An Adaptive Cost Model for Parallel Program Instrumentation

Jeffrey K. Hollingsworth†

Computer Science Department
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Barton P. Miller

University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
bart@cs.wisc.edu

Abstract

Software based instrumentation is frequently used to measure the performance of parallel and distributed programs. However, using software instrumentation can introduce serious perturbation of the program being measured. In this paper we present a new data collection cost system that provides programmers with feedback about the impact data collection is having on their application. In addition, we introduce a technique that permits programmers to define the perturbation their application can tolerate and then we are able to regulate the amount of instrumentation to ensure that threshold is not exceeded. We also describe an implementation of the cost model and presents results from using it to measure the instrumentation overhead for several real applications.

1. Introduction

Monitoring is critical to understanding the performance of an execution of a parallel or distributed application. For technical and economic reasons, software based monitoring is generally used to measure applications. However, software based monitoring introduces overhead into the application and can alter its performance. In this paper we present a new way to manage the perturbation caused by data collection. Our approach is based on an instrumentation cost system that ensures that data collection and analysis can be accomplished while controlling the performance overhead of the instrumentation. The unique feature of our approach is that it lets the programmer see and control the overhead introduced by monitoring rather than simply being subjected to it.

---

† Presenting and Corresponding author.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), NSF Grants CCR-9100968 and CDA-9024618, Department of Energy Grant DE-FG02-93ER25176, and Office of Naval Research Grant N00014-89-J-1222. Hollingsworth was supported in part by an ARPA Graduate Fellowship in High Performance Computing. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government

The best way to handle instrumentation overhead is to avoid introducing it in the first place. In a previous paper [3], we described a new approach to performance monitoring called *Dynamic Instrumentation*. Dynamic Instrumentation delays instrumenting an application program until it is in execution, permitting dynamic insertion and alteration of the instrumentation during program execution. This 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. The purpose of our cost system is to control the instrumentation overhead in an environment that uses Dynamic Instrumentation.

To manage the perturbation caused by instrumentation, we have developed an instrumentation cost system to ensure that data collection and analysis does not excessively alter the performance of the application being studied. 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.

By computing the predicted cost of instrumentation before data collection starts, it is possible to decide if the requested data is worth the cost of collection. For example, if the user requests performance data whose predicted cost of collection is 100% of the application's run-time, they might decide that the impact of collecting the data is too high to warrant collection. This predictive information can be used as feedback to reduce or defer an instrumentation request. Our higher-level performance analysis tools use the cost prediction to control how aggressively they instrument a program in search of performance bottlenecks. In many cases, control of instrumentation overhead can allow our tools to more quickly isolate a performance problem (examples of this situation are given in Section 7).

Although predicting the cost of data collection prior to instrumentation execution provides useful data, it is important to make sure that the actual cost of data collection matches the predicted cost. The observed cost tracks the impact the currently enabled instrumentation has on the application. By computing this value, we can verify that the actual impact of instrumentation is held within predefined limits. If the observed cost exceeds these limits, feedback is provided to the user or higher-level tool; this feedback allows us to dynamically maintain (approximately) a fixed level of instrumentation overhead.

Our two part cost model provides an effective way to not only measure the perturbation caused by instrumentation but also to regulate it. In the next section, we summarize our previous work in

Dynamic Instrumentation and automated control of the instrumentation using the W^3 Search Model. Section 3 introduces the cost model and describes how we use it to compute the perturbation due to instrumentation. In Section 4, we show how to use the cost model to control the level of instrumentation overhead. In Sections 5 and 6, we 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 running on a UNIX workstation and a Thinking Machines CM-5. In Section 7 we present a short case study of the effectiveness of higher level performance tools using our cost system. Section 8 discusses related work and conclusions are presented in Section 9.

2. Dynamic Instrumentation and W^3 Search Model

Since our cost system builds on our previous work in Dynamic Instrumentation and search tools for performance bottlenecks, we will briefly review that work here. Our recent work in performance monitoring tools has focused on two areas. First, how can we help programmers to understand the source of their performance problems rather than providing them raw performance data. Second, how can we efficiently collect performance data for large, long running applications. In this section, we summarize our work to address these two problems.

Data collection is a critical problem for any parallel program performance measurement system. To understand the performance of parallel programs, 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. A variety of different approaches have been tried to efficiently collect performance data. Two common approaches are event tracing and statistical sampling. Both of these techniques have limitations in either the volume of data they gather or granularity of data collected. We take a new approach to 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.

