemistry benchmark which does electron integral derivatives. It consists of 13 files and 13 procedures. The un-instrumented running time of the program is just under 5 minutes with an average of 2,100 procedure calls per second. The observed perturbation for this program ranges from 4 to 16%. The difference between the measured and observed cost metric ranges from 1.7 to 3.6 percent.

The third program is Doduc, a Monte Carlo simulation of the time evolution of a thermo-hydraulical model of a nuclear reactor. It is composed of 41 files and 41 procedures. This program averages 107,000 procedure calls per second. The un-instrumented version runs for just over two minutes. The perturbation for this program ranged from 1 to 61 percent of the application's CPU time. This program has the largest error in the observed cost metric. This happens on the Performance Consultant full search case. The reason for the divergence between the observed overhead and measured overhead is due to the additional overhead required to insert and delete the instrumentation from the program. We conducted a number of experiments to verify this fact, but we are currently unable to fully account for the cause of this overhead. We are continuing to investigate why this occurs.

The fourth program we measured is a parallel graph coloring application running on a 32 node partition of a Thinking Machines CM-5. The nodes of the CM-5 are 33 Mhz (non super-scalar) SPARC processors. This program is written in C++ and uses the explicit message passing library, CMMD, provided by TMC. A comparison of the Observed Cost Model values and the actual perturbation for this program appears in Figure 7.3. The un-instrumented version of the program ran for 216 seconds. The actual perturbation ranges from 0.5% to 2.6% of the

Version	Time	Measured Overhead Time	Measured Overhead Percent	Observed Overhead Time	Observed Overhead Percent	Difference (of Percent)
Application: Ear						
Un-Instrumented	736.7					
Base	748.1	11.5	1.6%	0.0	0.0%	1.6%
Procedure	781.3	44.6	6.1%	34.6	4.7%	1.4%
PC Base	754.2	17.6	2.4%	3.8	0.5%	1.9%
PC Full	811.1	74.4	10.1%	71.1	9.7%	0.4%
Application: Fpppp						
Un-Instrumented	298.1					
Base	303.2	5.1	1.7%	0.0	0.0%	1.7%
Procedure	314.2	16.1	5.4%	5.7	1.9%	3.5%
PC Base	307.4	9.3	3.1%	1.2	0.4%	2.7%
PC Full	314.7	16.6	5.6%	5.7	1.9%	3.7%
Application: Doduc						
Un-Instrumented	73.3					
Base	74.4	1.1	1.5%	0.0	0.0%	1.5%
Procedure	118.4	45.1	61.6%	47.1	64.2%	-2.6%
PC Base	91.7	18.4	25.0%	17.1	23.3%	1.7%
PC Full	90.3	17.0	23.2%	13.7	18.6%	4.6%

Figure 7.2. Measured vs. Observed Overhead.

This table shows the difference between the instrumented and un-instrumented CPU time (measured overhead) and the Observed Cost Model (observed overhead) values for the three sequential applications. The final column shows the difference in percent of CPU time between the observed overhead plus the un-instrumented time and the measured overhead. **All times are shown in seconds.**

CPU time of the program. Just enabling the Dynamic Instrumentation system slowed the program down by 1.7%. We believe this is due to the addition of the measurement process and its periodic use of ptrace to poll the application for performance data.

7.4. Implementation of Predicted Cost

We were also interested in evaluating the quality of the predicted cost model. The implementation of the Predicted Cost Model is based on using the same techniques to compute the cycle time of each trampoline that we

Version	Time	Measured Overhead Time	Measured Overhead Percent	Observed Overhead Time	Observed Overhead Percent	Difference (of Percent)
Un-instrumented	216.0					
Base	220.1	3.7	1.7%	0.0	0.0%	1.7%
Procedure	221.8	5.3	2.4%	1.5	0.7%	1.7%
PC Base	223.6	7.1	3.3%	1.4	0.7%	2.6%
PC Full	226.8	10.8	5.0%	9.7	4.5%	0.5%

Figure 7.3. Measured vs. Observed Overhead for a Parallel Application.

This table shows the difference between the measured overhead and the value of the Observed Cost Model (observed overhead) for the parallel graph coloring application. **All times are shown in seconds.**

used in the Observed Cost Model. However, rather than counting each time a trampoline is executed, we must estimate the event frequency. The values we used were 1,000 calls per second for user procedures and 100 calls per second for system and library routines.

To gauge how effective the Predicted Cost Model is, we ran the same applications we used to study the observed cost metric, and measured the predicted cost metric. These numbers were based on using the static predicted cost information and do not include any compensation based on the observed cost.

The predicted cost for the Ear program is shown in Figure 7.4. The errors in three of the four cases are within 10% of the actual CPU time of the program. However, for the procedure call case the number is over 100% off. This is not surprising given that the Predicted Cost Model is using a point execution frequency estimate of 1,000 calls per second for **each** procedures in the program. The numbers are much better for the cases that are more typical of the amount and type of instrumentation inserted by the W^3 Search Model.

The middle section of Figure 7.4 shows the predicted cost for the Fpppp application. For all cases, the estimated cost was within 11% of the measured CPU time. In addition, the largest error was again for the all procedure profiling case.

The last part of Figure 7.4 shows the predicted cost for the Doduc application. The errors in the Predicted Cost Model ranged from 1.5% to 37.7% in this case. However, again the largest error was for the case of instrumenting the CPU time for all procedures.

Finally, we measured the predicted cost compared to actual perturbation for the parallel graph coloring program. Figure 7.5 shows the results of this case. The case is interesting because the predicted cost is much closer to

Version	Measured Overhead Time	Measured Overhead Percent	Predicted Overhead Time	Predicted Overhead Percent	Difference (of Percent)
Application: Ear					
Base	11.5	1.6%	0.0	0.0%	1.6%
Procedure	44.6	6.1%	876.9	118.9%	-112.9%
PC Base	17.6	2.3%	29.6	4.1%	-1.6%
PC Full	74.4	10.1%	1.1	0.2%	9.9%
Application: Fpppp					
Base	5.1	1.7%	0.0	0.0%	1.7%
Procedure	16.1	5.4%	50.1	16.8%	-10.4%
PC Base	9.3	3.1%	18.7	6.2%	-3.1%
PC Full	16.6	5.6%	37.6	12.6%	-7.0%
Application: Doduc					
Base	1.1	1.5%	0.0	0.0%	1.5%
Procedure	45.1	61.6%	72.8	99.3%	-37.7%
PC Base	18.4	25.0%	5.6	7.6%	17.5%
PC Full	17.0	23.2%	14.8	20.2%	3.0%

Figure 7.4. Measured vs. Predicted Overhead.

This table compares the measured overhead and the value of the Predicted Cost Model (predicted overhead) for the three sequential applications. **All times are shown in seconds.**

the measured cost than for the sequential applications. The reason for this lies with the number and frequency of procedure calls. The program is one of several the author wrote to compare graph coloring heuristic. Most of the time is spent in a graph library that was not instrumented for this study. The non-library procedures calls that were instrumented happened to match well with the Predicted Cost Model.

In this chapter, we have described an observed and predicted cost model that can be used to regulate the impact performance instrumentation has on an application. We also studied the quality of the cost model for several real programs. In the next chapter, we combine the W^3 Search Model, Dynamic Instrumentation, and the cost models to measure the performance of two real applications.

Version	Measured Overhead Time	Measured Overhead Percent	Predicted Overhead Time	Predicted Overhead Percent	Difference (of Percent)
Base	3.7	1.7%	0.0	0.0%	1.7%
Procedure	5.3	2.4%	9.3	4.3%	-1.9%
PC Base	7.1	3.3%	4.2	2.0%	1.3%
PC Full	10.8	5.0%	6.8	3.2%	1.8%

Figure 7.5. Measured vs. Predicted Overhead for a Parallel Application.

This table compares the measured overhead and the value of the Predicted Cost Model (predicted overhead) for the graph coloring application. **All times are shown in seconds.**

Chapter 8

BRINGING IT ALL TOGETHER

In previous chapters, we described the components required to build a performance monitoring environment for large-scale parallel computing: the W^3 Search Model, Dynamic Instrumentation, and a data collection cost model. In this chapter, we bring all of these ideas together. First, we describe how the ideas were integrated into a parallel performance monitoring tool. Second, we report the performance of two real programs written by application programmers, running on a Thinking Machines CM-5. For each of the two applications, we found a bottleneck that accounted for over 45% of the execution time of the program. In addition, in both cases the instrumentation perturbation of the application was less than 5% of the un-instrumented running time of the program. For the case studies, we were interested in judging the quality of the tool, not the programmer using the tool, therefore we located performance bottlenecks, but did not try to fix them.

8.1. Implementation

The ideas presented in the earlier chapters of the thesis have been incorporated into a parallel program measurement system called Paradyn that is being developed at the University of Wisconsin. First, we outline how Paradyn is structured. We do this to illustrate how the various components of the thesis interact in a real implementation. Second, we describe the Paradyn user interface. The purpose of doing this is to provide an idea of what it is like to use a performance tool that incorporates W^3 Search Model and Dynamic Instrumentation. Third, we describe several additional features that have been incorporated into Paradyn to make it a useful tool for measuring programs. Paradyn is an ongoing project and the implementation described here is a group effort.

Paradyn currently runs on the CM-5 and a network of workstations running PVM. The Paradyn Performance Tools Suite is a modular collection of tools to measure and understand the performance of large-scale parallel programs. Each module has a well defined interface and can communicate with any other module via that module's interface. By using these well defined interfaces, additional components can easily be added to the system.

Figure 8.1 shows the structure of the Paradyn system and how the Performance Consultant (the name of the implementation of the W^3 Search Model) and Dynamic Instrumentation have been integrated into it. The central part of the tool is a multi-threaded process that contains the Performance Consultant, the User Interface, and threads to communicate with data collection and visualization processes. Interactions with the tool user are handled by the User Interface Manager thread. Requests to collect data, notifications of new resources, and delivery of performance data are handled by a Data Manager thread. Visualizations are separate processes and makes requests to the main Paradyn process via an RPC interface. Each visualization process also has a thread in the Paradyn main process to handle requests to the User Interface and Data Manager threads. The Performance Consultant is another thread in the main Paradyn process; it requests data collection and receives performance data via the same interface to the Data Manager that the visualization threads use. Dynamic Instrumentation is implemented by Paradyn daemons. There is one Paradyn daemon per machine, and they communicate with the main Paradyn process via an RPC interface.

The design philosophy of Paradyn is that any module can request that any other module do something. This architecture means that any component of the system can ‘‘drive’’ the tool. For example, in manual search mode the user, via the user interface manager thread, requests data to be collected and visualizations to be displayed. In automated mode, the Performance Consultant thread makes the same types of requests. In addition, visualizations may need to enable additional data collection and this happens transparently. Which thread is driving the system is irrelevant to most of the threads.

