An Adaptive Cost System for Parallel Program Instrumentation *

Jeffrey K. Hollingsworth¹ and Barton P. Miller²

¹ University of Maryland, hollings@cs.umd.edu
² University of Wisconsin, bart@cs.wisc.edu

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 we regulate the amount of instrumentation to ensure that threshold is not exceeded. Our approach is unique in that the type of data gathered remains constant; instead we regulate when it is collected. This permits programmers to trade speed of isolation of a performance problem for less application perturbation. We implemented this cost system in the Parodyn Performance Tools and present case studies demonstrating the accuracy of the cost system.

1 Introduction

We present a new way to manage the perturbation caused by software 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.

The best way to handle instrumentation overhead is to avoid introducing it in the first place. In a previous paper [4], we described a new approach to performance monitoring called Dynamic Instrumentation. Dynamic Instrumentation delays instrumenting an application until it is in execution, permitting dynamic insertion and alteration of the instrumentation during program execution. Enabling instrumentation only when it is needed greatly reduces the amount of data collected, and thus the overhead due to the instrumentation system.

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 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

* 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 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 WLAD or the U.S. Government.
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. 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 permits our tools to more quickly isolate a performance problem (see Section 6).

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. To be useful, our observed cost system needs to be both cheap to compute and accurately reflect the true impact of data collection. If the observed cost exceeds predefined 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.

2 Dynamic Instrumentation and $W^3$ Search Model

Our recent work in performance monitoring tools has focused on two areas. First, how can we efficiently collect performance data for large, long running applications? Second, how can we help programmers to understand the source of their performance problems rather than providing them raw performance data.

Our approach, 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 can start collecting a metric for a particular combination of resources. To satisfy this request, instrumentation code is generated and inserted into the program. When performance data is no longer required, its instrumentation code is removed from the 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. Resources include processors, interconnection networks, processes, procedures, and synchronization objects. Metrics are time varying functions that characterize a program's performance; they can be computed for any subset of the resources in the system. For example, CPU time can be computed for a single procedure executing on one processor or for the entire application.

We have also been investigating how to help programmers interpret the collected performance data. The $W^3$ Search Model[6] is a structured methodology 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