At any time during a program’s execution, a consumer of performance data (e.g., the Performance Consultant) can request that the Dynamic Instrumentation system start collecting a metric for a particular combination of resources. To satisfy this request, instrumentation code is generated and inserted into the application program. When a consumer of the performance data no longer needs the performance data, the instrumentation code is removed from the application program.

To meet the challenges of providing an efficient, yet detailed 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. The interface we developed is based on two abstractions: *resources* and *metrics*. Resources are similar to nodes in the ‘‘where’’ axis of the W^3 Search Model. Metrics are time varying functions that characterize some aspect of a parallel program’s performance. 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.

Dynamic instrumentation uses two types of ‘‘runtime code synthesis’’. First, a library of commonly used primitive functions is included in the application’s executable image. Second, small bits of code, called *trampolines* are generated and inserted into the application when collection of a specific metric and focus is requested. Two types of trampolines are used. First, there is a base trampoline for each point in the application with instrumented. Second, there is one mini-trampoline for each call to the instrumentation library.

The second topic in performance measurement that we have been investigating is to help programmers to make sense out of the collected performance data. The W^3 Search Model [2], is methodology that provides a structured way for programmers to quickly and precisely isolate a performance problem without having to examine a large amount of extraneous information. It is based on answering three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. By iteratively refining the answer to these three questions, we can precisely describe to programmers the reason their program is not performing as expected. To deliver answers rather than just posing the questions, we automate this search process.

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. 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 evaluate to true (e.g., more than 20% of the time is spent waiting for synchronization).

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. We must also find which synchronization operation is causing the problem. 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. Resource hierarchies include: synchronization objects, source code, threads, 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.

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. The third component of the W^3 Search Model Search Model is the ‘‘when’’ axis. The ‘‘when’’ axis 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.

A key component of the W^3 Search Model is its ability to automatically search for performance bottlenecks. This automation 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.

Dynamic instrumentation has been implemented as part of the Paradyn parallel program performance monitoring environment. An initial implementation of the W^3 Search Model model, called the Performance Consultant, has also been incorporated into Paradyn.

3. Cost Model

With Dynamic Instrumentation, the data collected at a particular point in the program no longer remains fixed for the entire program’s execution. Each time a new request for instrumentation is received, the instrumentation overhead for that point can change. In addition, different types of instrumentation requests can have decidedly different effects on a program's performance. Our model associates an instrumentation 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 can be used to estimate the overhead for the desired instrumentation. Observed cost is computed while the instrumentation executes and provides notification if the perturbation has exceeded the user's expectations. In essence, the Observed Cost Model is just another performance metric; the techniques for computing performance metrics using dynamic instrumentation and monitoring for performance problems using the W^3 Search Model described in the previous section are used to implement much of the model.

3.1. Predicted Cost

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 a Predicted Cost Model 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.

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 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 measurements of Dynamic Instrumentation [3], we know the precise cost of each instrumentation primitive and trampoline request. The difficult part is estimating the frequency of execution of each point.