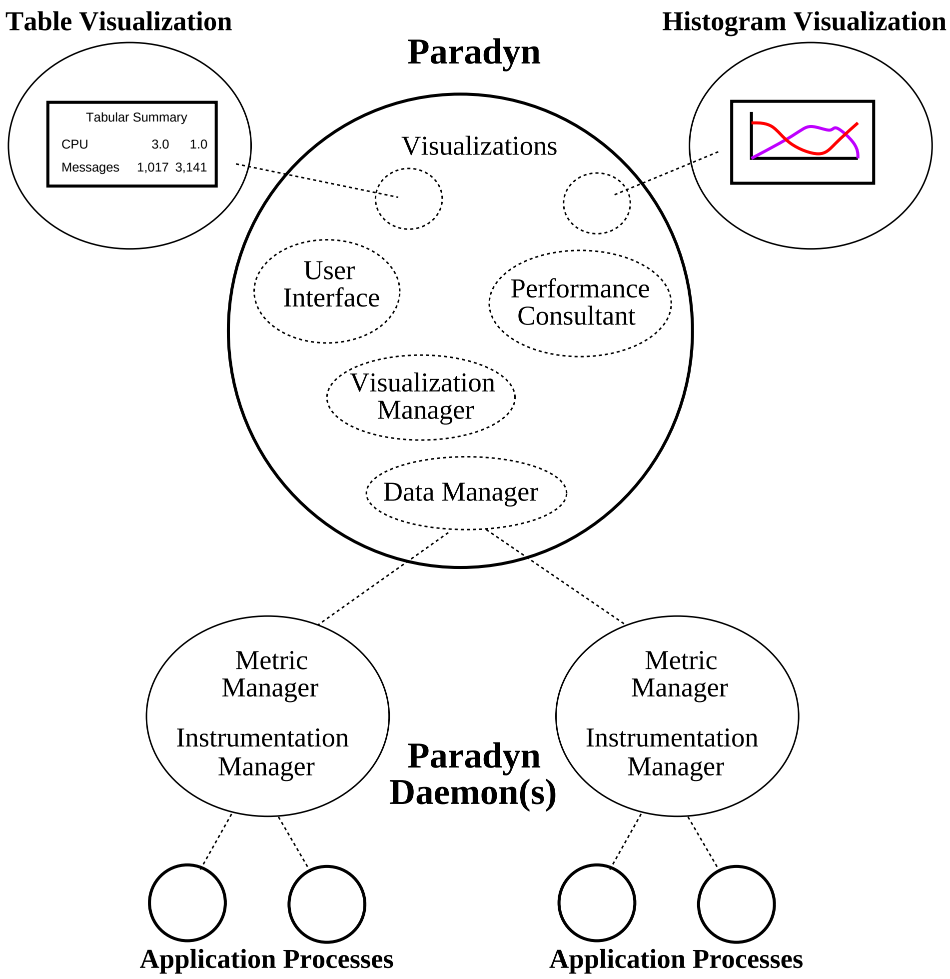

Figure 8.1. Architecture of the Paradyn Tool Suite.

The user interface for Paradyn is still being developed, but currently consists of a window for defining the application to run and displaying the trees of the resource hierarchies (i.e., ‘‘where’’ axis). Additional windows appear in response to actions by the user. A typical Paradyn main window appears in Figure 8.2. The options along the top of the display are pull down menus that provide ways to specify the application to measure, to start the Performance Consultant, and to start visualizations. In addition, the top row includes a pull down menu to create visualizations; currently time histograms and tables are available. The middle area shows the resource hierarchies of the ‘‘where’’ axis as a forest of trees. The row of buttons along the bottom is used to control the application’s execution and query its status.

When the Performance Consultant is enabled, an additional window appears to show the search process. A sample of this window appears in Figure 8.3. The top row contains a series of pull down menus to configure the search. The middle area has space for text messages that report the status of the search (i.e., when the sufficient observation time has elapsed, or when a new set of refinements is being considered). The largest area is a graphical illustration of the Search History Graph. The graph is color coded to indicate the state of each node in the graph. States include: true, tested false, un-tested, and being tested. The bottom of the window contains four buttons to control the search process.

To build a usable Performance tool, we also implemented two additional features not described earlier in this thesis. First, we needed to be able to run the Performance Consultant on a machine with other users. To accommodate this, we added a load compensation factor to the thresholds used in the ‘‘why’’ axis of the W^3 Search Model. The compensation factor is defined as the fraction of the available time that the measured application was running. For the CM-5, computing this compensation factor was easy since the machine uses synchronous gang scheduling

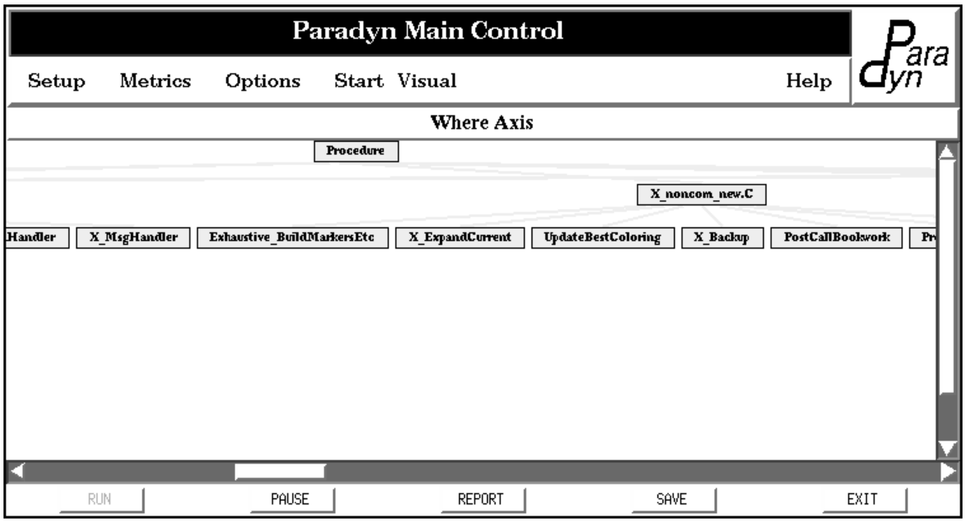

Figure 8.2. The Main Paradyn Control Window.

and provides a timer to record the fraction of wall time that the user application was running. Paradyn Daemons periodically sample this value and report it to the Performance Consultant.

Second, we added a feature for the user to specify that a particular bottleneck was not interesting and to continue searching for other bottlenecks. We provide two interfaces to this feature. First, the user can select a single node from the Search History Graph and request that refinements not be made to this node. This provides specific control for a particular combination of the ‘‘why’’, ‘‘where’’ and ‘‘when’’ axes. Second, we permit users to select nodes along the ‘‘where’’ axis and indicate that no refinement to a ‘‘where’’ axis focus involving that node should be considered. Using this second interface, all types of bottlenecks for a particular resource can be ignored. In the near future, we plan to add a similar ignore option to the ‘‘why’’ axis to permit different types of bottlenecks to be ignored for all ‘‘where’’ and ‘‘when’’ combinations. Ignoring a node was implemented by changing the ordering function that determines when refinements are evaluated. As a result, the semantics of the ignore request are to defer testing the selected refinement until all other refinements have been considered. This definition permits the ignore mechanism to also be used to control the order in which refinements are considered. We used this feature in our case studies to specify that process resources should only be considered after all other refinements are tried. Saving process refinements until the end is useful for SPMD style programs since any significant bottlenecks are likely to be diffused across many processes.

8.2. Graph Coloring Application.

The first application we measured with the Paradyn system was a graph coloring program called ‘‘match-maker’’. Match-maker was one of several programs written by another PhD student at the University of Wisconsin to compare different parallel graph coloring algorithms[57]. Match-maker is a branch and bound search with a central manager that brokers work to idle processors. It uses TMC’s explicit message passing library CMMD. The program is written in C++ and contains 4,600 lines of code in 37 files. Part of the application is contained in several libraries that provide data structures and communication routines common to all the graph coloring algorithms that the author tried. The algorithm specific part of the match-maker program consists of 1,150 lines of C++ in 9 files. We decided to measure the performance of the algorithm specific part of the program. To do this, we used the Performance Consultant’s ignore ‘‘where’’ axis option to suppress instrumentation of the library. Since un-instrumented procedures are not separately tracked by Dynamic Instrumentation, the time spent in the libraries was automatically folded into the calling routines.

The un-instrumented version of the program runs for 216 seconds on a 32-node CM-5 partition. To try out all of the ideas presented in this thesis, we started the application from within Paradyn, and enabled the Performance Consultant to measure the application. The Performance Consultant was run in automated mode and discovered a CPU bottleneck in the program 10 seconds into its execution. A screen dump of the Performance Consultant window for the graph coloring application appears in Figure 8.3. In the first step, the root hypothesis (there is a bottleneck somewhere in the program) is evaluated. Next, different types of bottlenecks are considered (synchronization, I/O, CPU, virtual memory, and instrumentation). At this point, a CPU bottleneck was identified. At the next step, refinements to that hypothesis were considered, and the hypothesis that the program was CPU bound was tested and confirmed. Next, the Performance Consultant refined the CPU bottleneck to a specific module in the program, `X_noncom_new.C`. The bottleneck was then isolated to the procedure `PostCallBookwork` in that module. Since the problem was diffused across all of the processes, the Performance Consultant could not further refine the bottleneck. We verified the output of the Performance Consultant by displaying a time histogram and a table of the CPU time for the application as a whole and for the procedure `PostCallBookwork`. The procedure `PostCallBookwork` was responsible for over 40% of the execution time of the program.

