To Appear International Conference on Supercomputing (Tokyo, July 19-23, 1993).

Dynamic Control of Performance Monitoring on Large Scale Parallel Systems

Jeffrey K. Hollingsworth
hollings@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Performance monitoring of large scale parallel computers creates a dilemma: we need to collect detailed information to find performance bottlenecks, 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. We present a new approach called the W^3 Search Model, *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. We present a case study describing how a prototype implementation of our technique was able to identify the bottlenecks in three real programs. In addition, we were able to reduce the amount of performance data collected by a factor ranging from 13 to 700 compared to traditional sampling and trace based instrumentation techniques.*

1. Introduction

Performance monitoring of applications running on large scale parallel computers can generate vast seas of (mostly useless) data. This wealth of information is a problem for the programmer who is forced to navigate through it, and for the tools which must store, process, or display it. The large volume of data results from the number and speed of the CPUs and the need to collect performance data down to the loop and basic block level to explain certain performance problems. As a result, there is so much data available that it is impossible to collect it all, and so programmers and tools are forced to select a subset of the data to collect. We propose a new paradigm based on dynamic on-the-fly selection of what performance data to collect. This approach permits us to collect the data we need when we need it. Although this approach provides great flexibility, it also requires many decisions be made about what to collect, when to collect it, and how to display what we collect. In this paper, we introduce a strategy called the W^3 Search Model, that is designed to provide decision support for the selection and presentation of performance data. We also describe a prototype implementation of the Performance Consultant, a system that incorporates the W^3 Search Model.

The W^3 Search Model is based on trying to answer three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the *why* question, our system includes hypotheses about potential bottlenecks in parallel programs. We collect performance data to test if these bottlenecks exist in the program. Answering the *where* question means that we isolate a performance bottleneck to a specific resource used by the program (e.g., a disk system, a synchronization variable, or a procedure). To identify *when* a problem occurs, we try to isolate a bottleneck to a specific phase of the program's execution. Finding a performance problem is an iterative process of refining our answers to these three questions.

We address the problem of selecting what performance data to collect by using our search model to direct the instrumentation to collect only the data necessary to test the next step in our search process. This means that instead of collecting large amounts of data to anticipate what the user wants, we only collect the data we need. As the system and the user refine their view of the performance problem, we adjust the instrumentation to collect more detailed, but more focused information. In addition, we can control the granularity of the performance data collected by dynamically adjusting the sampling frequency.

The W^3 Search Model addresses the problem of what to display by providing decision support for selecting appropriate displays and analyses. It is designed to give specific advice about where the performance bottlenecks lie in a program. Different displays are useful to explain different bottlenecks. We integrate existing display techniques and associate them with the appropriate types of bottlenecks. Since our search model identifies the type of bottleneck in a program, we can create useful displays. As a result, the user is freed from having to sort through a sea of pictures and tables trying to find one that will explain the

This work was supported in part by National Science Foundation grants CCR-8815928 and CCR-9100968, Office of Naval Research grant N00014-89-J-1222, and a grant from Sequent Computer Systems Inc.

performance of their program. While our search model was designed for dynamic instrumentation, it is also a useful paradigm for post-mortem analysis.

In Section 2 we describe the W^3 Search Model. In Section 3, we describe the prototype implementation of the Performance Consultant along with some results from running it on real programs. Next, we discuss how this work relates to previous performance tools. Finally, we outline our plans for a full scale implementation.

2. The W^3 Search Model

The potentially large amount of available performance data is a problem both for the user to navigate and for tools to collect. The W^3 Search Model addresses the user's information overload problem by providing a well defined, logical search model to assist them in finding performance bottlenecks. In this section, we describe our goals for such a search model and define the W^3 Search Model. Our model is based on answering three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. Finally, we explain how to automate this search.

The main goal of the W^3 Search Model is to work with long running applications that use large scale parallelism (hundreds if not thousands of processors). Many performance problems do not show up when applications are tried on toy data sets or a small number of processors. To help users to find real bottlenecks in real applications, we need to be able to run our tools on a full scale version of the program. We also need to minimize the performance impact of our system on the application so that we find existing bottlenecks and do not create new ones. Dynamic control of the instrumentation makes a wealth of performance data available while requiring that only a limited amount be collected at any given time.