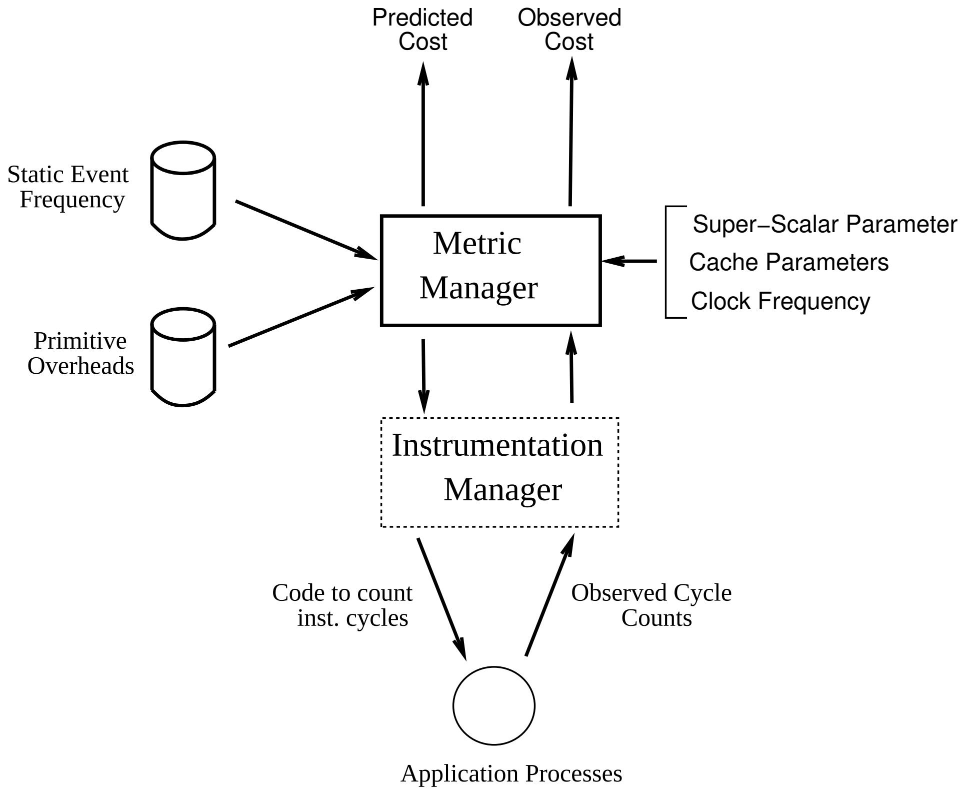

Figure 1. Computing Predicted and Observed Costs.

Our data about the execution frequency of points comes from a static model of procedure call frequency. This approach is at best a wild guess, but since we adjust this value based on runtime data the initial value does not need to be accurate. We associate with every point in the program an expected frequency. The value of each point is static and based on the 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 implement 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.

The total predicted CPU time perturbation is the sum of the predicted CPU time perturbation at each instrumentation point. We denote the predicted cost for the application as C_{pred}.

3.2. 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 is 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 goal of the Observed Cost Model is to verify this value. To compute the Observed Cost Model, we add a small amount of 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 1 shows how we use observed event execution frequency to compute the predicted cost. We denote the observed cost as C_{obs}.

Observed cost 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 (such as too much synchronization blocking time) and use the W^3 Search Model to look for it. In the W^3 Search Model, the observed cost 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.

4. Using Predicted Cost to Control Perturbation

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 that 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 a 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, then we start to consider refinements of that bottleneck. However, if after evaluating a set of hypotheses for a pre-defined time interval none of the hypotheses are true, 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 (potential performance problems) 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 tested; it simply changes when they get tested.

5. Implementation

We now describe an initial implementation of the cost model. We added the cost model to the Paradyn Parallel Performance Tools[2, 3]. Paradyn runs on a network of workstations (running PVM), or on parallel hardware such as the Thinking Machines CM-5. We also describe the results of a case study we conducted to measure the accuracy of both the observed and predicted cost models. This case study involved both sequential applications running on a UNIX workstation, and a parallel application running on a CM-5. We found that, for a wide variety of degrees of instrumentation inserted, the measured program running time with instrumentation was always within 5% of the original running time plus the observed cost estimate.

Our initial implementation of the cost model only includes the direct CPU cost of instrumentation. This is a reasonable approximation on machines such as the CM-5. The CM-5 uses gang scheduling and applications are context switched on each processor and the interconnection network at the same time. Because of the way Dynamic Instrumentation has been implemented on this machine, the instrumentation does not share a time quanta with the application and so there is no perturbation of the interconnection network.

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 for each instrumentation point (trampoline).

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, 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†.

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 to 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 framework of modeling instructions[6] 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 one cache miss per 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 existing facilities of Dynamic Instrumentation to constrain the metric to particular resource combinations within the application.

† 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 implementation of the Predicted Cost Model is based on using the same techniques to compute the cycle time of each trampoline that we 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. In the future, we plan to employ more sophisticated prediction techniques such as those developed by Wu and Larus[7].

6. Evaluation of the Cost System

Once we had implemented the observed cost system, 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[1], and one parallel application. For each test program program, we recorded three cost measures:

T_{no_inst}: the user CPU time needed to run the application program with no instrumentation, as measured by UNIX timing commands.

T_{obs}: the user CPU time of the application program with the dynamic instrumentation, as measured by UNIX timing commands.

C_{pred}: the cost of the dynamic instrumentation as calculated from our Predicted Cost Model (using known instrumentation costs and the event frequency estimates given at the end of Section 5).

