To Appear in Proceedings of the
1994 Scalable High Performance Computing Conference, May 1994 (Knoxville, TN).

Dynamic Program Instrumentation for Scalable Performance Tools

Jeffrey K. Hollingsworth — hollings@cs.wisc.edu
Barton P. Miller — bart@cs.wisc.edu
Jon Cargille — jon@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

Abstract

In this paper, we present a new technique called dynamic instrumentation that provides efficient, scalable, yet detailed data collection for large-scale parallel applications. Our approach is unique because it defers inserting any instrumentation until the application is in execution. We can insert or change instrumentation at any time during execution by modifying the application's binary image. Only the instrumentation required for the currently selected analysis or visualization is inserted. As a result, our technique collects several orders of magnitude less data than traditional data collection approaches. We have implemented a prototype of our dynamic instrumentation on the CM-5, and present results for several real applications. In addition, we include recommendations to operating system designers, compiler writers, and computer architects about the features necessary to permit efficient monitoring of large-scale parallel systems.

1. Introduction

Efficient data collection is a critical problem for any system that monitors the performance of a parallel or distributed application. We have estimated that monitoring programs at a reasonable level of detail on current RISC processors can easily generate two megabytes per second per processor of performance data. For a massively parallel computer (say 1000 nodes), this amount of data is impractical to collect for all but the shortest programs. However, to understand the performance of parallel programs, it is necessary to collect data for full-sized data sets running on large numbers of processors. In this paper, we present a new approach to performance instrumentation that defers instrumenting the program until it is in execution, permitting dynamic insertion and alteration of the instrumentation during program execution.

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

A detailed instrumentation system needs to be able to collect data about each component of a parallel machine. To correct bottlenecks, programmers need to know as precisely as possible how the utilization of these components is hindering the performance of their program.

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

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

Scalable instrumentation requires that data collection remain detailed yet frugal as the number of resources in the system grows. This means that we should be able to handle increases in the application code size, length of program execution, data set size, and number of processors. Providing scalable instrumentation is important because the nature of bottlenecks in parallel programs change as each dimension is increased.

We tried to remove obstacles that could deter programmers from using our system. We feel that recompiling an application to use a performance tool is an unrea-

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

sonable burden. The only step required to use our tool is to run a standalone pre-processor that appends our instrumentation library to the application's binary. The instrumentation is enabled by modifying the application's binary image while it is executing. Our technique has no impact on the application until we insert instrumentation. Also, once we insert the instrumentation, there is no overhead to check if it is enabled. At any point during the application's execution, the programmer can invoke our tool and attach it to their already running program and start to investigate its performance.

While our dynamic approach makes it possible to collect a wide variety of performance data, it also requires that runtime decisions be made about what data to collect and when to collect it. Although it is possible for programmers to manually control data collection, this is difficult and tedious. We have built a system called the Performance Consultant[6] that liberates the programmer from needing to make these decisions.

The rest of this paper describes an instrumentation system we have built to meet goals of being detailed, frugal, and scalable. Section 2 describes the design of our system. Section 3 provides details about its implementation. Section 4 presents a series of performance benchmarks for our initial implementation of dynamic instrumentation on the CM-5. Sections 5 and 6 describe related work and conclusions about our approach respectively.

2. Design

Our instrumentation system is designed as an interface to be used by higher level analysis and visualization tools. We present two abstractions to these tools: *resources* and *metrics*. Resources are the objects about which we gather performance information. Metrics are quantitative measures of performance.

2.1. Resources and Metrics

The first abstraction provided by our dynamic instrumentation is resources. Resources are separated into several different hierarchies, each representing a class of objects in a parallel application. For example, there is a resource hierarchy for CPUs, containing each processor. Figure 1 shows three sample resource hierarchies. The left-most hierarchy is for code resources. The root of the hierarchy is *Code Objects*. The next level contains the modules (files) that constitute the application under study. Below each module are the procedures defined in that module.

Individual resource instances are created both when an application starts execution, and during its execution. The static components are at the top of each hierarchy, and the more dynamic nodes are at the lower levels in each hierarchy. The lowest levels in each hierarchy, representing specific resource instances, are added during the application's execution when the resource is first used. For example, the procedure resources are created when the program to be monitored is specified. However, information about data files used is discovered dynamically when the files are opened.

The second abstraction is metrics. Metrics are time varying functions that characterize some aspect of a parallel program's performance; examples include CPU utilization, counts of floating point operations, and memory usage. They are defined by higher level performance tools that use our instrumentation system (e.g., the Performance Consultant[6]). However, to assist tool builders, we provide a standard set of metrics as part of our system. 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.

