Using cost to control instrumentation overhead

Jeffrey K. Hollingswortha,*, Barton P. Millerb

Abstract

We present a new data collection cost system that provides programmers with feedback about the impact data collection is having on their application. We allow programmers to define the level of perturbation their application can tolerate, and then regulate the amount of instrumentation to ensure that the threshold is not exceeded. Our approach is unique in that the type of data gathered remains constant; instead, we regulate when data are collected. This permits programmers to trade speed of isolation of a performance problem for less application perturbation. We implemented this cost system in the Paradyn Performance Tools and present several case studies demonstrating the accuracy of the cost system. © 1998—Elsevier Science B.V. All rights reserved

Keywords: Parallel program instrumentation; Perturbation; Overhead; Data collection; Paradyn performance tools

1. Introduction

Monitoring is critical to understand the performance of an execution of a parallel or distributed application. For technical and economic reasons, software-based monitoring generally is 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

*Corresponding author. E-mail: hollings@cs.umd.edu.

†Supported in part by Wright Laboratory Avionics Directorate (WLAD), Air Force Material Command, USAF, grant F33615-94-1-1525 (ARPA order B550), NSF Grants CCR-9100968 and CDA-9024618, DOE Grant DE-FG02-93ER25176, and ONR Grant N00014-89-J-1222. The US 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 WLAD or the US Government.

0304-3975/98/$19.00 © 1998—Elsevier Science B.V. All rights reserved

PII S0304-3975(96)00203-X
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.

The best way to handle instrumentation overhead is to avoid introducing it. In a previous paper [5], 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 do not excessively alter the performance of the application being studied. The system associates a cost with different resources. Possible 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 6). In the extreme, effective use of our perturbation budget means that a performance problem can be isolated with fewer executions of the program.2

Although predicting the cost of data collection prior to instrumentation execution provides useful data, it is important to ensure 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. To be useful, our observed cost system needs to be inexpensive to compute and to accurately reflect the true impact of data collection. By computing the observed cost, we can verify that the actual impact of

---

2 We are currently extending Paradyn to provide automated support for experiment management. This will permit the synthesis of results from multiple application program executions.
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 system provides an effective way to measure not only 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 system and describes how we use it to compute the perturbation due to instrumentation. In Section 4, we present results that show how accurately our cost system predicts and measures the actual perturbation for several applications running on UNIX workstation clusters. In Section 5, we show how to use the cost system to control the level of instrumentation overhead. In Section 6, we present a short case study of the effectiveness of higher-level performance tools using our cost system. Section 7 discusses related work and conclusions are presented in Section 8.

2. Dynamic instrumentation and $W^3$ search model

Our recent work in performance monitoring tools has focused on two areas: efficiently collecting performance data for large, long-running applications and helping programmers to understand the source of their performance problems rather than providing them raw performance data.

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. Our approach to data collection, called dynamic instrumentation, 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., a visualization or analysis module) can request that the dynamic instrumentation system starts 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.

Dynamic instrumentation is designed to be usable for a variety of high-level tools, and so it has a simple interface. The interface is based on two abstractions: resources and metrics. Resources are the objects about which we gather performance information. Typical resources include processors, interconnection networks, processes,
procedures, and synchronization objects. 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.

To implement dynamic instrumentation, we use “runtime code synthesis” to insert instrumentation primitives at specific points in the application. Typical primitives include timer operations and incrementing event counters. Points are locations in the application program, such as a subroutine call statement, where calls to primitives may be inserted. The specific points and primitives in use at any time depend on the combination of metrics and resources requested by the higher level tools.

We have been also investigating how to help programmers make sense of the collected performance data. The W3 search model [4], is a 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 why their program is not performing as expected. Refining the answer to these questions requires testing different hypotheses about the source of performance problems. To deliver answers rather than just posing the questions, we automate this search process. In an automated search, the tool refines the answers to these three questions by enabling and disabling the collection of performance data.

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 Boolean functions that indicate if a program exhibits a specific performance problem. For example, a synchronization bottleneck might be defined as any application with more than 20% of the application’s time spent waiting for synchronization operations to complete.

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. Components of the “where” axis include: synchronization objects, source code, threads, processes, processors, and disks.