C_{obs_time}: the cost of the instrumentation, calculated as $T_{obs} - T_{no\_inst}$.

C_{obs_cycles}: the cost of the dynamic instrumentation as calculated by recording the number of machine cycles actually executed in instrumentation.

Any difference between the C_{obs_time} and C_{obs_cycles} represents the inaccuracies in our calculations of the observed cost. Differences between C_{obs_time} and C_{pred} represent the inaccuracies in our calculations of the predicted cost. Accurate calculation of observed cost is crucial; accurate calculation of predicted cost is much less important.

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: This 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: Procedure profiling consisted of turning on CPU metrics for each user supplied procedure in the application. This is similar to the UNIX utility prof.

PC Base: The Performance Consultant base case enabled the initial instrumentation used by the Performance Consultant to search for a bottleneck in the application.

PC Full: 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 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 were unable to measurably perturbed it. The programs were run on an idle SPARCstation 10 running at 40Mhz.

6.1. Observed Cost

The results for the first application, Ear, are shown at the top of Figure 2. This program consists of 16 files containing 93 functions. T_{no_inst} for this program is just over ten minutes, and averages 11,000 procedure calls per second during its execution. The results shown in the table show that the cycle-counted observed cost is within 2% of the timed observed cost. The total instrumentation overhead (C_{obs_time}) ranged from 1.6% to just over 10% of the CPU time of the un-instrumented version (T_{no_inst}).

The second program we measured was Fpppp, a quantum chemistry benchmark which does electron integral derivatives. It consists of 13 files and 13 procedures. T_{no_inst} for this program is just under 5 minutes with an average of 2,100 procedure calls per second. The timed observed cost for this program ranges from 4 to 16%. The results also show that the error in cycle-counted observed cost ($C_{obs\_time} - C_{obs\_cycles}$) is less than 4% of the base CPU time of the application.

The third program is Doduc, a Monte Carlo simulation of the time evolution of a thermohydraulical model of a nuclear reactor. It is composed of 41 files and 41 procedures. This program averages 107,000 procedure calls per second. T_{no_inst} for this program is just over two minutes. The

Version	Time	C_{obs_time} Time	C_{obs_time} Percent	C_{obs_cycles} Time	C_{obs_cycles} Percent	$C_{obs\_time} - C_{obs\_cycles}$ (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 2. Cycle-counted vs. Timed Observed Overhead.

This table shows the difference between the cycle-count estimates of observed cost (C_{obs_cycles}) and the actual timed values (C_{obs_time}). The percentages are relative to the uninstrumented CPU time (T_{no_inst}). **All times are shown in seconds.**

timed observed cost for this program ranged from 1 to 61 percent of the T_{no_inst}. This program has the largest error in the observed cost metric, which happens on the Performance Consultant full search case. The cycle-counted observed cost and the time observed cost diverge because of 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 all of components of this overhead. We are continuing to investigate.

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 cycle-counted observed cost and time observed cost for this program appears in Figure 3. T_{no_inst} for the program as 216 seconds. On the CM-5, 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. The difference between the cycle-counted observed cost and the timed observed cost ranges from 0.5% to 2.6%.

Version	Time	C_{obs_time} Time	C_{obs_time} Percent	C_{obs_cycles} Time	C_{obs_cycles} Percent	$C_{obs\_time} - C_{obs\_cycles}$ (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 3. Cycle-counted vs. Timed Observed Overhead for a Parallel Application. This table shows the difference between the timed observed cost (C_{obs_time}) and the cycle-counted observed cost (C_{obs_cycles}) for the parallel graph coloring application. The percentages are relative to the uninstrumented CPU time (T_{no_inst}). **All times are shown in seconds.**

6.2. Predicted Cost

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 4. The errors in three of the four cases are within 10% of T_{no_inst} for the program. However, for the procedure call case, the predicted cost 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.

Version	C_{obs_time} Time	C_{obs_time} Percent	C_{pred} Time	C_{pred} Percent	$C_{obs\_time} - C_{pred}$ (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 4. Timed Observed Costs vs. Predicted Costs.

This table compares the timed observed costs (C_{obs_time} and the value of the Predicted Cost Model (C_{pred}) for the three sequential applications. The percentages are relative to the uninstrumented CPU time (T_{no_inst}). **All times are shown in seconds.**

The middle section of Figure 4 shows the predicted cost for the Fpppp application. For all cases, the estimated cost was within 11% of T_{no_inst}. In addition, the largest error was again for the all procedure profiling case.