To find this bottleneck, the Performance Consultant enabled 6 different metrics for a total of 61 different combinations of the ''where'' axis. The number of resource combinations tried is an indication of the variety of hypotheses for different ''where'' axis combinations considered. A total of 3,540 samples were collected. Since we sampled once per second and the instrumented version of the program ran for 227 seconds, we were collecting only 16 samples per second on average. The low volume of data collected is impressive. To get an idea of how little data needed to be collected, consider how much data would have been gathered if just the 6 metrics used were enabled for each of the 61 foci considered for the entire execution of the application. In this case, Dynamic Instrumentation would have collected 78,324 samples. However, there were many more metrics and resource

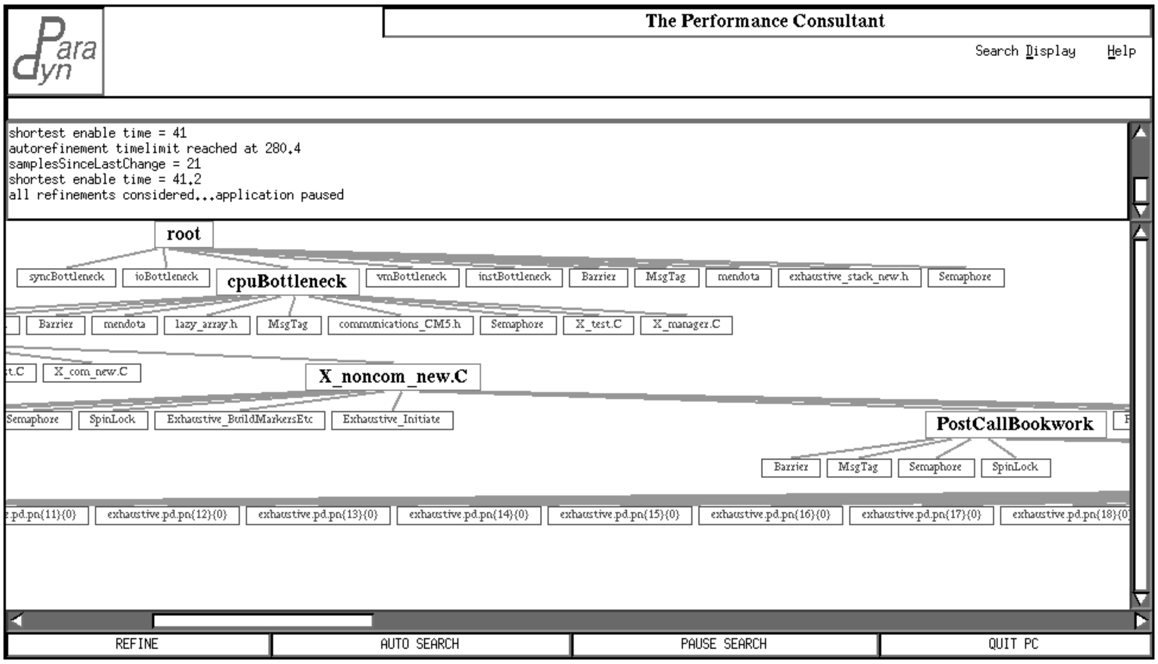

Figure 8.3. Performance Consultant Search for the Graph Coloring Application.

The bold nodes show hypotheses and foci that are true and being further refined. The bottleneck shown in this example is due to high CPU utilization in the file X_noncom_new.C resulting from a single procedure, PostCallBookwork.

combinations that were available to the Performance Consultant but not required to find this bottleneck.

We also we were interested in comparing the volume of performance data collected by Dynamic Instrumentation to an event based tracing system such as IPS-2. Since IPS-2 does not run on the CM-5, we needed to estimate the size of the trace log that would have been created by using IPS-2 style instrumentation. To do this we made a separate execution of the application and turned on metrics to count the number of procedure calls made and the number of messages sent by the application. These are the events that IPS-2 logs that occur frequently during an application's execution. By computing the trace size of these events, we can approximate the size of an IPS-2 trace file. The application made 24,606,100 procedure calls and sent 39,305 messages. As a result, the IPS-2 trace for this execution would be over 78 megabytes. Dynamic instrumentation's 3,540 samples totaled 99,120 bytes or a factor of 790 less than full tracing!

We also measured the impact of Dynamic Instrumentation on the application. We quantified the perturbation in two ways. First, we used the Observed Cost Model to record the perturbation of the application. Observed cost indicated 9.7 seconds (4.5%) perturbation for this application. A time histogram for the CPU utilization of this application and the observed cost appears in Figure 8.4. We also compared the running time of the un-instrumented version of the program to the version with the Performance Consultant running. We measured the running time of the program by inserting calls to the vendor supplied timer routines at the start and end of the application, and reported the total time the application was executing. The measured running time of the program with instrumentation was 226.8 seconds or 5.0% perturbation. The numbers are encouraging for two reasons. First, the perturbation was well within the 5-10% range that we consider reasonable. Second, the Observed Cost Model provided an accurate estimate of the actual perturbation.

Finally, we wanted to quantify the perturbation of the application caused by generating instrumentation code during application execution. To do this, we added some code to the Instrumentation system to record statistics about code generation. For this application, Dynamic Instrumentation generated and inserted 853 mini-trampolines in 1.4 seconds. To insert this instrumentation, the application being measured must be paused. The impact of inserting instrumentation can be seen in Figure 8.4. The four dips in the CPU and observed cost curves are due to Dynamic Instrumentation pausing the application. At each of these points, the sufficient observation time has expired and the Performance Consultant is disabling the old instrumentation and requesting new instrumentation for the next batch of refinements to be tested. Although these dips are visually noticeable, they have no impact on the search system since the time the application is paused is included as non-execution time in the load compensation factor. Since the load compensation factor is a normal metric in the system, users can plot it on a time histogram and see the loss in performance due to other users on the machine and Dynamic Instrumentation code generation.

8.3. Msolv Application

The second application we measured was called Msolv. Msolv uses a domain-decomposition method for optimizing large-scale linear models. The application consists of 1,793 lines code in the C programming language.

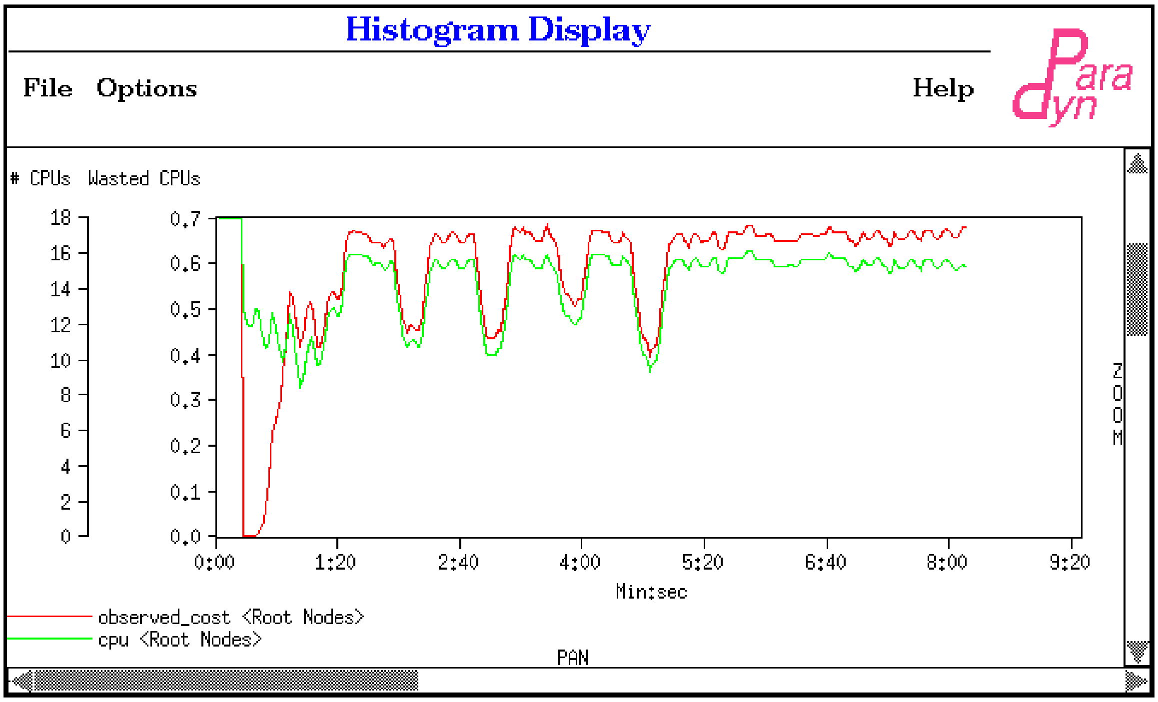

Figure 8.4. Time Histogram of the Graph Coloring Application.

The figure shows the CPU time and observed cost for the graph coloring application. The observed cost is shown as total instrumentation across the 32 processors in the partition (color original).

In addition, this program makes use of a sequential constrained optimization package, Minos[70]. We choose to instrument only the application specific code. Again, we used the ignore ‘‘where’’ axis option to suppress instrumentation of the library. This approach treats the library routines as a black box and records the performance of the program directly in terms of procedures written by the author of the application. From our informal observations of users of other performance tools, this is generally how programmers start a measurement session. However, when we were measuring the performance of Dynamic Instrumentation using this application in Chapter 6, we were interested in having as much instrumentation as possible to measure so we did instrument the library routine. As a result, the perturbation numbers for the application are different between these two chapters.

Un-instrumented, Msolv runs for 1 hour 48 seconds on a 32-node CM-5. We also ran this program using the Performance Consultant in automated search mode. The Performance Consultant found three bottlenecks in this program. First, during an initialization phase, it found a synchronization bottleneck as the nodes were getting started. This bottleneck lasted less than a minute, and was not further refined by the Performance Consultant. Second, during an initial computation phase a CPU time bottleneck in the module `minos_part.c` was identified. Third, the Performance Consultant located a key synchronization bottleneck. This third bottleneck persisted for the rest of the program’s execution. The Search History Graph of the isolation of this third bottleneck appears in Figure 8.5. Locating the cause of this problem involved a five step refinement. First, the Performance Consultant