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. As a result, decomposing a program’s execution into phases provides a convenient way for programmers to understand the performance
of their program. The “when” axis is a way for programmers to exploit the phase
behavior of their programs to find performance bottlenecks.

Dynamic instrumentation has been implemented as part of the Paradyn [7] parallel
program performance monitoring environment. An initial implementation of the
W³ search model, called the Performance Consultant, has also been incorporated into
Paradyn.

3. Cost system

With dynamic instrumentation, the data collected at a particular point in the
program no longer remain 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 decided-
dly different effects on a program’s performance. Our system associates an instrumenta-
tion cost with different resources. Possible resources include processors, interconnec-
tion 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. This approach is at best a wild guess, but we can adjust this
value based on run-time data, therefore, the initial value does not need to be accurate.
Observed cost is computed while the instrumentation executes and provides notifica-
tion if the perturbation has exceeded the user’s expectations. In essence, the observed
cost is just another performance metric; the techniques for computing performance
metrics using dynamic instrumentation and monitoring for performance problems
using the W³ search model described in the previous section are used to implement
much of the system.

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 system 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 system.

Predicted cost is the expected overhead of collecting the data necessary to compute
a metric for a particular 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 measured resource in the system required to collect the desired data.

To make the system 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 the instrumentation primitives. Third, we need to know the frequency of execution of that point. 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. Based on the measurements of dynamic instrumentation, we know the precise cost of each instrumentation primitive. The difficult part is estimating the frequency of execution of each point.

Data about the execution frequency of points come from a static estimate of procedure call frequency. This approach is at best a wild guess, but since we adjust this value based on run-time 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 system 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. In the future, we plan to employ more sophisticated prediction techniques such as those developed by Wu and Larus [11].

To compute the total predicted CPU time perturbation, we add up the predicted CPU time cost for each instrumentation point. We denote the predicted cost for the application as $C_{pred}$.

3.2. Observed cost

Predicted cost is based on an estimate of how much overhead of the enabled instrumentation should have on a program's execution. However, it is always a good idea to check that the system matches reality. That is where the observed cost is used. The observed cost monitors 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. Also, if the predicted and observed costs differ significantly, our system can adjust the amount of instrumentation enabled.

Actual cost also might differ from predicted cost because resource contention between the application and the data collection can affect cost. For example, the predicted cost system does not include the memory hierarchy (e.g., caches and TLB). If the application is not constrained by the memory hierarchy then the impact on the performance will likely be minimal. 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. Because perturbation effects are difficult to fully predict a priori, measuring instrumentation is vital.
We now consider how to compute the observed cost of executing our instrumentation primitives. The primary cost is the time required to execute primitives; we call this the direct cost. A second cost is the effect of executing the instrumentation code on the memory hierarchy (e.g., caches). Instrumentation can displace the application's data and instructions from the caches. We call this cache pollution. Cache pollution can cause additional overhead to re-load the displaced items.

In our current implementation of the cost system, we only include the direct CPU and cache-pollution costs. This is a reasonable approximation on MPP machines such as the CM-5. For example, the CM-5 uses gang scheduling where applications are context switched onto processors and the interconnection network at the same time. Dynamic instrumentation extracts performance data from compute nodes in a way that does not compete with the application for the interconnection network. Networks of workstations use a distributed scheduling scheme that makes it harder to isolate the overhead of moving performance data from compute nodes to monitoring stations. However, the data volume required by our system is low (the maximum rate of data transferred in our study was 4 K bytes per second). In addition, we currently consider only first-order perturbation and do not consider the impact of potential event re-ordering due to perturbation.

Conceptually, computing the cost of executing the instrumentation is easy: we simply record the time spent on executing the primitives. However, computing the cost of cache pollution is problematic. The difficulty in computing the cost of cache pollution is that it is impossible, without sophisticated hardware instrumentation, to know if cache lines used by the instrumentation will cause subsequent cache misses for the application. However, we can compute bounds for the impact of cache pollution. The lower bound on cache pollution is that there was no cache pollution. This happens when none of cache items are replaced due to instrumentation were subsequently used by the application. An upper bound is that every cache item loaded by the instrumentation code will result in a subsequent cache miss for the application.\(^3\)

Our observed cost has two values reflecting the lower and upper bound of the cache pollution. The actual cost of instrumentation should lie within this range. To compute the observed cost range we use two values:

\[ C_{\text{obs-direct}}: \] The measured time spent executing the instrumentation.

\[ C_{\text{obs-cache}}: \] The time spent waiting for cache misses while executing instrumentation code.