The last part of Figure 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 timed observed cost for the parallel graph coloring program. Figure 5 shows the results of this case. The case is interesting because the predicted cost is much closer to the observed 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 heuristics. 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.

Version	C_{obs_time} Time		C_{pred} Percent		$C_{obs\_time} - C_{pred}$ (Percent)
RBase	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 5. Timed Observed Cost vs. Predicted Cost for a Parallel Application.
This table compares the timed observed costs (C_{obs_time}) and the value of the Predicted Cost Model (C_{pred}) for the graph coloring application. The percentages are relative to the uninstrumented CPU time (T_{no_inst}). **All times are shown in seconds.**

7. Using Cost to Control Searching

The current principal use of our cost system is to control hypothesis evaluation in the Performance Consultant, so we were interested in quantifying how well our cost system could regulate the perturbation in this environment. To study this use of our cost system, we conducted several experiments that compared searching for bottlenecks with (1) a cost system that controlled how many hypotheses are evaluated simultaneously, (2) a fixed limit on the number of hypotheses that are evaluated at once, and (3) an unlimited number of hypotheses being evaluated. For each application, we ran the Performance Consultant three times. The first time was with a cost limit of 10% perturbation, the second time for a fixed limit of three refinements to the current hypothesis, and third with no limit on the refinements to the current hypothesis. The limit of three refinements was intended to provide a comparison to an alternative strategy for controlling the cost of data collection. The unlimited case was to measure the worst case impact of instrumentation for each application.

We were interested in evaluating two criteria about the effectiveness of our search system. First, we wanted to verify that the instrumentation cost was held within the cost limit set by the user (10% in this case). Second, we were interested in comparing how quickly a performance problem could be isolated using each method.

For each run of an application, we compared the bottlenecks identified by the Performance Consultant. For the Ear application, the same performance bottleneck was found for all three cases. The order in which the refinements were considered was slightly different, but the conclusion was the same. For the Doduc application, the same performance bottleneck was found in the 10% limit and three hypothesis limit, but the perturbation was so high in the unlimited case that no bottleneck was identified. For the Fpppp application, the hypothesis limit and unlimited cases identified one procedure as the bottleneck and the cost limit identified another procedure. A CPU time profile for the application showed that both procedures consumed enough CPU time to be flagged as bottlenecks according the thresholds.

The results for the three serial applications Doduc, Ear, and Fpppp are shown in Figure 6. The Time column reports the amount of elapsed time required for the Performance Consultant to execute its search. For all the applications, the time required for the search was least when cost was used to regulate hypothesis evaluation. The improvement in search time ranged from 29% for Fpppp to 71% faster for Ear when compared to the limit of three hypotheses. The cost based limit was able to evaluate the available hypotheses faster because different hypotheses have different costs and the cost based limit permitted evaluation of more hypotheses simultaneously, while keeping the overhead within the limit. The cost based limit was able to identify a problem faster than the unlimited search case because it saved time by not generating and inserting instrumentation for each of the possible refinements.

Application Control Method	Time	Avg C_{obs_cycles}	Max C_{obs_cycles}	Variance C_{obs_cycles}
Doduc				
10%	48.2	3.1%	8.6%	5.1
3 Hypotheses	77.1	3.1%	5.6%	2.9
unlimited	258.7†	6.2%	41.6%	87.7
Fpppp				
10%	52.0	1.2%	3.7%	0.9
3 Hypotheses	73.2	0.5%	1.3%	0.1
unlimited	227.2	1.9%	3.7%	0.2
Ear				
10%	37.9	2.7%	8.1%	11.2
3 Hypotheses	130.8	5.1%	17.5%	40.2
unlimited	226.5	15.4%	17.2%	12.2

Figure 6. Summary of fixed vs. cost based hypothesis evaluation.

The table summarizes the differences between using a 10% limit on the cost, a fixed limit of three hypotheses considered at once, and no cost control mechanism. The second column shows the amount of elapsed time required to execute the search. The third column shows the average CPU time perturbation as reported by C_{obs_cycles}. The fourth column shows the maximum CPU time perturbation reported by C_{obs_cycles}. The fifth column shows the variance of C_{obs_cycles}.