We feel execution time searching is an appropriate approach because in long running programs, the interesting behaviors tend to last a relatively long time, so 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 we might miss it, 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.

Our search model is designed for an execution-time search; however we feel programmers will find searching for bottlenecks in a running program to be difficult and confusing. To make the system easier to use, we provide *post-mortem semantics.* This means that programmers can view their search as if it took place after the program had completed execution, and the data necessary to answer their performance questions has been collected. When it is not possible to maintain this illusion, the system informs the user.

We have two additional goals of usability and portability. Programmers should not have to write their application for a 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. The most that should be required is to re-compile an application. Finally, our 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.

To permit users to quickly find their bottleneck without having to look at extraneous details, our search model starts from a high level view and iteratively refines the detail about what is causing the program to perform poorly. Users can independently refine the ''why'', ''where'', and ''when'' of a program's performance. The search process can be thought as traveling from one point to another in a three dimensional space, and at each step we can move in any direction.

2.1. ''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 correspond to bottlenecks, for example, a program is synchronization bound. Tests are boolean functions that indicate if a program exhibits a specific performance behavior related to the hypothesis (e.g., more than 10% of the time is spent doing synchronization).

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: (1) excessive synchronization due to small work units, (2) high contention for a synchronization object, or (3) excessive time spent waiting for messages from other processes. We then test each of these hypotheses to see if any of them are true.

These dependence relationships between hypotheses define the search hierarchy for the ''why'' axis. One hypothesis can have other hypotheses as pre-conditions and anti-conditions. These relationships form a directed acyclic graph, and searching the ''why'' axis involves traversing this graph. Figure 1 shows a partial ''why'' axis hierarchy with the current hypothesis being that the application has a *HighIOBlockingTime* bottleneck. This hypothesis was reached by first concluding that an *IOBottleneck* exists in the program.

The hierarchical ''why'' axis is one of the unique features of our system. 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.

† 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.

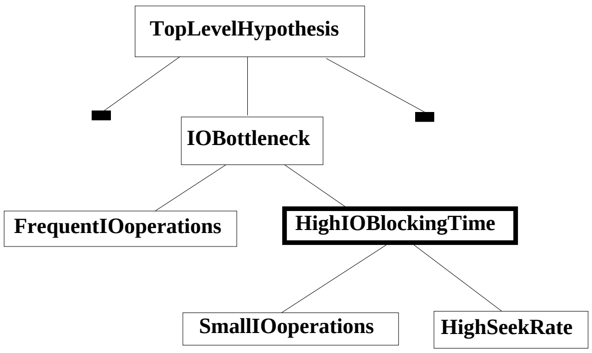

Figure 1. Sample ''Why'' (hypothesis) Hierarchy.

Our structured model helps to focus the user on the problem, not bury them with details.

2.2. ''Where'' Axis

The ''why'' axis is used to isolate the type of problem in a parallel application (e.g., a synchronization bottleneck). However, in a large application there might be many different synchronization operations. For the our search model to be useful, we must find which synchronization operation is causing the problem. To isolate a bottleneck to a specific resource (e.g., a procedure, or a lock), we search along the ''where'' axis.

The ''where'' axis is designed to be searched iteratively and has a hierarchical organization. It is separated into several different hierarchies, each representing a class of resources in a parallel application (e.g., synchronization objects, code, disks). There are multiple levels in each hierarchy, and the leaf nodes are the instances of the resources that are used by the application.

The left tree in Figure 2 shows a sample resource hierarchy. The root of the hierarchy is *Synchronization Objects*. The next level contains four types of synchronization (*Semaphore*, *Message Receive*, *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 Receive* 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. Part of the ''where'' axis is defined statically, part when the application starts, and part 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 hierarchy (resource class) is statically defined inside the tool. 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 machine and application are selected. This step creates nodes for the types of resources that the application might use. The types of nodes that get created depend on the type of machine used (e.g., shared memory vs. message passing) and the style of parallelism the application uses. The lowest levels in each hierarchy, representing specific resource instances, are added during the application's execution when the resource is first used.