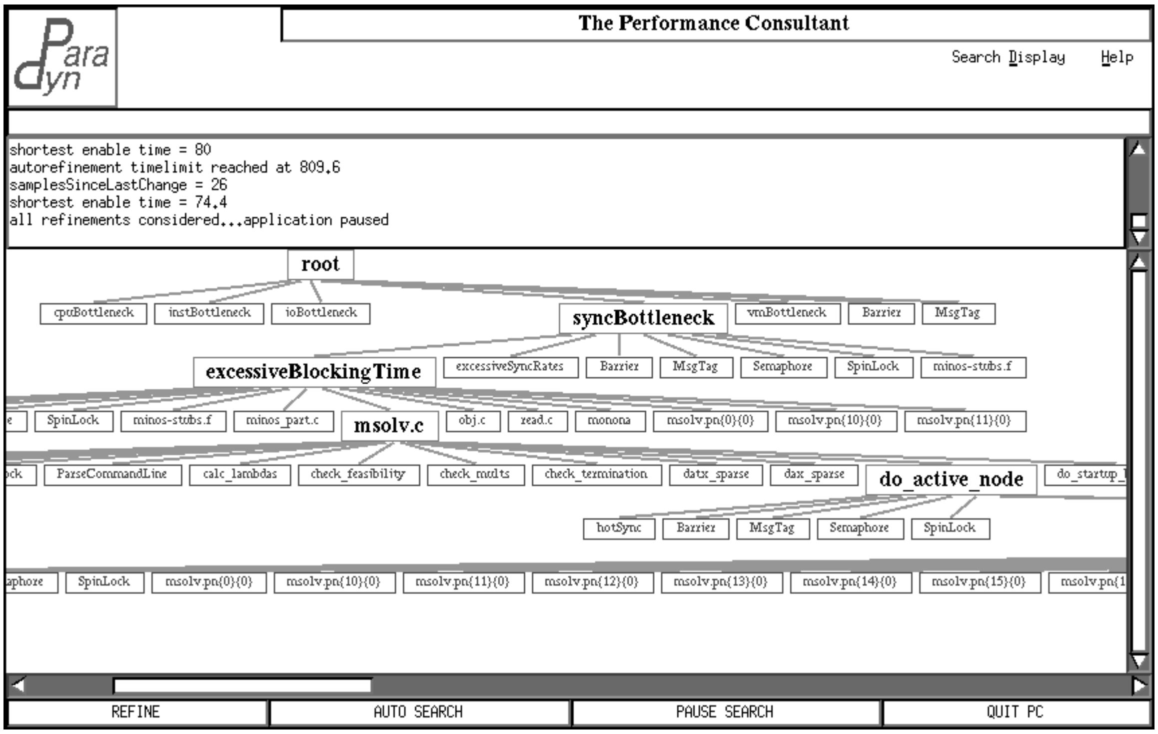

Figure 8.5. Performance Consultant Search for the Msolv Application.

discovered a synchronization bottleneck in this program. Second, it identified that the bottleneck was due to `excessiveBlockingTime` (as opposed to too frequent of small synchronization operations). Third, the synchronization bottleneck was then isolated to the file `msolv.c`. Fourth, the bottleneck was refined to the procedure `do_active_node`. Fifth, the problem was (trivially) isolated to the single partition used. The partition refinement is labeled mendota, the name of the partition's manager. (This node is not visible on the screen dump). Isolating a bottleneck to a specific machine when the application uses only one machine is trivial. Since the problem was diffused across all of the processes, no further isolation of the bottleneck was possible. This bottleneck was responsible for 46% of the running time of the program. The source of problem is that processes periodically exchange values using a parallel prefix operation (add). The synchronization delay is due to a load imbalance between the processes.

For the Msolv application, we also measured the instrumentation perturbation. The Observed Cost Model indicated the perturbation for this program was 0.13 seconds. The difference between the execution time of the uninstrumented version of the program and the version run with the Performance Consultant was -45 seconds or -1.2%. Since the runtime of the application program varies by about 2% from run to run, the instrumentation overhead for this application is less than the normal variation of its runtime. Dynamic instrumentation was able to keep the instrumentation overhead low due to the fact once the bottleneck had been found, the only instrumentation required was to measure synchronization time in the critical procedure.

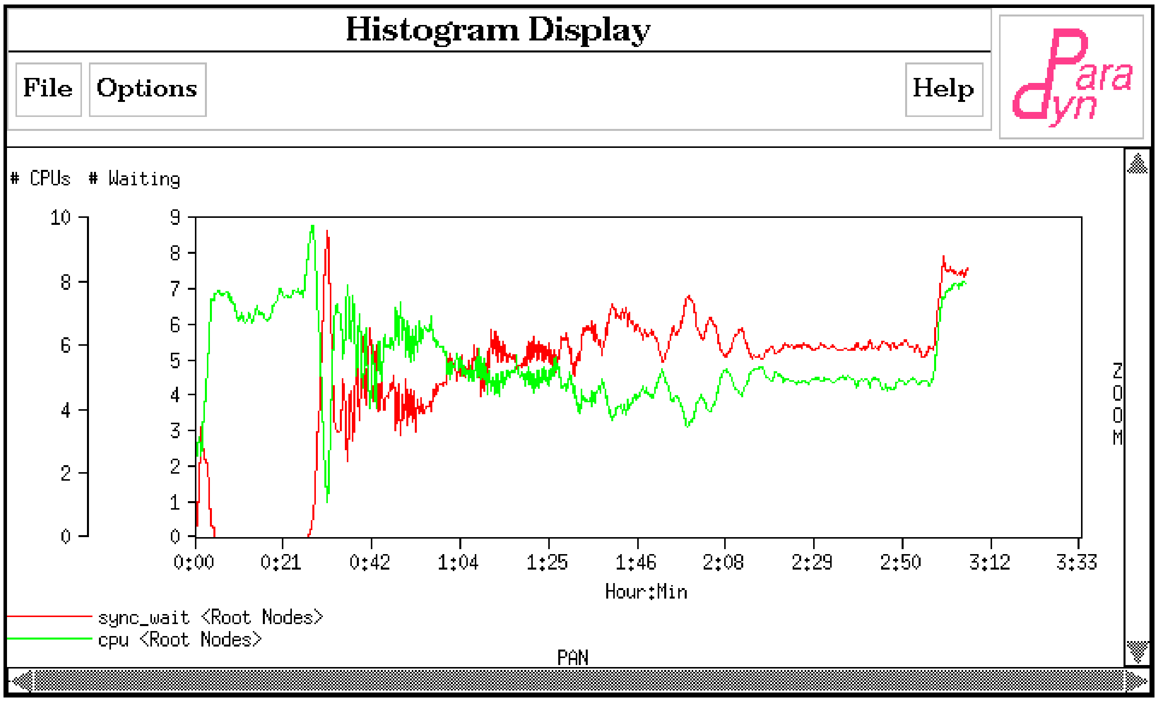

Figure 8.6. Time Histogram for the Msolv Application.

The curves show the CPU time and synchronization waiting time for the Msolv program. The program was run on a partition with two other applications, so the elapsed wall time for the application is almost 3 hours (color original).

Finally, we wanted to compare the amount of data collected by Dynamic Instrumentation for this application to the amount that a trace-based system like IPS-2 would collect. The application made 623,723 procedure calls, and sent 64,112 messages. As a result, IPS-2 would generate 2.2 megabytes of trace data. Dynamic Instrumentation delivered 18,146 samples totaling 508,088 bytes. This is a factor of 4 less than full event tracing.

8.4. Conclusion

Building a performance tool to incorporate the ideas in this thesis was time consuming, but worth the effort since it is the only way to demonstrate that all of the ideas can be combined in complementary ways. In addition, several valuable lessons were learned. First, it was necessary to include the load compensation factor into the Performance Consultant to make the tool usable on production parallel computers. Second, it is important to provide visual feedback that an automated performance tool is working. Users, especially the author, are impatient and suspicious so providing a visual cue that something is happening is important. Paradyn provides several types of visual feedback during a search including: dynamic update of the Search History Graph and time histograms. Third, refinements to specific processes should only be tried after all other refinements have been considered when there are a large number of processes in the application.

In this chapter we presented two case studies measuring real applications, written by other researchers at the University of Wisconsin, using an implementation of the W^3 Search Model and Dynamic Instrumentation. In both cases, we were able to isolate significant bottlenecks in these programs and do so with a low perturbation of the application. In the next chapter, we summarize the contribution of this thesis and outline areas of future work.

Chapter 9

CONCLUSIONS AND FUTURE WORK

9.1. Conclusions

This thesis has addressed the problem of locating the source of performance bottlenecks in large-scale parallel and distributed applications. We identified two key problems that must be solved to monitor large-scale applications. First, identifying a bottleneck necessitates collecting detailed information, yet collecting all this data can introduce serious data collection bottlenecks. Second, users are being inundated with volumes of complex graphs and tables that require a performance expert to interpret. To solve these two problems, we have developed the W^3 Search Model, which combines dynamic on-the-fly selection of what performance data to collect with decision support to assist users with the selection and presentation of performance data. To make it possible to implement the W^3 Search Model, we developed a new monitoring technique for parallel programs called Dynamic Instrumentation. We also presented a prototype implementation of these ideas and showed that the prototype is able to efficiently find performance problems in several real applications. A conclusion of this thesis is that not only is it possible to do on-line performance debugging, but for large scale parallelism it is mandatory.

The W^3 Search Model closes the loop between data collection and analysis. Searching for a performance problem becomes an iterative process of refining the answers to three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the *why* question, tests are conducted to identify the type of bottleneck (e.g., synchronization, I/O, computation). Answering the *where* question isolates a performance bottleneck to a specific resource used by the program (e.g., a disk system, a synchronization variable, or a procedure). Answering *when* a problem occurs, tries to isolate a bottleneck to a specific phase of the program's execution.

Dynamic Instrumentation differs from traditional data collection because it defers selecting what data to collect until the program is running. This permits insertion and alteration of the instrumentation during program execution. It also features a new type of data collection that combines the low data volume of sampling with the accuracy of tracing. Instrumentation to precisely count and time events is inserted by dynamically modifying the binary program. These counters and timers are then periodically sampled to provide intermediate values to the W^3 Search Model. Based on this intermediate data, changes are made in the instrumentation to collect more information to further isolate the bottleneck.

We have described a prototype implementation of the W^3 Search Model and of Dynamic Instrumentation. The prototype runs on Thinking Machine's CM-5 and a network of workstations running PVM. We also presented several studies to demonstrate the effectiveness of the ideas presented in this thesis. In the first study, the tool identified the bottlenecks in several real programs using two to three orders of magnitude less data than traditional techniques. In another study, Dynamic Instrumentation was used to monitor long running programs and introduced less than 10% perturbation. We also described a data collection cost model and showed that it was able to track the performance of several real applications to within 5% of their actual CPU time. Finally, we combined the W^3 Search Model, Dynamic Instrumentation, and the cost model and ran a prototype of the system on two real programs.

Although the ideas presented in this thesis combine to complement each other, they are also useful individually. The W^3 Search Model can be applied to existing post-mortem performance tools or even to simulated machines and environments. Moreover, the model can be viewed as a methodology for performance debugging that could be used by programmers whether or not it is built into the system. Dynamic Instrumentation has been used to collect performance data for other uses including visualization. Also, by delaying instrumentation decisions until the program is in execution, a much broader variety of data can be made available to programmers with no cost until they request it.

9.2. Future Work

The ideas presented in this thesis open up a new way of designing and building performance tools that is more dynamic and automatic than previous approaches. However, since many of these ideas are new, opportunities exist for further exploration and investigation. In this section, we identify several of these questions and briefly describe how they might be solved.