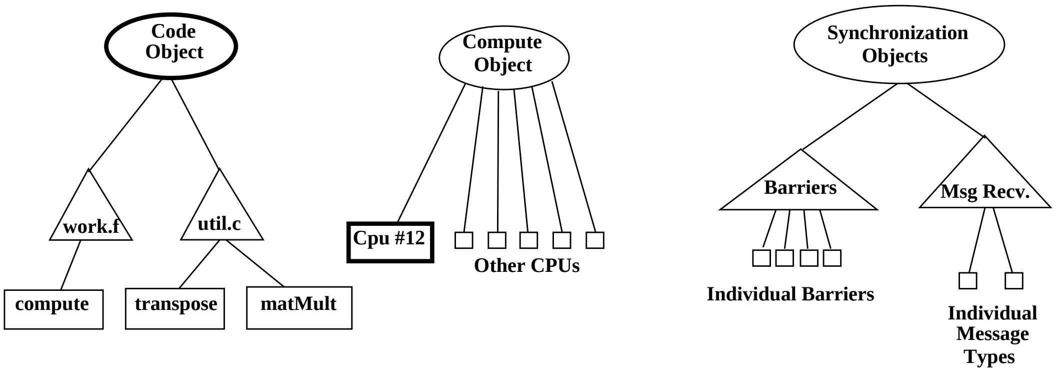

Figure 1. Resources showing three class hierarchies.
Each resource instance is associated with a single resource hierarchy. The shape of the resource indicates when it was defined. Oval objects are statically defined independent of the application. Triangles are static based on the application, and rectangles are dynamically (runtime) identified.

2.2. Data Collection Model

To provide the resource and metric abstractions, we developed a new dynamic data collection model that combines the advantages of tracing and sampling. Our approach is to dynamically modify the program to record precise information about relevant state transitions in counter and timer data structures. These structures are then periodically sampled to report performance information to the higher layers of our system. We control the volume of data collected in two ways: first by collecting only the information needed at a give moment, and second by controlling the sampling rate.

Our hybrid of dynamic tracing and sampling can provide more accurate information than can be provided by pure sampling. For example, if you periodically sample the program counter to compute the amount of time spent in a procedure, the accuracy of the result will depend on the sampling rate. Instead, we insert code to start and stop timers at the procedure entry and exit to accurately record time spent in the procedure. In this case, the sampling rate affects only how frequently we update our knowledge of the time, not the accuracy of the time.

We periodically sample our performance metrics for two reasons. First, we want to isolate performance bottlenecks to specific phases (time intervals) during a program's execution. Second, our higher level analysis system uses partial data to decide what additional data should be collected to further isolate a performance bottleneck. The dynamic instrumentation actually collects and calculates the performance data; sampling is used only to periodically report the collected data. Varying the sampling rate affects only our rate of decision making and granularity of phase boundaries; it does not affect the accuracy of the underlying performance data.

Collected data is stored in a data structure called a time histogram[5]. A time histogram is a fixed-size array whose elements store values of a performance metric for successive time intervals. Two parameters determine the granularity of the data stored in time histograms: initial bucket width (timer interval) and number of buckets. Both parameters are supplied by higher level consumers of our performance data. However, if the program runs longer than the initial bucket width times the number of buckets, we run out of buckets to store new data. In this case, we simply double the bucket width and re-bucket the previous values. In addition, we change the sampling rate to the new bucket width. This process repeats each time we fill all the buckets.

2.3. Points, Primitives, and Predicates

Recording performance information about the application program is accomplished by *points*, *primitives* and *predicates*. *Points* are well-defined locations in the application's code where instrumentation can be inserted†. *Primitives* are simple operations that change the value of a counter or a timer. *Predicates* are boolean expressions that can be associated with primitives that determine if the associated primitive gets executed. By inserting predicates and primitives at the correct points in a program, a wide variety of metrics can be computed.

`if (expr) actn`

If `expr` is non-zero then execute `actn`.

`actn`

An `if` statement or a call to a primitive.

`operand`

A counter, constant, or parameter.

`expr`

`operand operator operand` *or*
`operand`

`operator`

`+ - / * < > <= >= == != and or.`

Figure 2. Description of the Predicate Language.