Figure 2 shows a few sample resource class hierarchies, and the shape of each node indicates when it is defined. Oval nodes are statically defined. Triangle nodes are defined when the machine and application are selected. Rectangular nodes are created during the application's execution.

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 2 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 node).

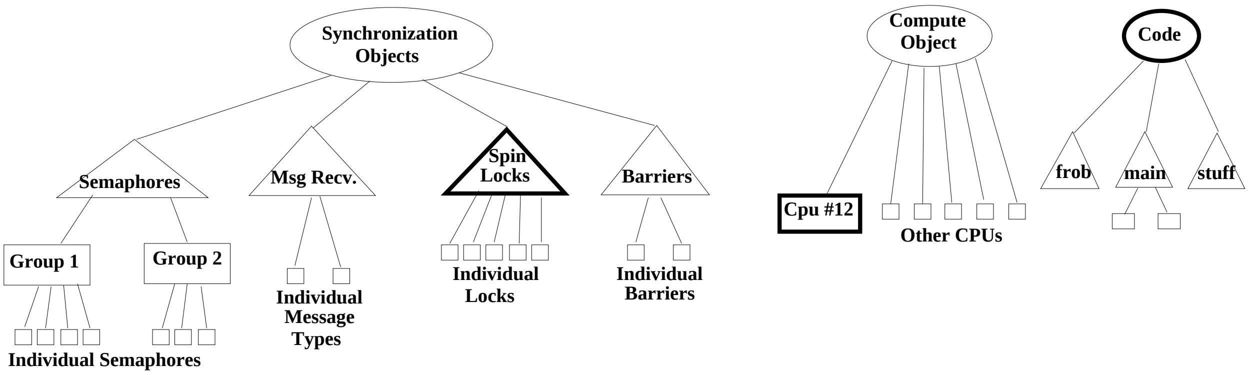

Figure 2. A sample ''where'' axis with three class hierarchies.
The highlighted object indicates the current focus. The oval objects are defined in the Performance Consultant. The triangles are static based on the application, and the rectangles are dynamically (runtime) identified.

Finally we test each 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 2. 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). Finally, we run the tests and conclude that the current hypothesis holds for *frob*. This results in the new focus all *Spin Locks* on *Cpu #12* in procedure *frob*. If the test was true for more than one procedure, they all would be added to the focus.

2.3. ''When'' Axis

The ''why'' and ''where'' axes isolate a performance problem to a specific type of bottleneck in a specific resource. However, the performance of parallel programs varies during different parts of its execution (i.e., the program goes through several phases). The purpose of the ''when'' axis is to isolate performance bottlenecks to specific time intervals during an execution. We first explain the ''when'' axis based on our post-mortem model, and then we describe how to approximate this model during execution.

Searching along the ''when'' axis involves testing the current hypotheses for the current focus (or focuses) for different intervals of time during the application's execution. We start by considering the entire execution-time of the program, and iteratively refine our search to smaller sub-intervals of time. For example, we might start by looking at an entire program, and then refine the bottleneck to an initialization interval, and finally isolate the problem to a sub-interval where data is being read into memory. Figure 3 shows a sample time-line for a program's execution, and a table showing the start and end of each interval.

Interval	Start Interval	End Interval	Description
i0	0	26	Entire Execution
i1	0	8	Initialization
i2	0	4	Read Data
i3	4	8	Init Nodes
i4	8	14	Compute1
i5	13	19	Exchange
i6	18	23	Compute2
i7	23	26	Output

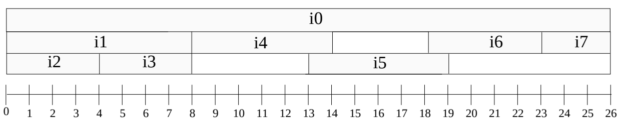

Figure 3. Intervals during a program's execution.