9.2.1. The W^3 Search Model

There remain many opportunities to enhance the W^3 Search Model. For example, the W^3 Search Model measures performance for one component from each resource hierarchy along the where axis; for example, a procedure in a process on one machine. However, sometimes the performance of a program can be better understood by considering the relationships between two (or more) components of a single resource hierarchy. For example, the dynamic call graph relationships between procedures is often important. To represent this type of information, I plan to extend the where axis from one item in each resource hierarch to combinations of resources. This would permit expressing not only call graph information, but also other types of relationships between resources (e.g., sender/receiver pairs for message traffic).

An additional major area of future work lies in the explanation system. In Chapters 3 and 4, we described an explanation system to relate performance information back to users. However, the initial version of this is quite primitive and only handles text based explanations. A full implementation of the explanation system requires development of a sophisticated visualization interface to support the Performance Consultant creating and altering visualizations. In addition, work needs to be done to figure out what visualizations are good for what types of bottlenecks and add this information to the hypotheses of the why axis.

There are also several minor enhancements to the decision support work. The W^3 Search Model could be extended to better support searching for multiple performance bottlenecks. Currently, we identify one bottleneck in the program and include a simple mechanism to allow the user to ignore that bottleneck and continue searching. For example, users can mark certain nodes of the Searching History Graph as ignore. A more complex problem with searching for multiple bottlenecks is that often they are not independent. For example, two bottlenecks might exist in the application and as a result, neither one consumes a large enough fraction of the program's time to satisfy the test conditions. In this case, we might need to scale the thresholds to permit searching for this type of double bottleneck.

A different type of multiple bottleneck arrises due to multi-programming on parallel computers. The reason a parallel program may not run as fast as desired could be due to other users of the machine (from either time or space sharing). Due to the cost and inherently shared nature of most parallel computers, it is important for a performance tool to be able to recognize and compensate for non-exclusive use of a machine. To be able to quantify performance degradation due to competition with other processes, we need to get information from the operating system about how long application processes are waiting due to other processes in the system. For the CM-5, since it uses synchronous gang scheduling and provides an execution timer, the information we need is available. Measuring interference is difficult in a non-gang scheduled environment because of the potential starvation of one process when another process from that application gets de-scheduled.

Another area of future work on the W^3 Search Model is to investigate the interactions of the current components of the model. The W^3 Search Model contains several parameters that affect its performance. In particular there is minimum observation time, sufficient observation time, perturbation threshold, and the hysteresis threshold. I plan to investigate the impact of changing these parameters on a real programs. The observation time parameters affect how quickly the search model is able to refine a bottleneck. There is also an interaction between the perturbation threshold that controls how many refinements can be considered at once and the sufficient observation time that determines the how long a refinement is considered.

9.2.2. Dynamic Instrumentation

Work could also be done to enhance Dynamic Instrumentation. Currently, several different instrumentation requests are generally enabled at one time. These multiple requests often contain overlapping instrumentation needs; this is especially true for resource constraint expressions. For example, several different metrics might be constrained on the same procedure. We could reduce the amount of instrumentation inserted into the application, and thus the perturbation, by taking advantage of these common instrumentation expressions. This type of optimization is similar to global common expression elimination and inter-procedural optimization in traditional compilers.

Another aspect of Dynamic Instrumentation that could be improved is the granularity of instrumentation requests. Currently, Dynamic Instrumentation works down to the level of procedure calls. However, many applications, especially scientific code, contain several distinct operations in a single procedure. To provide meaningful performance data for these types of programs, dynamic instrumentation needs to be extended to work at the loop and statement levels. While the design of Dynamic Instrumentation permits this, several technical details need to be resolved. In particular, the machine state that must be saved before making a call to Dynamic Instrumentation is larger. At procedure calls, registers are volatile and the value of the processor’s condition codes are undefined. However, within a procedure neither of these assumptions hold. To instrument at this level requires more information about the the program to determine live registers and condition codes. This information could be gathered by additional analysis of the binary image, or perhaps provided by the compiler. Alternatively, we could save all of the machine state, but that is usually too expensive.

One simple area where the current implementation of Dynamic Instrumentation could be improved is in the quality of the instrumentation code generated. First, the instrumentation complier is rather naive and could generate better code for each instrumentation request. In particular, register and delay slots utilization could be improved. The techniques required to do this are well understood, and should be a simple implementation issue. These enhancements are local optimization, while the common instrumentation expression optimizations described are more global.

9.2.3. Other Uses for Dynamic Monitoring

Initially, I have concentrated on the problem of monitoring and tuning a single application. A closely related problem is monitoring and tuning a machine or cluster for a collection of programs (i.e., workload). I plan to investigate how my ideas could be used to to tune machines and help with capacity planing.

An area of future work is to apply the ideas and techniques presented in this thesis to applications besides parallel computing. For example, computer networks can be viewed as a large distributed program. Like other programs, networks contain performance problems due to coding inefficiencies and resource utilization imbalances. Large computer networks are also continuously evolving and capacity planing is an important problem. In addition, the same driving problems of efficient data collection and information overload exist in this domain. I plan to apply the ideas I have developed about dynamic monitoring and decision support to computer networks.

Another idea from this thesis that might have broader application is the time histogram. The time histogram data structure can be generalized into a dynamic histogram by using variables besides time as the x-axis. Dynamic histograms are a good data structure for recording the distribution of frequency of events where the range of the distribution is not known until the data arrives. Since we fold adjacent buckets together when the bound of the distribution axis has been exceeded, the fold operation is simple because it only combines pairs of complete buckets. In addition, this style of folding reduces the aliasing of samples due to folding. One possible application of dynamic histograms is to record memory use statistics where the amount of memory used by a program is not known until the program is running.

Finally, perhaps the time has arrived to integrate performance and correctness debuggers. The contributions of this thesis in moving performance debugging to an interactive, execution time activity via the use of dynamic code generation means that the already hazy line between correctness and performance debugging is vanishing. First, both activities are now taking place during program execution. Second, the techniques of Dynamic

Instrumentation to insert code at runtime provide an efficient mechanism for conditional breakpoints. In addition, Dynamic Instrumentation provides a way to enable the most widely used debugger, print statements, at runtime.

Appendix A

DAMPING TEST OSCILLATIONS

Tests are boolean expressions of a metric and a constant threshold. As a test is evaluated, if the value of the metric used in the test is close to the test threshold, the result of the test may oscillate back and forth between true and false. This oscillation is undesirable since it causes the W^3 Search Model to search and then back up and try again when the test becomes false. Searching and backing up creates excessive requests to turn instrumentation on and off. In Chapter 3, we introduced a hysteresis parameter to damp these oscillations. With the addition of the hysteresis parameter, we would like to know that no matter how the value of the metric used in the test varies, the frequency of the oscillations is damped. We will now show that this is true:

Let H be a constant which is the threshold used to indicate when a test changes from false to true.

Let f(t) denote the value of the metric at time t. The value of $f(t) \geq 0$ for all t. This assumption is true for all metrics we currently compute.

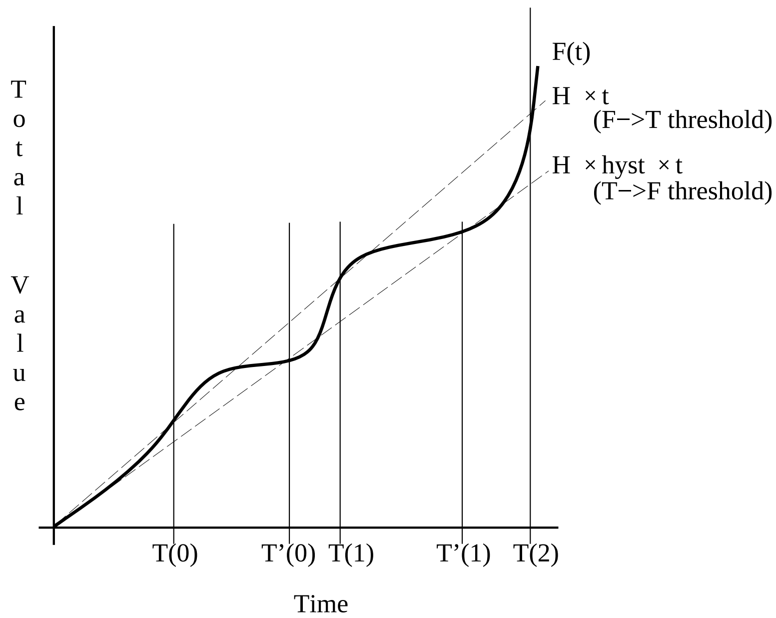

Figure A.1. Graph of cumulative metric value vs. time (F(t)).

Let F(t) (shown in Figure A.1) denote the cumulative value of the function f(t) at time t.

Let *hyst* be the hysteresis constant ($0 < hyst < 1$).

Let $T(i)$ denote the time of the ith occurrence of the test changing from false to true. Therefore at time $T(i)$ the average value of the metric is larger than *H*. $T(i)$ for several values of i is shown in Figure A.1.

By the definition of the time $T(i)$:

$$F(T(i)) \geq H \times T(i) \tag{1}$$

Likewise, for the i+1 value of T:

$$F(T(i+1)) \geq H \times [T(i+1)] \tag{2}$$

Now, let $T'(i)$ denote the time of the ith transition of the test from true to false. Therefore at time $T'(i)$ the average value of the metric is less than the threshold multiplied by the hysteresis constant *hyst*:

$$hyst \times H \times T'(i) \geq F(T'(i)) \tag{3}$$

Since the cumulative value of the metric must become less than the $H \times hyst$ before it can be true again, the following time order holds:

$$T(i+1) > T'(i) \geq T(i) \tag{4}$$

Since f(t) is non-zero for all values of t, F(t) is monotonically increasing, therefore we can use the time ordering of (4) to get:

$$F(T(i+1)) > F(T'(i)) \geq F(T(i)) \tag{5}$$

$$hyst \times H \times T'(i) \geq F(T(i)) \qquad \text{combine 3 \& 5 (6)}$$

$$hyst \times H \times T'(i) \geq H \times T(i) \qquad \text{combine 6 \& 1 (7)}$$

$$T'(i) \geq \frac{T(i)}{hyst} \tag{8}$$

$$T(i+1) \geq \frac{T(i)}{hyst} \qquad \text{combine 4 \& 8 (9)}$$

Now we reduce the recurrence relation to its closed form:

$$T(i+1) \geq \frac{T(0)}{hyst^i} \tag{10}$$

Equation 10 gives an expression for the time of the i+1 transition of the test value from false to true in terms of the time of the 0*th* transition and the value of the hysteresis constant. However, since we wait a minimum observation time before concluding that a test is true, $T(0) \geq minObsTime$ and therefore:

$$T(i+1) \;\geq\; \frac{minObsTime}{hyst^{\,i}} \tag{11}$$

Equation 11 shows that no matter how the metric value used in the test varies with time, by using a hysteresis constant, the oscillations of the test are damped. The analysis used no knowledge of how the value of the metric changes with time (except that it is non-negative). However, in practice we often know the largest possible value of a metric (e.g. CPU utilization of a processor is less than one). This additional information can be used to derive a tighter bound on the rate at which the oscillations are damped. The intuition is that since we know the metric has a maximum value, it takes time proportional to this maximum for the cumulative average to increase.