This value is used as an upper bound on the cache pollution.

We then compute the lower and upper bounds for the observed cost as

\[ C_{\text{obs-low}} = C_{\text{obs-direct}}, \]

\[ C_{\text{obs-high}} = C_{\text{obs-direct}} + C_{\text{obs-cache}}. \]

\(^3\) Although this one-to-one ratio is the worst case for direct mapped caches, for set-associative caches, the worst case cache pollution penalty is also a function of the associativity.
Since $C_{\text{obs\_direct}}$ reports the time consumed by instrumentation, the simplest way to implement it would be to add additional code 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 an inexpensive, but relatively accurate, way to compute the cost of our instrumentation.

To efficiently compute $C_{\text{obs\_direct}}$ we use statistical sampling. This is implemented using UNIX profiling that records a histogram of the distribution of time spent in different code regions. Since all of our instrumentation code is either generated at run time or located in a well-defined set of instrumentation subroutines, it is easy to identify the components of the histogram that represent time spent executing instrumentation. We then compute the fraction of the program's execution time spent executing instrumentation.

To compute $C_{\text{obs\_cache}}$, we use one more value:

$C_{\text{obs\_ideal}}$: The time required to execute the instrumentation assuming an ideal-memory model where all memory requests are satisfied by the cache.

Any difference between these measured and ideal times to execute the instrumentation code is due to the memory hierarchy. So:

$$C_{\text{obs\_cache}} = C_{\text{obs\_direct}} - C_{\text{ideal}}.$$  

To compute $C_{\text{obs\_ideal}}$, we added an additional instruction at each instrumentation primitive to record the number of machine cycles required to execute the primitives at that point. Since there are a small number of primitives, their performance can be computed once for each platform. There is also a small amount of code generated to call each primitive; we also record this time. The cycle count provides a precise measure of the number of instructions that are executed for instrumentation. However, we still need to convert instruction counts to time. For the SPARC processors used in this case study, we can simply divide the cycle times of the instrumentation instruction sequences by the clock frequency of the machine.\(^4\)

We implemented our cost system in the Paradyn parallel performance measurement tools. Integrating the implementation of the observed cost with dynamic instrumentation was not difficult. Since our instrumentation system was designed to support reading external sources of performance data (e.g., hardware and OS counters), we simply treated the observed cost as metrics and used the normal dynamic instrumentation mechanisms to report this data to the higher-level

\(^4\)For super-scalar processors a more sophisticated approach will be required. Although super-scalar processors can issue more than one instruction per machine cycle, dependencies between the operands of instructions mean that rarely can instructions be issued at the maximum rate. For these machines we need to analyze the instrumentation instruction sequences to more precisely estimate the number of cycles required for each instrumentation block. To do this, we could use Wang's [10] framework of modeling instructions by their functional unit requirements to get a more accurate estimate.
consumers of performance data. By treating this observed cost as a normal metric, we can use the existing facilities of dynamic instrumentation to constrain it to a particular resource combinations within the application.

Once we have computed observed cost, what should we do with it? Displaying the value to the user provides them with some idea of the impact of the instrumentation system on their application. There are, however, other ways to use this information. Observed cost can also be viewed as another performance metric to characterize the type of bottleneck in a parallel program. The only difference is that the bottleneck in which we are interested was created by the data collection system rather than the programmer. Our $W^3$ search model provides a way of isolating bottlenecks in parallel programs. We treat instrumentation as a 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. Evaluation of the cost system

We evaluated our two-part cost system by running applications programs and comparing the values of the predicted and observed costs to the actual perturbation. We ran three sequential and three parallel applications. For each program, we measured its performance with four different levels of instrumentation enabled: Base, Procedure, PC Base, and PC Full.

*Base*: The minimum amount of instrumentation is 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*: CPU time metrics for each user supplied procedure are inserted into the application for the lifetime of the application. This is similar to the UNIX utility proof.