We represent the intervals of time along the ''when'' axis with a hierarchical structure with the root being the entire execution of the program. The children of a node are sub-intervals of that node. In addition, sub-intervals can overlap†. Figure 4 shows a ''when'' axis for a hypothetical application. The Initialization interval (*i0*) contains two sub-intervals Read Data (*i1*) and Init Nodes (*i2*). The fifth interval, Exchange, overlaps both of the compute intervals (*i4* & *i6*). A final interval, Output (*i7*), is at the end of the computation. We have restricted our search to *i1*, Initialization.

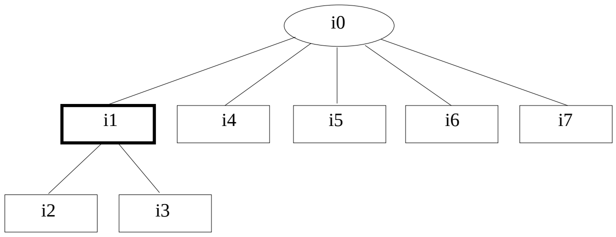

Figure 4. Sample ''When'' (time) Hierarchy.

Approximating post-mortem semantics is difficult because we do not have a fully built ''when'' hierarchy. We have part of the hierarchy for the time from when the program started until the current time, but only for those sub-intervals that the user defined and searched. Consider the sample time interval shown in Figure 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*.

Searching a time interval introduces four questions:

1.) How do we quickly and easily define the start of a new interval so we can start collecting data as soon as possible (i.e., where is *Start Interval*)?

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

3.) 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*)?

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

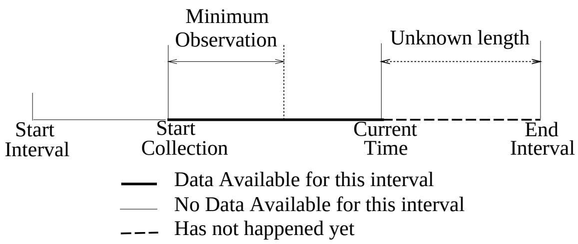

Figure 5. Potential problems with changing data collection during execution.

† This implies that a sub-interval might be contained in two different time intervals, and so the hierarchy is a DAG, not a tree.

The first question, defining the start of an interval, 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 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[18], and TOPSYS[3]) use similar predicates. 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. Our prototype implementation currently uses manual definition, but we will add trigger predicates in the future.

The second question is how long does an interval last. Our post-mortem model requires performance 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. 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, the data aggregation window grows with it. We also require a minimum observation time (shown as *Minimum Observation* in Figure 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.

There are two potential problems with using data for 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 the increasing window size) and might appear to be true after they become false. However, we stop aggregation at the end of a time interval and according to our definition of post-mortem semantics, we aggregate over the entire time interval. So momentum is not a problem because the window of aggregation is bounded by current time interval.

The third question is what to do when we start collecting data and testing hypotheses for a time interval after the interval has begun. Since our post-mortem model averages 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.

2.4. Automated Guidance

The axes of the W^3 Search Model help to direct the user to find performance bottlenecks; however at each step there are several choices of possible refinements to consider. 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. For example, a ''why'' axis hypothesis for a synchronization bottleneck might return a hint to refine along the ''where'' axis based on synchronization objects. Hints have two important characteristics. First, they are only hints and exist to order refinements, not alter what refinements are possible. Second, hints violate the orthogonality of the three axes because a hint associated with one axis can refer to refinements along other axes.

Hints are helpful for narrowing down the choices that the user has to make, but they still require the user to select a refinement to consider. 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 combinations of manual and automated searching are possible.

Selecting a refinement is a three step process: determining all possible refinements, ordering the refinements, and finally selecting one to try. The possible refinements are the children of the current nodes along each axis. We use hints 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.

An important question about automated refinement is how long to evaluate a hypothesis before concluding the test criteria are not met. This is an interesting problem because we are searching during the application's execution, and we want to be able to try several refinements before the current time interval ends. 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 conclude that no matter what happens in the remainder of the interval, the result will be the same.

3. Prototype Implementation

Because many of the techniques used by the W^3 Search Model are new, we implemented a prototype of the system to study the ability of our search model to identify performance bottlenecks using dynamic on-the-fly data selection. We also wanted to compare the amount of performance data generated by our method to existing trace based and sampling approaches. To validate the guidance supplied by the Performance Consultant, we also studied the application programs using the IPS-2[14] performance tools and compared results. We used our prototype to study three applications: two are from the Splash[17] benchmark suite and one is a database application.