To perform this analysis, let's assume the metric function has some maximum value M, then the change in the cumulative value of the function between time $T'(i)$ and $T(i+1)$ must be less than or equal to the maximum value times the time between $T'(i)$ and $T(i+1)$:

$$M \times [T(i+1) - T'(i)] \;\geq\; F(T(i+1)) - F(T'(i)) \tag{12}$$

Combine equations 2 and 12:

$$F(T'(i)) + M \times [T(i+1) - T'(i)] \;\geq\; F(T(i+1)) \geq H \times T(i+1) \tag{13}$$

Now use equation 3 to substitute for the first term in equation 13:

$$hyst \times H \times T'(i) + M \times T(i+1) - M\,T'(i) \;\geq\; H \times T(i+1)$$

$$(M - H) \times T(i+1) \;\geq\; M \times T'(i) - hyst \times H \times T'(i)$$

$$(M - H) \times T(i+1) \;\geq\; T'(i) \times (M - hyst \times H) \tag{14}$$

$$(M - H) \times T(i+1) \;\geq\; \frac{T(i)}{hyst}\,(M - hyst \times H) \qquad \text{combine 14 and 8}$$

$$T(i+1) \;\geq\; \left(\frac{M - hyst \times H}{(M - H) \times hyst}\right) T(i)$$

Reduce the relation to closed form:

$$T(i+1) \;\geq\; \left(\frac{M - hyst \times H}{(M - H) \times hyst}\right)^{i} T(0)$$

Since we wait a minimum observation time before concluding that a test is true, $T(0) \geq minObsTime$:

$$T(i+1) \geq \left(\frac{M - hyst \times H}{(M - H) \times hyst}\right)^{i} minObsTime$$

Therefore as *hyst* gets smaller, the ratio:

$$\frac{M - hyst \times H}{(M - H)}$$

gets bigger so the value of $T(i+1)$ gets bigger, and thus we have a faster damping of the oscillations than in the case where we don't know the maximum value of the metric function.

This analysis has shown that the hysteresis parameter has the desired damping affect. However, an important question is how to set the hysteresis value. We (somewhat arbitrarily) currently use the value 0.9. This has proved to be effective in meeting our goal of damping the oscillations, and at the same time it prevents us from concluding that a hypothesis is true long after its average value has become low enough to be false.

Appendix B

METRIC DEFINITION LANGUAGE

B.1. Introduction

The purpose of this appendix is to describe the features of the Metric Description Language (MDL). This appendix is not intended as a language tutorial, but a number of examples appear in the final section.

MDL is a simple programming language for specifying templates of instrumentation to be inserted into an application program to gather information about its performance. The language has four parts: variable definitions, unit definitions, metric definitions and constraint definitions. The goal of the language is to describe how to compute metrics (interesting variables about a program), and constrain these metrics to particular resources used during the program's execution (e.g., a particular procedure).

MDL differs from most languages in that part of the program specification is executed when a request to insert instrumentation is received, and part of it executes inside the application process to measure its performance. These two modes are called metric insertion (insert) and instrumentation execution (exec).

In describing MDL, we use the following typesetting conventions. **Bold** words indicate keywords in the language. `Courrier` refers to predefined symbols. *Italic* denotes a syntactic category (e.g., a non-terminal in the grammar for the language).

B.2. Lexical Attributes

The lexical attributes of MDL are intended to be similar to Algol-like languages. The language C++ is probably the closest lexically. All whitespace including spaces, tabs, and newlines are ignored except to separate tokens and inside string constants.

The token two slashes (//) starts a comment. All information from the start of comment token to the end of the line is ignored.

Identifiers are a sequence of characters, digits, and the underscore character (_); they must start with a character. Identifiers are case sensitive. Predefined identifiers start with a dollar sign ($).

The following identifiers are reserved as keywords and may not appear elsewhere:

aggregateOperator	**append**	**at**
base	**constraint**	**constrained**
derived	**foldOperator**	**foreach**
if	**is**	**in**
metric	**name**	**prepend**
replace	**type**	**units**

There are three types of constants in the languages: integer, real, and string. Integer literals are a sequence of digits optionally starting with the minus sign. Real constants are a sequence of digits followed by a period and another sequence of digits. Reals may also have an optional leading minus sign in front of them. String constants start with a double quote (") and are a sequence of characters terminated by a second double quote. The backslash character (\) is an escape character in strings and can be used to embed double quotes (\") in a string. Two backslashes (\\) are taken to be a single backslash in the string.

B.3. Variables

The metric description has two flavors of variables corresponding to the two modes of evaluation: metric insertion variables and instrumentation execution variables. Instrumentation execution variables (exec) can only appear in instrumentation request blocks (described below). However, metric insertion variables (insert) may appear anywhere. When a metric instantiation variable appears in an instrumentation request block, its value when the block is inserted into the application program is used. Everytime the instrumentation block executes, the variable will have the same value.

There is one name space for variables in MDL. Metric instantiation variables, instrumentation execution variables, metric names, units, constraints and functions share this name space. The scoping of all names is global.

Variables also have a type associated with them. The predefined types of variables are: list, counter, timer, function, and callsite.

Lists are metric insertion time variables and are aggregates of predefined types. The elements of lists must be metric instantiation time variables. Lists may be accessed either sequentially via the **foreach** iterator or directly via indexing (e.g., *identifier*[*int*]).

The syntax for declaring a list is:

listDefinition:
    identifier **= {** *identifier-list* **};**

identifier-list:
    identifier
    identifier-list **,** *identifier*

Counters are instrumentation execution time variables and are like integer variables in conventional languages.

Timers are instrumentation execution time variables that can record the time between two events. The representation of timers is machine specific, and they are an abstract data type that can only be accessed via timer functions (described in section B.8).

The function type is a structure that has fields for the instrumentation points associated with a function. The definition of the type is:

function {
    point `entry`*;*
    point `return`*;*
    list callSite `calls`*;*
};

The callsite type is a structure that has fields for the instrumentation points before and after a subroutine call. The definition of the type is:

callsite {
    point `preCall`*;*
    point `postCall`*;*
};

B.4. Units

Unit definitions are used to assign a name to the units of a metric. Different metrics can have the same units (e.g. messages sent and floating point instruction executed might be defined in operations per second). Unit definitions are the way to express this information. Unit definitions are used by higher level consumers of data.

unitDefinition:
    unit *ident* **{ name** *string-constant* **; }**

B.5. Constraints

Constraints define a way to restrict a metric to a single instance of a resource from one of the resource hierarchies. Constraints declare a counter which should be positive only when the desired resource is active. The name of the counter is the same as the name of the constraint. Each constraint also contains a path name which defines the resource hierarchy (or part of the hierarchy) that the constraint clause applies to. For example, a path of `/Procedure` defines a constraint that restricts a metric to a single procedure. The variable `$constraint` is implicitly initialized at metric instantiation time to be the trailing component of the resource path name. For the procedure example, `$constraint` is the name of the procedure we are constraining to.

constraintDefinition:
    constraint *identifier matchPath* **is** `counter` *statementList*

matchPath:
    / *identifier*

/ *identifier matchPath*

B.6. Metrics

A metric definition declares a time varying variable that describes some aspect of the performance of a parallel program. Each metric definition also includes clauses to define how to constrain the metric along components of resource hierarchies.

metricDefinition:
  metric *identifier* **is** *identifier* **{** *metricBody* **}**

metricBody:
  metricHeaderList constraintList baseMetric

metricHeader:
  name *stringConstant* **;**
  unit *identifier* **;**
  foldOperator `sum|average` **;**
  aggregationOperator `sum|average|min|max`**;**

constraintList:
  constraint *identifier*
  constraintList **constraint** *identifier*

baseMetric:
  base is *identifier* **{** *statementList* **}**

B.7. Statements and Expressions

Statements appearing inside an instrumentation request block are evaluated at metric execution time. All other statements are evaluated during metric interpretation.

statement:
  foreach *identifier* **in** *identifier statement*
  instrumentationRequest
  if *expression statement*
  { *statementList* **}**
  functionCall **;**

There are several types of expressions in the MDL. Expressions are either be evaluated at instrumentation execution time if they occur in an instrumentation block, or at metric instantiation otherwise.

expression:
  rval
  rval binaryOperator rval
  (*expression* **)**

binaryOperator:
  *+ - / * < > <= >= == != and or*

rval:

identifier

integerConstant

stringConstant

floatConstant

rval **->** *ident*

functionCall

B.8. Function Calls

A function call is indicated by an identifier followed by an optional list of expressions which are the actual parameters to the function.

functionCall:
identifier **(** *argList$_{opt}$* **)**

argList:
expression
argList **,** *expression*

MDL includes a number of pre-defined functions. For each function, we show its name, when it executes, and a brief description. All functions execute either at metric insertion time (shown as Insert) or metric execution time (shown as Exec).

Function	When	Description
addCounter(counter)	Exec	Adds one to the passed counter
car(path)	Insert	returns the next component of a path name
cdr(path)	Insert	returns the rest of a path after the next component is removed
lookupModule(string)	Insert	lookup the passed string and return its module info
lookupProcedure(string)	Insert	lookup the passed string and return its procedure info
setCounter(counter, int)	Exec	sets counter to the passed value
startProcessTimer(timer)	Exec	starts the passed timer recording CPU time
stopProcessTimer(timer)	Exec	stop the passed timer
startWallTimer(timer)	Exec	starts the passed timer recording wall time
stopWallTimer(timer)	Exec	stops the passed timer
subCounter(counter)	Exec	Subtracts one from the passed counter

B.9. Predefined Variables

MDL includes a number of predefined variables:

Name	When	Type	Description
$arg	Exec	List (int)	The arguments to the procedure call†
$modules	Insert	List (strings)	Modules in the application program
$procedures	Insert	List (strings)	Procedures in the application program
$constraint	Insert	string	The trailing component of a resource path after the matchPath
$return	Exec	int	The return value of the procedure†

B.10. Instrumentation Requests

An instrumentation request consists of a description of a point where instrumentation should be inserted and a block of instrumentation to insert at that point. However, multiple instrumentation blocks can be inserted at the same point and the order of evaluation of instrumentation can affect the value of the metric. To control the order of execution of instrumentation at a point, MDL provides a mechanism to ensure instrumentation is evaluated in the correct order. Instrumentation at a point is a list of instrumentation requests for that point. At metric request time, constraint and base clauses are evaluated in the order in which they are specified in a metric definition. In addition, each instrumentation request block is either appended or prepended to the instrumentation block list for that point.

instrumentationRequest:
position **at** *point* **constrained** $_{opt}$ **[** *statementList* **]**

position:
append
prepend

point:
an expression whose type is `point`

B.11. Examples

This example shows a unit definition for a unit called operations per second.