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

*PC Full*: The Performance Consultant is run in fully automated mode, turning instrumentation on and off as needed.

Since we were interested in assessing the accuracy of the cost system, we did not want to use the cost system to control the number of refinements being considered. However, we also did not want to overwhelm the application with instrumentation by enabling all refinements at once. As a compromise, we configured the Performance Consultant to consider 10 refinements at once.

For each of the four levels of instrumentation, we recorded $C_{\text{obs-low}}$, $C_{\text{obs-high}}$, and $C_{\text{pred}}$. In addition we recorded two additional values:
The user CPU time of the application program with the dynamic instrumentation, as measured by UNIX timing commands.

The timed cost of the instrumentation, calculated as the difference between $T_{\text{obs}}$ for the current level of instrumentation and $T_{\text{obs}}$ for the base level of instrumentation.

The range between $C_{\text{obs,low}}$ and $C_{\text{obs,high}}$ indicates the bounds of the instrumentation overhead. If $C_{\text{observed}}$ is inside this range, our system accurately computed the instrumentation overhead. Differences between $C_{\text{observed}}$ and $C_{\text{pred}}$ represent the inaccuracies in our calculations of the predicted cost. Accurate calculation of observed cost is crucial; accurate calculation of predicted cost is less critical since it can be corrected by feedback from the observed cost system.

The three sequential applications we measured (Ear, Fpppp, and Doduc) are from the floating point SPEC92 [9] benchmark suite. These 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 size and procedure call frequency. We also measured another program, Tomcatv, to represent the low end of procedure call frequency but were unable to measurably perturb it, so it is not reported here. The programs were run on an otherwise idle SPARC station 5 running at 85 MHz.

Three parallel applications were used. The programs are PVM versions of the computational fluid dynamics (CFD) kernels from the NAS parallel benchmarks. They were selected to provide a sample of applications running on a cluster of workstations.

4.1. Observed cost

The results for the first sequential application, Ear, are shown at the top of Fig. 1. The base time for this program is about 11.5 min, and averages 11 000 procedure calls per second during its execution. The values in the table show that the measured observed cost is within the range between $C_{\text{obs,low}}$ and $C_{\text{obs,high}}$. The total instrumentation overhead ($C_{\text{observed}}$) ranged from 9% to just over 42% of the CPU time of the base time to run the program in Paradyn.

The Performance Consultant overhead times are bit larger than we expected. We investigated why, and discovered that the (currently) naive code generator for dynamic instrumentation was inserting duplicate copies of primitives at the same point to satisfy different instrumentation requests. We are working to add an optimizer to dynamic instrumentation to prevent this situation.

The second program we measured was Fpppp, a quantum chemistry benchmark which does electron integral derivatives. The base running time for this program is just less than 5 min with an average of 2 100 procedure calls per second. The timed observed cost for this program ranges from 5% to 7%. The results also show that all of the times are within the range between $C_{\text{obs,low}}$ and $C_{\text{obs,high}}$. 
The third program is Doduc, a Monte Carlo simulation of the time evolution of a thermo-hydraulical model of a nuclear reactor. This program averages 107,000 procedure calls per second. The uninstrumented running time of this program is about 1 min. The timed observed cost for this program ranged from 5% to 89% of the base time to run the program using Paradyn. This program has the largest difference between $C_{\text{obs}_{-}\text{low}}$ and $C_{\text{obs}_{-}\text{high}}$. This is an indication that the program is sensitive to cache perturbation.

Next, we tested our cost system with several applications running on a network of workstations. We selected the three NAS computational fluid dynamics (CFD) benchmarks [1]. The applications were run on SPARCstations 5's connected by Ethernet. PVM [2] was used as the parallel-programming model. The applications were configured with one master and four worker processes. For each of the three PVM applications, we ran the programs with the same four levels of instrumentation (Base, Procedure, PC Base, and PC Full) that we used for the three previous applications.

A comparison of the low and high values of the observed cost and measured perturbation for these program appears in Fig. 2. The time column shows the total time of all processes. $C_{\text{observed}}$ ranged from 1.8% to 12.7% for these three programs. For all of the programs at all levels of instrumentation, the value of $C_{\text{observed}}$ was in the range from $C_{\text{obs}_{-}\text{low}}$ to $C_{\text{obs}_{-}\text{high}}$.