3.1. Experimental Method

Our test implementation includes 15 hypotheses (shown in Figure 6) dealing with CPU, I/O, synchronization, and virtual memory bottlenecks. In a full implementation of our system, we plan to permit users to create and modify hypotheses and tests during program execution. However, to simplify our prototype implementation, we wrote hypotheses and tests as C++ functions compiled into the system. We also created 3 resource classes (code, process, and synchronization object). This means that generally we can 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 have also implemented a manual version of the ‘‘when’’ axis.

Since no available system provided the necessary infrastructure for dynamic control of the instrumentation, 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 our 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[11]. 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).

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

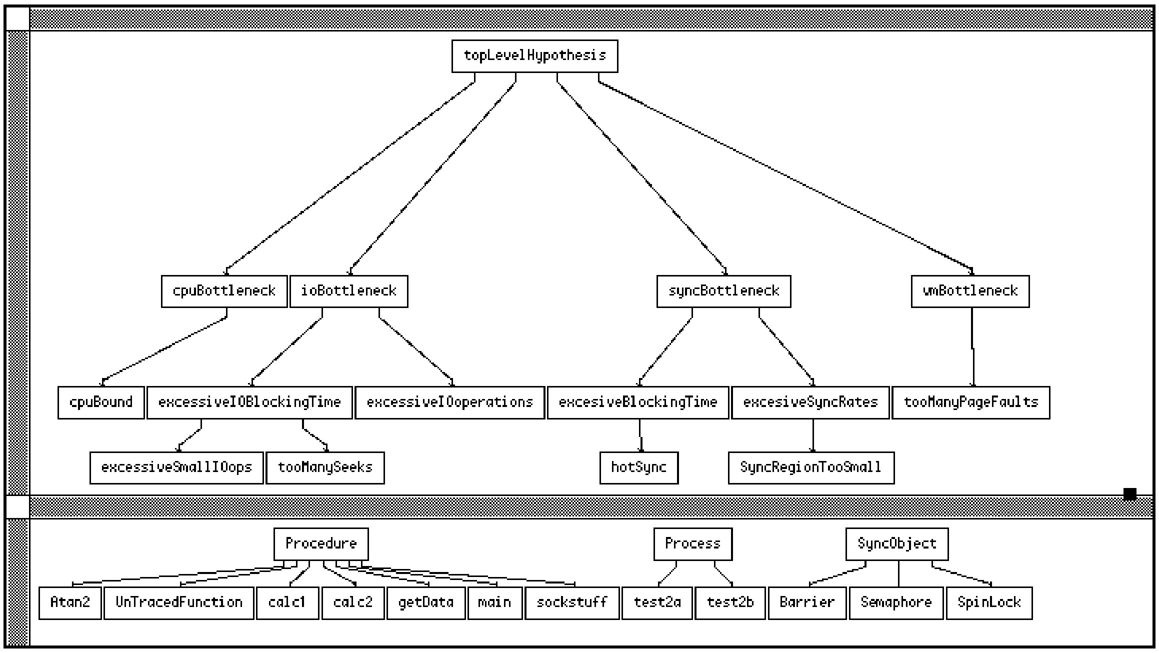

Figure 6. A Display showing the ‘‘why’’ and ‘‘where’’ axes.

data. When evaluating tests, we only looked at performance data for the time interval from when the data for that test was "enabled" until the current time. 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 from a running program. A counter records an event in the program at a specific focus (e.g., a procedure call counter exists for 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 which 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. We associate an explanation function 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[9] style profile table for the current focus along the ‘‘Where’’ axis.

Our prototype implementation has an X interface that allows users to navigate the three search axes. We also provide a command line interface to the explanations of the bottlenecks found. A screen dump from the interface appears in Figure 6. In the future we plan to expand our interface to incorporate realtime profile tables and visualizations of the performance data. The prototype implementation consists of 6,000 lines of C++.