† $arg is only valid at procedure entry and pre-call points, and $return is only valid at procedure exit and procedure return points.

```
units ops_per_second {
    name "Operations/Second";
};
```

The next example is a constraint clause that demonstrates the use of the foreach statement and the use of the instrumentation execution time variables to get the argument to a procedure. The constraint definition iterates through a list of message passing functions at metric insertion time and inserts calls to set the counter `MSGTagPred` to positive if the second argument `$argv[2]` to the function is equal to the target message tag (`$constraint`). This constraint also clears the counter at the return point from each message passing routine.

```
msgTagFuncs = { "cmmd_send", "cmmd_recv" };

constraint MSGTagPred /SyncObject/MsgTag is counter {
    foreach func in msgTagFuncs {
        prepend at lookupFunction(func)->entry [
            if ($arg[2] == $constraint)
              setCounter(MSGTagPred, 1);
        ]
        append at lookupFunction(func)->exit [
            setCounter(MSGTagPred, 0);
        ]
    }
}
```

The next example constraint demonstrates the use of the replace clause to define a constraint that (when requested) is used in lieu of the base metric. The replace clause is useful for metric definitions where the constrained version of the metric can be written more efficiently than a general metric plus a constraint clause. For example, computing the times a single procedure is called could be expressed as a constraint counter that is active when the desired procedure is called, and a base metric at every procedure call to check if the current procedure call is the desired one. However, a far more efficient metric uses the replace clause and simply increments a counter at entry to the desired procedure. The second version appears below. The instrumentation request also contains the constrained clause. This means that although this constraint replaces the base metric, it can still be constrained by other constraint clauses for the desired metric.

```
constraint callsPerProcedure /Procedure is
    replace counter {
      append at $constraint->entry constrained [
          addCounter(procedureCalls, 1);
      ]
}
```

The next example is a metric description for a metric that records the number of procedures called.

```
metric procedureCalls {
    name "Procedure Calls";
    units callPerSecond;
    foldOperator sum;
    constraint processConstraint;
    constraint procedureCallsPerProcedure;
    base is counter {
        foreach func in $procedures {
            append at lookupFunction(func)->entry
            constrained [
                addCounter(procedureCalls, 1);
            ]
        }
    }
}
```

REFERENCES

1. T. E. Anderson and E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", *1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boston, May 1990, pp. 115-125.
2. Z. Aral and I. Gertner, "A high-level debugger/profiler architecture for shared-memory multiprocessors", *1988 International Conference on Supercomputing*, New York, NY, 1988, pp. 131-139.
3. *Butterfly Product Overview*, BBN Advanced Computers, Inc., Cambridge, MA, 1987.
4. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs", *19th ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, January 19-22, 1992, pp. 59-70.
5. P. C. Bates and J. C. Wileden, "EDL: A Basis For Distributed System Debugging Tools", *15th Hawaii International Conference on System Sciences*, January 1982, pp. 86-93.
6. P. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 5-6, 1988, pp. 11-22. appears as SIGPLAN Notices, January 1989.
7. P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems", *8th Int'l Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 308-315.
8. A. Beguelin, J. Dongarra, A. Geist and V. S. Sunderam, "Visualization and debugging in a heterogeneous environment", *IEEE Computer 26*, 6 (June 1993), pp. 88-95.
9. T. Bemmerl, A. Bode, P. Braum, O. Hansen, T. Tremi and R. Wismuller, "The Design and Implementation of TOPSYS", TUM-INFO-07-71-440, Technische Universitat Muenchen, July 1991.
10. D. Bernstein, A. Bolmarcich and K. So, "Performance visualization of parallel programs on a shared memory multiprocessor system", *International Conference on Parallel Processing (ICPP)*, University Park, PA, USA, August 1989, pp. 1-10.
11. M. Bishop, "Profiling Under UNIX by Patching", *Software—Practice & Experience 17*, 10 (Oct. 1987), pp. 729-739.
12. W. C. Brantley, K. P. McAuliffe and T. A. Ngo, "RP3 Performance Monitoring Hardware", in *Instrumentation for Future Parallel Computer Systems*, M. Simmons, R. Koskela and I. Bucker (eds), Addison-Wesley, 1989, pp. 35-47.
13. O. Brewer, J. Dongarra and D. Sorensen, "Tools to aid in the analysis of memory access patterns for FORTRAN Programs", *Parallel Computing 9*, 1 (1988/89), pp. 25-35.
14. M. H. Brown, "Zeus: a system for algorithm animation and multi-view editing.", *1991 IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991, pp. 4-9.
15. J. S. Brown, *The Application of Code Instrumentation Technology in the Los Alamos Debugger*, Los Alamos National Laboratory, October 1992.
16. B. Bruegge, "A Portable Platform for Distributed Event Environments", *1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 20-21, 1991, pp. 184-193. appears as SIGPLAN Notices, December 1991.
17. H. Burkhart and R. Millen, "Performance Measurement Tools in a Multiprocessor Environment", *IEEE Trans. Computers 38*, 5 (May 1989), pp. 725-737.

18. “UNICOS File Formats and Special Files Reference Manual”, SR-2014 5.0, Cray Research Inc.

19. “Cray T3D System Architecture Overview”, HR-04033, Cray Research Inc., Eagan, MN.

20. M. E. Crovella and T. J. LeBlanc, “Performance Debugging Using Parallel Performance Predicates”, *1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, May 17-18, 1993, pp. 140-150.

21. J. Csirik, J. B. G. Frenk, M. Labbe and S. Zhang, “On the multidimensional vector bin packing.”, *Acta Cybernetica 9*, 4 (1990), pp. 361-9.

22. D. DeWitt and R. Gerber, “Multiprocessor Hash-Based Join Algorithms”, *1985 VLDB Conference*, Stockholm, Sweden, August 1985, pp. 151-164.

23. P. J. Denning, “Working Sets Past and Present”, *IEEE Transactions on Software Engineering SE-6,* 1 (Jan. 1980), pp. 64-84.

24. J. B. Dennis, “Segmentation and the Design of Multiprogrammed Computer Systems”, *Journal of the Association for Computing Machinery 12*, 4 (Oct. 1965), pp. 589-602.

25. J. Dongarra, A. Geist, R. Manchek and V. S. Sunderam, “Integrated PVM framework supports heterogeneous network computing”, *Computers in Physics 7*, 2 (March-April 1993), pp. 166-174.

26. C. E. Fineman and P. J. Hontalas, “Selective Monitoring Using Performance Metric Predicates”, *1992 Scalable High Performance Computing Conference*, Williamsburg, Virginia, April 26-29, 1992, pp. 162-165.

27. 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”, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 5-6, 1988, pp. 163-173. appears as SIGPLAN Notices, January 1989.

28. J. M. Francioni, L. Albright and J. A. Jackson, “Debugging Parallel Programs Using Sound”, *1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 20-21, 1991, pp. 68-75. appears as SIGPLAN Notices, December 1991.

29. M. Friedell, M. LaPolla, S. Kochhar, S. Sistare and J. Juda, “Visualizing the Behavior of Massively Parallel Programs”, *Supercomputing’91* , Albuquerque, NM, Nov. 18-22, 1991, pp. 472-480.

30. M. R. Garey, R. L. Graham and D. S. Johnson, “Resource Constrained Scheduling as Generalized Bin Packing”, *Combinatorial Theory 21* (1976), pp. 257-298.

31. M. R. Garey and D. S. Johnson, Computers and Intractability, 1979.

32. G. A. Geist, M. T. Heath, B. W. Peyton and P. H. Worley, *PICL - A Portable Instrumented Communication Library*, Oak Ridge National Laboratory, May 1990.

33. A. J. Goldberg and J. L. Hennessy, “Performance Debugging Shared Memory Multiprocessor Programs with MTOOL”, *Supercomputing’91* , Albuquerque, NM, Nov. 18-22, 1991, pp. 481-490.

34. S. L. Graham, P. B. Kessler and M. K. McKusick, “gprof: a Call Graph Execution Profiler”, *SIGPLAN ’82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120-126.

35. D. Haban and D. Wybranietz, “A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems”, *IEEE Transactions on Software Engineering 16*, 2 (Feb 1990), pp. 197-211.

36. G. J. Hansen, C. A. Linthicum and G. Brooks, “Experience with a Performance Analyzer for Multithreaded Application”, *1990 International Conference on Supercomputing*, Amsterdam, June 11-15, 1990, pp. 124-131.

37. M. T. Heath and J. A. Etheridge, “Visualizing Performance of Parallel Programs”, *IEEE Software 8*, 5 (Sept 1991).

38. U. Hecksen, R. Klar, W. Kleinoder and F. Kneibl, “Measuring Simultaneous Events in a Multiprocessor System”, *1982 SIGMETRICS Conference*, August 1982, pp. 77-88.

39. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, “The Integration of Application and System Based Metrics in A Parallel Program Performance Tool”, *1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming* , Williamsburg, VA, April 21-24 1991, pp. 189-200. appears as SIGPLAN Notices, July 1991.

40. J. K. Hollingsworth and B. P. Miller, “Parallel Program Performance Metrics: A Comparison and Validation”, *Supercomputing 1992*, Minneapolis, MN, November 1992, pp. 4-13.

41. M. Homewood and M. McLaren, “Meiko CS-2 interconnect”, *Proceedings of the 1993 World Transputer Congress*, Sept. 1993, pp. 1209-18.

42. A. A. Hough and J. E. Cuny, “Initial Experiences with a Pattern-Oriented Parallel Debugger”, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 5-6, 1988, pp. 195-205. appears as SIGPLAN Notices, January 1989.

43. A. A. Hough and J. E. Cuny, “Perspective Views: A Technique for Enhancing Parallel Program Visualization”, *International Conference on Parallel Processing (ICPP)*, August 1990, pp. II.124-132.

44. *Paragon X/PS Product Overview*, Intel Supercomputer Systems Division, 15201 N.W. Greenbrier Parkway, Beaverton OR, 1992.

45. S. C. Johnson, “Postloading for Fun and Profit”, *USENIX Winter Conf.*, Jan. 22-26, 1990, pp. 325-330.

46. J. Joyce, G. Lomow, K. Slind and B. Unger, “Monitoring Distributed Systems”, *ACM Transactions on Computer Systems 5*, 2 (May 1987), pp. 121-150.