We verified that the cost control mechanism maintained instrumentation overhead within the target limits. The last three columns of table in Figure 6 summarize the cost of data collection for each application and approach. For all three applications, using cost to control hypothesis evaluation

† The application finished execution before the search finished in this case.

held the CPU time perturbation under the user defined limit of 10%. In two cases the peak CPU perturbation was higher for the cost based limit than for the 3 hypothesis limit, although it was still less than the 10% threshold. For these applications, it was possible to evaluate more than three hypotheses without having a major impact on the application. Finally, for the unlimited evaluation case the the peak value of C_{obs_cycles} ranged from 3.7% to over 40%. The wide range of cost for the unlimited case shows that trying to evaluate all hypothesis at once can have a wildly different impact depending on the application. Likewise, since the cost of evaluating a hypothesis depends on the hypothesis and the application, regulating instrumentation based on a limit on how many hypotheses are evaluated at once can result in higher costs than desired or longer searching than necessary.

8. Related Work

A related topic to our two part cost model is the work that has been done on perturbation compensation[4, 8]. The goal of perturbation compensation is to reconstruct the performance of an unperturbed 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.

A second area of related work is Pablo's [5] adaptive instrumentation system. In Pablo, the programmer specifies the events to be recorded in an event log for post mortem analysis. However, if during the program's execution, the volume of data collected exceeds certain thresholds, the system will fall back from producing event logs to producing summary information. If the amount of data being collected is still too high, even the summary information will be disabled. The approach used in Pablo leaves the underlying instrumentation in place and controls the logging of data. However, our technique has the advantage that with Dynamic Instrumentation, disabling data collection completely removes the instrumentation code and so there is no latent perturbation due to instrumentation code that is disabled but must execute a few instructions to learn that it is disabled. Also since we control the number of hypotheses being evaluated (and hence the amount of instrumentation) rather than changing what data gets collected, the type of the data gathered remains constant no matter the perturbation.

9. Conclusions and Future Work

Our cost model controls software instrumentation overhead based on feedback. This feedback correlates the high-level instrumentation abstractions with resource limits such as percent CPU overhead. We predict the amount of overhead we will cause and then use our instrumentation facility to provide information about the actual costs. The mechanisms that we have built as part of the Paradyn Parallel Performance Tool give the programmer direct control over their instrumentation. Instrumentation data is not discarded to reduce overhead, nor is it buffered. Instead, the generation and insertion of the instrumentation is deferred. We expose the overhead of data collection as a first class metric in Paradyn. The programmer is also given explicit control of the overhead, which controls the rate at which the performance tool searches for bottlenecks.

The W^3 Search Model is a natural fit with the cost model. We have implemented the cost model in Paradyn and demonstrated that we could accurately track the cost of data collection in several applications.

We are working to extend this research in several directions. First, we plan to incorporate additional resources besides CPU time into our cost model. Second, we would like to take advantage of more sophisticated and accurate models for caches and super-scalar processors.

References

1. ‘‘System Performance Evaluation Cooperative,’’ *Capacity Management Review*, vol. 21, no. 8, pp. 4-12, Aug. 1993.
2. Jeffrey K. Hollingsworth and Barton P. Miller, ‘‘Dynamic Control of Performance Monitoring on Large Scale Parallel Systems,’’ *7th ACM International Conf. on Supercomputing*, pp. 185-194, Tokyo, Japan, July 1993.
3. Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille, ‘‘Dynamic Program Instrumentation for Scalable Performance Tools,’’ *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., 1994.
4. Al Malony, *Performance Observability*, PhD Dissertation, Department of Computer Science, University of Illinois, OCT 1990.
5. Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera, ‘‘Scalable Performance Analysis: The Pablo Performance Analysis Environment,’’ in *Scalable Parallel Libraries Conference*, ed. Anthony Skjellum, IEEE Computer Society, 1993.
6. Ko-Yang Wang, ‘‘Precise Compile-Time Performance Prediction of Superscalar-Based Computers,’’ *ACM SIGPLAN’94 Conf. on Programming Language Design and Implementation*, pp. 73-84, Orlando, FL, June 20-24, 1994.
7. Youfeng Wu and J. R. Larus, ‘‘Static Branch Frequency and Program Profile Analysis,’’ *27th IEEE/ACM Inter’l Symposium on Microarchitecture (to appear)*, Nov. 1994.
8. Jerry 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.