Finally, a note about how we configured the Performance Consultant for our study. Since we were primarly 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). We plan to explore the impact of these parameters in the future.

3.2. Water

Water is 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 primarly 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 7 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.

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 7. Comparison of Full Sampling vs. Dynamic Instrumentation for the Water application.

Figure 8 compares the volume of sample data generated by our approach to the full tracing done by IPS-2. "Samples" is the number of samples used that also appeared in Figure 7. To compare dynamic instrumentation to IPS-2 tracing we needed to figure out how big a single sample from dynamic instrumentation would be. We feel that 12 bytes is a reasonable size (4 byes for a counter identifier, 4 bytes for a time stamp, and 4 bytes for the value). This value is shown in the column "Size". "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.

CPUs	Dynamic Inst. Samples	Size†	Trace Size	Ratio
4	1,390	0.02	1.2	73
16	28,692	0.34	13	38

Figure 8. Comparison of Dynamic Instrumentation vs. Tracing for Water (Size is in megabytes).

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 processors case (shown in Figure 7). 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 collect the data for each lock for the entire execution.

3.3. LocusRoute

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 this program on four processors. 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 a lot of 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 can also use our search 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. This early success with this idea of using our search model to test if our instrumentation significantly alters the performance was encouraging. We plan to expand this functionality in our full implementation.

Figures 9 and 10 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.

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 9. Comparison of Full Sampling vs. Dynamic Instrumentation for LocusRoute.

Version	Dynamic Inst. Samples	Size	Trace Size	Ratio
initial	1,050	0.01	9.7	770
collection hypothesis	4,226	0.05	9.7	191

Figure 10. Comparison of Dynamic Instruementation vs. Tracing for LocusRoute.

3.4. Shared Memory Join

The shared memory join application is an implementation of the join function for a relational database. It implements a hash-join algorithm[7] using shared memory for inter-process communication. 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 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 how even when our 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 feature of our work. To validate this result, we again used the IPS-2 performance tools. Since we had previously studied this program[11], 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. A few small changes to the program reduced this page fault behavior and improved the execution time by 10%.

Figures 11 and 12 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 10 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.

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 11. Comparison of Full Sampling vs. Dynamic Instrumentation for Shmjoin.

Sample Interval	Dynamic Inst. Samples	Size	Trace Size	Raio
100 msec.	8,379	0.10	4.1	41
20 msec.	41,958	0.51	6.3	13

Figure 12. Comparison of Dynamic Instrumentation vs. Tracing for Shmjoin.

3.5. Lessons Learned

Our prototype implementation has helped us to better understand the interactions between the different components in our system. 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 hot (highly contested) 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 plan to continue to study how people use manual refinement to improve automated refinement.

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. We need to be able to report this fact, and continue our search for the rest of the execution.

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

Finally, we gained confidence in our approach and are proceeding with a full implementation. We showed that it is possible to identify and isolate performance problems by using dynamic instrumentation and searching based on partial performance data. In addition, the reduction in the amount of performance data collected (typically a factor of 50 to 100) means that our approach will scale well to large machines. In fact, we feel that when we try it 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.

4. Related Work

Prior work in managing the complexity and volume of performance data has concentrated in three areas: performance metrics, visualization and data collection.

Performance metrics address the user side of the performance problem by reducing large volumes of performance data into single values or tables of values. Many metrics have been proposed for parallel programs: Critical Path[21], NPT[1], MTOOL[8], Gprof[9]. Each of these metrics can provide useful information; however in an earlier paper[12] we compared several of these metrics (and a few variations) and concluded that no single metric was optimal for all programs. However, we did discover several factors that can be used to help select appropriate metrics. For example, whether an application is well balanced (e.g., all processes do a similar amount of work) or does a large amount of synchronization influences which metrics are useful. The W^3 Search Model can be used to identify these characteristics in a program and help the user select an appropriate metric to use. We view our approach as a complement and an enhancement of performance metrics, not a replacement for them.