47. P. B. Kessler, “Fast Breakpoints: Design and Implementation”, *ACM SIGPLAN’90 Conf. on Programming Language Design and Implementation*, White Plains, NY, June 20-22, 1990, pp. 78-84.

48. K. Kinoshita, “An Experience with the ANALYZER/SX Performance Tuning Tool”, in *Instrumentation for Future Parallel Computer Systems*, M. Simmons, R. Koskela and I. Bucker (eds), Addison-Wesley, 1989, pp. 223-231.

49. J. Kohn and W. Williams, “ATExpert”, *Journal of Parallel and Distributed Computing 18*, 2 (June 1993), pp. 205-222.

50. L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, *Comm. of the ACM 21*, 7 (July 1978), pp. 558-564.

51. F. Lange, R. Kroger and M. Gergeleit, “JEWEL: Design and Implementation of a Distributed Measurement System”, *IEEE Transactions on Parallel and Distributed Systems 3*, 6 (Nov 1992), pp. 657-671.

52. J. R. Larus, “Abstract Execution: A Technique for Efficiently Tracing Programs”, *Software—Practice & Experience 20*, 12 (Dec 1990), pp. 1241-1258.

53. T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging Parallel Programs with Instant Replay”, *IEEE Transactions on Computers 36*, 4 (April 1987), pp. 471-482.

54. T. J. LeBlanc, J. M. Mellor-Crummey and R. J. Fowler, “Analyzing Parallel Program Executions Using Multiple Views”, *Journal of Parallel and Distributed Computing 9*, 6 (1990), pp. 203-217.

55. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung and C. E. Fineman, “Visualizing Performance Debugging”, *IEEE Computer 21*, 10 (October 1989), pp. 38-51.

56. T. Lehr, D. Black, Z. Segall and D. Vrsalovic, “MKM: Mach Kernel Monitor Description, Examples and Measurements”, CMU, Carnegie-Mellon University, Pittsburgh, PA-CS-89-131, March 1989.

57. G. Lewandowski, *????*, PhD Thesis, University of Wisconsin-Madison, August 1994.

58. A. D. Malony, “Program Tracing in Cedar”, CSRD Report No. 660, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, April 1987.

59. A. Malony, *Performance Observability*, PhD Dissertation, Department of Computer Science, University of Illinois, Oct. 1990.

60. A. D. Malony and D. A. Reed, “A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube”, *1990 International Conference on Supercomputing*, Amsterdam, June 11-15, 1990, pp. 213-226.

61. K. Marzullo and M. D. Wood, “Tools for Constructing Distributed Reactive Systems”, 91-1187, Cornell University, January 1991.

62. H. Massalin and C. Pu, “Threads and Input/Output in the Synthesis Kernel”, *ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Dec 3-6 1989, pp. 191-201.

63. G. Mcdaniel, “METRIC: a kernel instrumentation system for distributed environments”, *6th Symp. on Operating System Prin.*, November 1977, pp. 93-99.

64. B. P. Miller, C. Macrander and S. Sechrest, “A Distributed Programs Monitor for Berkeley UNIX”, *5th International Conference on Distributed Computing Systems*, May 1985, pp. 43-54.

65. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, 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 (April 1990), pp. 206-217.

66. D. L. Mills, “On the accuracy and stability of clocks synchronized by the Network Time Protocol in the Internet system”, *Computer Communication Review 20*, 1 (Jan. 1990), pp. 65-75.

67. D. L. Mills, “Internet time synchronization: the network time protocol”, *IEEE Transactions on Communications 39*, 10 (Oct. 1991), pp. 1482-93.

68. A. Mink and R. Carpenter, “A VLSI Chip Set For A Multiprocessor Performance Measurement System”, in *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela (eds), Addison-Wesley, 1990, pp. 213-232.

69. A. Mink, R. Carpenter, G. Nacht and J. Roberts, “Multiprocessor Performance Measurement Instrumentation”, *IEEE Computer 23*, 9 (September 1990), pp. 63-75.

70. B. A. Murtagh and M. A. Saunders, “Large-scale Linearly Constrained Optimization”, *Mathematical Programming 14*, 1 (Jan. 1978), pp. 41-72.

71. C. M. Pancake and S. Utter, “Models for visualization in parallel debuggers”, *Supercomputing 1989*, Reno, NV, November 1989, pp. 627-636.

72. C. M. Pancake and C. Cook, “What Users Need in Parallel Tools Support: Survey Results and Analysis”, *1994 Scalable High-Performance Computing Conference*, May 1994, pp. 40-47.

73. S. E. Perl and W. E. Weihl, “Performance Assertion Checking”, *14th ACM Symposium on Operating Systems Principles*, December 5-8, 1993, pp. 134-145.

74. C. Ponder and R. Fateman, “Inaccuracies in Program Profilers”, *Software—Practice & Experience 18*, 5 (May 1988), pp. 459-476.

75. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz and L. F. Tavera, “Scalable Performance Analysis: The Pablo Performance Analysis Environment”, in *Scalable Parallel Libraries Conference*, A. Skjellum (ed), IEEE Computer Society, 1993.

76. M. Reilly, “Instrumentation for Application Performance Tuning: The M31 System”, in *Instrumentation for Future Parallel Computer Systems*, M. Simmons, R. Koskela and I. Bucker (eds), Addison-Wesley, 1989, pp. 143-158.

77. J. F. Reiser and J. P. Skudlarek, “Program Profiling Problems, and a Solution via Machine Language Rewriting”, *SIGPLAN Notices 29*, 1 (Jan. 1994), pp. 37-45.

78. B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards and W. Smith, “The Paragon Performance Monitoring Environment”, *Supercomputing’93*, Portland, OR, Nov 15-19, 1993, pp. 850-859.

79. D. A. Schneider and D. J. DeWitt, “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”, Tech. Report 836, Dept. of Comp. Sci., University of Wisconsin, April 1989.

80. K. Schwan, R. Ramnath, S. Vasudevan and D. M. Ogle, “A language and system for parallel programming”, *IEEE Transactions on Software Engineering,* April 1988, pp. 455-471.

81. 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*, 1 (January 1983), pp. 4-14.

82. Z. Segall and L. Rudolph, “PIE: A programming and instrumentation environment for parallel processing”, *IEEE Software 2*, 6 (November 1985), pp. 22-37.

83. J. P. Singh, W. Weber and A. Gupta, “SPLASH: Stanford Parallel Applications for Shared-Memory”, *Computer Architecture News 20*, 1 (March 1992), pp. 5-44.

84. S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons and R. Title, “Data Visualization and Performance Analysis in the Prism Programming Environment”, in *Programming Environments for Parallel Computing*, N. Topham, R. Ibbett and T. Bemmerl (eds), North-Holland, 1992, pp. 37-52.

85. R. L. Sites, ed., *Alpha Architecture Reference Manual*, Digital Press, 1992.

86. E. T. Smith, *Debugging Techniques for Communicating, Loosely-Coupled Processes*, PhD Thesis, University of Rochester, December 1981.

87. S. Smith and M. G. Williams, “The Use of Sound in an Exploratory Visualization Environment”, University of Lowell Computer Science Department Technical Report #R-89-002, 1989.

88. D. Socha, M. L. Baily and D. Notkin, “Voyeur: Graphical Views of Parallel Programs”, *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 5-6, 1988, pp. 206-215. appears as SIGPLAN Notices, January 1989.

89. D. H. Sonnenwald, B. Gopinaht, G. O. Haberman, W. M. K. III and J. S. Myers, “InfoSound: An Audio Aid to Program Comprehension”, *23rd Hawaii International Conferences on System Sciences 11* (1990), pp. 541-546.

90. J. T. Stasko and E. Kraemer, “A Methodology for Building Application-Specific Visualizations of Parallel Programs”, *Journal of Parallel and Distributed Computing 18*, 2 (June 1993), pp. 258-264.

91. C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild and M. Tsao, “The SP1 High-Performance Switch”, *1994 Scalable High-Performance Computing Conference,* May 1994, pp. 150-157.

92. S. S. Thakkar, Personal Communication.

93. R. Title, “Connection Machine Debugging and Performance Analysis: Present and Future”, *ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 20-21, 1991, pp. 272-275.

94. D. F. Vrsalovic, “Performance Efficient Parallel Programming in MPC”, *22nd Hawaii International Conference on System Sciences*, January 1989.

95. R. Wahbe, L. Lucco and S. L. Graham, “Practical data breakpoints: design and implementation”, *ACM SIGPLAN’93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM, June 23-25, 1993, pp. 1-12.

96. K. Wang, “Precise Compile-Time Performance Prediction of Superscalar-Based Computers”, *ACM SIGPLAN’94 Conf. on Programming Language Design and Implementation*, Orlando, FL, June 20-24, 1994, pp. 73-84.

97. E. H. Welbon, C. C. Chen-Nui, D. J. Shippy and D. A. Hicks, “The POWER2 Performance Monitor”, *IBM Journal of Research and Development (submitted)*, .

98. W. Williams, T. Hoel and D. Pase, “The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D”, in *Programming Environments for Massively Parallel Distributed Systems*, (ed), North-Holland, (To Appear) 1994.

99. D. Wybranietz and D. Haban, “Monitoring and Performance Measuring Distributed Systems during Operation”, *SIGMETRICS*, Santa Fe, New Mexico, May 1988, pp. 197-206.

100. J. C. Yan and S. Listgarten, “Intrusion Compensation for Performance Evaluation of Parallel Programs on a Multicomputer”, *6th International Conference on Parallel and Distributed Systems*, Louisville, KY, Oct. 14-16, 1993.

101. J. C. Yan, “Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers”, *27th Hawaii International Conference on System Sciences*, Jan 4-7 1994, pp. Vol II, 625-633.

102. C. Yang and B. P. Miller, “Critical Path Analysis for the Execution of Parallel and Distributed Programs”, *8th Int’l Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 366-375.

103. C. Yang and B. P. Miller, “Performance measurement of parallel and distributed programs: A structured and automatic approach”, *IEEE Trans. Software Eng. 12* (Dec. 1989), pp. 1615-1629.

104. D. Zernik and L. Rudolph, “Animating Work and Time for Debugging Parallel Programs Foundation and Experience”, *1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 20-21, 1991, pp. 46-56. appears as SIGPLAN Notices, December 1991.

105. “System Performance Evaluation Cooperative”, *Capacity Management Review 21*, 8 (Aug. 1993), pp. 4-12.

TABLE OF CONTENTS

TABLE OF FIGURES

Note: Figures 8.4 and 8.6 are in color in the thesis and some detail will be lost in a black and white reproduction.