Overall, the observed cost provides accurate feedback about the cost of instrumentation. For both the sequential and parallel applications measured, the value of $C_{\text{observed}}$ lies between $C_{\text{obs}_{-}\text{low}}$ and $C_{\text{obs}_{-}\text{high}}$. 

---

### Table 1: Application Costs

<table>
<thead>
<tr>
<th>Application</th>
<th>$C_{\text{observed}}$</th>
<th>$C_{\text{obs}_{-}\text{low}}$</th>
<th>$C_{\text{obs}_{-}\text{high}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ear</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>687.9</td>
<td>65.5, 10%</td>
<td>52.1, 8%</td>
</tr>
<tr>
<td>Procedure</td>
<td>753.4</td>
<td>150.7, 22%</td>
<td>124, 18%</td>
</tr>
<tr>
<td>PC Base</td>
<td>838.6</td>
<td>290.2, 42%</td>
<td>261.2, 38%</td>
</tr>
<tr>
<td>PC Full</td>
<td>978.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fpppp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>293.8</td>
<td>15.6, 5%</td>
<td>11.1, 4%</td>
</tr>
<tr>
<td>Procedure</td>
<td>309.4</td>
<td>14.6, 5%</td>
<td>11.9, 4%</td>
</tr>
<tr>
<td>PC Base</td>
<td>308.4</td>
<td>21, 7%</td>
<td>14.6, 5%</td>
</tr>
<tr>
<td>PC Full</td>
<td>314.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doduc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>58.0</td>
<td>51.7, 89%</td>
<td>48.1, 83%</td>
</tr>
<tr>
<td>Procedure</td>
<td>109.7</td>
<td>3.2, 6%</td>
<td>2.1, 4%</td>
</tr>
<tr>
<td>PC Base</td>
<td>61.2</td>
<td>9.3, 16%</td>
<td>7.1, 12%</td>
</tr>
<tr>
<td>PC Full</td>
<td>67.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1.** Observed vs. timed overhead for SPEC programs.
4.2. Predicted cost

To gauge the effectiveness of the predicted cost, 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 at the top of Fig. 3. The value for the PC Full case had an error of almost 40%. This is due the Performance Consultant inserting instrumentation into a single procedure that is called thousands of times a second. The middle section of the table shows the predicted cost for the Fpppp application. For all three cases, the estimated cost was within 6% of base time to run the application using Paradyn. The last part of the table shows the predicted cost for the Dodo application. The errors in the predicted cost ranged from 4.7 to 73.1% in this case. The largest error was for the case of instrumenting the CPU time for all procedures. This was the single largest error we saw for the predicted cost. We also compared the predicted cost data for the three PVM applications. The results are shown in Fig. 4. The difference between the actual predicted running time for all the three applications was within 6% for all three levels of instrumentation.

5. 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
<table>
<thead>
<tr>
<th>Application Version</th>
<th>$C_{\text{observed}}$</th>
<th>$C_{\text{pred}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Percent</td>
</tr>
<tr>
<td>Ear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>65.5</td>
<td>10%</td>
</tr>
<tr>
<td>PC Base</td>
<td>150.7</td>
<td>22%</td>
</tr>
<tr>
<td>PC Full</td>
<td>290.2</td>
<td>42%</td>
</tr>
<tr>
<td>Fpppp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>15.6</td>
<td>5%</td>
</tr>
<tr>
<td>PC Base</td>
<td>14.6</td>
<td>5%</td>
</tr>
<tr>
<td>PC Full</td>
<td>21</td>
<td>7%</td>
</tr>
<tr>
<td>Doduc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>51.7</td>
<td>89%</td>
</tr>
<tr>
<td>PC Base</td>
<td>3.2</td>
<td>6%</td>
</tr>
<tr>
<td>PC Full</td>
<td>9.3</td>
<td>16%</td>
</tr>
</tbody>
</table>

Fig. 3. Timed observed cost vs. predicted cost.