Another technique to manage the amount of performance data available to the user is visualization. Visualization presents large amounts of performance data in a graphical or aural way. The problem with most visualizations is that they are only useful for finding a specific type of bottleneck and so most tools provide a rich library of different visualizations. For example Paragraph[10] provides over twenty different visualizations and many of these displays can be configured to plot values for different resources (e.g., CPU and disk utilization). Unfortunately the user is left with the formidable task of selecting appropriate visualizations and resources to display. Our system improves this situation by being able to associate visualizations (and resources) with specific types of performance problems, and so we help the user to select useful visualizations to explain the performance problem.

Several approaches have been proposed to address the problem of how to efficiently collect performance data. One approach is to define a set of predicates that describe the interesting events in a program, and only collect data for those events that satisfy the predicate. EDL[2], ISSOS[16], and BEE[4] use this approach. The first two use a static set of predicates for an entire program's execution and lack the fine granularity of control of our approach. BEE permits dynamic control of the predicates, however, it does not provide any guidance of what predicates to select. Another approach is to build special hardware to collect performance data. The Sequent Symmetry[19], and the Cray Y-MP[5] provide a set of programable counters to collect performance data. However, since the systems can collect more data than they have counters, the user is left to select what to collect. In addition, not all interesting events are visible to hardware data collectors. Another approach used in MultiKron[15], TMP[20] and HYPERMON[13] is to build hardware that generates trace data and sends it to a data reduction node (or file). Hardware-assisted trace generation eliminates most of the perturbation of the CPU and inter-connection network. However, it makes it easy to generate so much data that it swamps any file system or data analysis station.

One system that tries to provide high level decision support about the performance of parallel programs is Atexpert[6] from Cray Research. It uses rules to recognize performance problems in Fortran programs. This tool solves a special case of the problem we address: Fortran programs that have been automatically parallelized by the compiler. In addition, it is a post-mortem tool that does not address how to reduce the volume of data collected.

5. Conclusions

We have presented a new approach, called the W^3 Search Model, for the design of performance tools that addresses the problems of how to efficiently collect performance data and how to provide users with useful guidance to find bottlenecks. We described a prototype implementation of the system, and presented a case study based on the prototype. These preliminary studies showed that our new technique can reduce the volume of performance data that needs to be collected by 1 to 2 orders of magnitude.

Based on the results of the prototype implementation, we have started to build a full scale implementation on the Thinking Machines CM-5, and plan to develop a version for the Intel Paragon in the near future. We are also creating the necessary infrastructure for dynamic instrumentation on these machines. In addition, we plan to enhance the graphical interface of the Performance Consultant.

References

1. T. E. Anderson and E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boston, May 1990, pp. 115-125.
2. 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.
3. 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 Munchen, July 1991.
4. B. Bruegge, "A Portable Platform for Distributed Event Environments", *Proc. of the 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.
5. UNICOS File Formats and Special Files Reference Manual, SR-2014 5.0, Cray Research Inc.
6. UNICOS Performance Utilities References Manual, SR-2040 6.0, Cray Research Inc.
7. D. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proc. of the 1985 VLDB Conference*, Stockholm, Sweden, August 1985, pp. 151-164.
8. A. J. Goldberg and J. L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL", *Proc. of Supercomputing'91* , Albuquerque, NM, Nov. 18-22, 1991, pp. 481-490.
9. 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.
10. M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software 8*, 5 (September 1991), pp. 29-39.
11. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool", *Proc. of the 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.
12. J. K. Hollingsworth and B. P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation", *Supercomputing 1992*, Minneapolis, MN, November 1992, pp. 4-13.
13. 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.
14. 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.
15. A. Mink, R. Carpenter, G. Nacht and J. Roberts, "Multiprocessor Performance Measurement Instrumentation", *IEEE Computer 23*, 9 (September 1990), pp. 63-75.
16. 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.
17. 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.
18. E. T. Smith, *Debugging Techniques for Communicating, Loosely-Coupled Processes*, PhD Thesis, University of Rochester, December 1981.
19. S. S. Thakkar, Personal Communication.
20. D. Wybranietz and D. Haban, "Monitoring and Performance Measuring Distributed Systems during Operation", *SIGMETRICS*, Santa Fe, New Mexico, May 1988, pp. 197-206.
21. 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.