<table>
<thead>
<tr>
<th>Application Version</th>
<th>$C_{\text{observed}}$</th>
<th>$C_{\text{pred}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Percent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVM_BT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>113.3</td>
<td>13%</td>
</tr>
<tr>
<td>PC Base</td>
<td>21.9</td>
<td>2%</td>
</tr>
<tr>
<td>PC Full</td>
<td>33.0</td>
<td>4%</td>
</tr>
<tr>
<td>PVM_LU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>6.2</td>
<td>6%</td>
</tr>
<tr>
<td>PC Base</td>
<td>4.4</td>
<td>4%</td>
</tr>
<tr>
<td>PC Full</td>
<td>5.6</td>
<td>5%</td>
</tr>
<tr>
<td>PVM_SP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>8.6</td>
<td>2%</td>
</tr>
<tr>
<td>PC Base</td>
<td>27.2</td>
<td>6%</td>
</tr>
<tr>
<td>PC Full</td>
<td>35.0</td>
<td>7%</td>
</tr>
</tbody>
</table>

Fig. 4. Timed observed cost vs. predicted cost for PVM applications.

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-determined 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 the W3 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 satisfied without undue perturbation. In automated search mode, the interaction between the predicted cost and the search process is more complex. 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 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 a 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.

6. Evaluation of using cost to control searching

Currently, the principal use of our cost system is to control hypothesis evaluation in the Performance Consultant, therefore 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) the cost system controlling 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 for a fixed limit of three refinements to the current hypothesis, and the 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 to the thresholds.

The results for the three serial applications Doduc, Ear, and Fpppp are shown in Fig. 5. The search 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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Control method</th>
<th>Search time</th>
<th>C_measured</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td>Doduc</td>
<td>10%</td>
<td>48.2</td>
<td>3.1%</td>
</tr>
<tr>
<td></td>
<td>3 Hypotheses</td>
<td>77.1</td>
<td>3.1%</td>
</tr>
<tr>
<td></td>
<td>unlimited</td>
<td>258.7</td>
<td>6.2%</td>
</tr>
<tr>
<td>Fpppp</td>
<td>10%</td>
<td>52.0</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td>3 Hypotheses</td>
<td>73.2</td>
<td>0.5%</td>
</tr>
<tr>
<td></td>
<td>unlimited</td>
<td>227.2</td>
<td>1.9%</td>
</tr>
<tr>
<td>Ear</td>
<td>10%</td>
<td>37.9</td>
<td>2.7%</td>
</tr>
<tr>
<td></td>
<td>3 Hypotheses</td>
<td>130.8</td>
<td>5.1%</td>
</tr>
<tr>
<td></td>
<td>unlimited</td>
<td>226.5</td>
<td>15.4%</td>
</tr>
</tbody>
</table>

Fig. 5. Summary of fixed vs. cost-based hypothesis evaluation.
We verified that the cost control mechanism maintained instrumentation overhead within the target limits. We were interested not only in the total cost of instrumentation, but since we were changing the instrumentation during program execution, we also wanted to characterize how the instrumentation overhead changed with time. To measure this effect, we treated the observed cost as a Paradyn metric and periodically sampled its value. From these samples, we computed the average, maximum, and standard deviation of the observed cost. The last three columns of table in Fig. 5 summarize the cost of data collection for each application and approach. For all three applications, using cost to control hypothesis evaluation 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 three hypotheses 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 peak value of $C_{\text{obs,ideal}}$ ranged from 3.7 to over 40%. The wide range of cost for the unlimited case shows that trying to evaluate all hypotheses at once can have a wildly differing 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.

7. Related work

Perturbation compensation [6, 12] reconstructs 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 using the predicted cost, and quantify it using the observed cost.

Pablo [8] uses an 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 code 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.
Goldberg and Hennessy [3] used the difference between the measured and predicted time of a code region to quantify the effects of the memory hierarchy. Our approach differs in two ways from theirs. First, since we need to be able to characterize the impact of small, but (potentially) frequently accessed instrumentation code blocks, we use statistical sampling instead of timers. Second, our goal is to compute the impact of the instrumentation on the original code rather than the impact of the cache on a single basic block.

8. Conclusions

Our cost system controls the 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 are not discarded to reduce overhead, nor are they 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 W³ search model is a natural fit with the cost system. We have implemented the cost system in Paradyn and demonstrated that we could accurately track the cost of data collection in several applications.

References