Our system consists of six primitives: set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. Predicates are simple conditional statements that consist of an expression and an action. If the value of the expression is non-zero, the associated action is taken. Expressions can contain numeric and relational operators. The operands of predicate expressions can be either counters or constants. If the point where the predicate is inserted is a procedure call or entry, the parameters to the procedure can be used as operands in expressions. If the predicate is inserted at a procedure exit, the procedure's return value can be used. Actions are either other predicates or calls to one of the six primitives. Figure 2 describes the predicate facility.

Examples of how primitives and predicates can be combined to create metrics is shown in Figure 3. This example shows two different metrics. The first example computes the number of times procedure `foo` is called. A single primitive to increment a counter by one has been inserted at the entry point to the procedure `foo`. The second example shows a metric to compute the number of bytes transferred via a message passing procedure. The second parameter to the add counter primitive is a multiplication expression that uses the third and fourth arguments to the message passing function to compute the number of bytes transferred.

† Currently the available points are the procedure entry, exit and individual call statements. In future versions of our instrumentation, points will be extended to include basic blocks and individual statements.

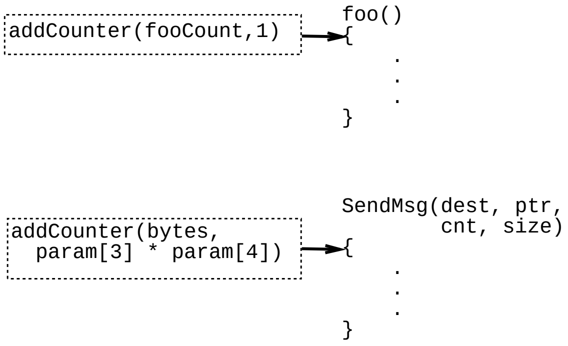

Figure 3. Example Showing Two Different Metrics.

Figure 4 shows a slightly more complex example of dynamic instrumentation. Four instrumentation points are used to compute the waiting time due to message passing constrained to a single procedure. The top two primitives maintain a counter `fooFlg`, which is non-zero whenever the procedure `foo` is active. The lower two points are calls to primitives to start and stop the timer `msgTme`. However, the timer operations only occur when the counter `fooFlg` is non-zero.

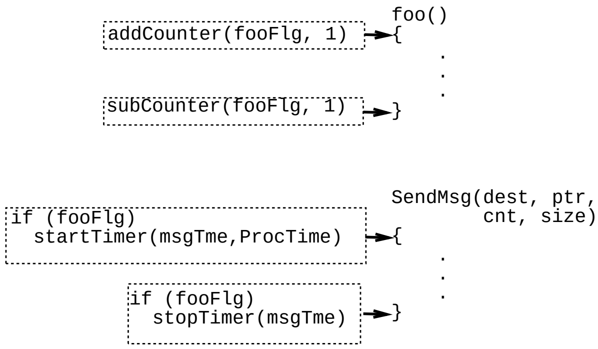

Figure 4. Sample Constrained Metric.

3. Implementation

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

3.1. Metric Manager

The Metric Manager's role is two-fold; it translates requests for metrics into primitives and predicates and informs higher level consumers about new resources. It also provides a way to monitor and control the amount of overhead that the instrumentation will inflict on the application's execution. The Metric Manager is designed to be application and machine independent.

The translation from metrics to primitives is described by *metric definitions*. These definitions can be provided by higher level performance tools, although we provide a substantial library of them. A metric definition can be viewed as a template that describes how to compute a metric for different resource combinations. It consists of a series of code fragments that create primitives and predicates to compute the desired metric.

Our strategy of enabling instrumentation only when it is needed greatly reduces the amount of data collected, and therefore the potential perturbation caused by our instrumentation system. However, instrumentation requests still have an impact on the program's performance. To help quantify this effect, the Metric Manager includes a *predicted cost model*. The predicted cost model provides information about the expected perturbation of the application for each instrumentation request. The Metric Manager uses the metric's definition to compute what instrumentation primitives and predicates need to be inserted and where. For each instrumentation point, the overhead of its predicates and primitives is multiplied by the point's expected execution frequency to compute the predicted perturbation. The overhead information for each predicate and primitive is measured once for each hardware platform and stored in a system configuration file. The execution frequency of points comes from a static model of procedure call frequency. While the cost model does not perfectly predict the impact of instrumentation on the application, data collected during execution can be used to dynamically tune the estimates during the course of program execution[6].

3.2. Instrumentation Manager

The Instrumentation Manager performs two functions: it identifies the potential instrumentation points, and handles requests for primitives and predicates and inserts them into the application program. The requests are translated into small code fragments, called *trampolines*, and inserted into the program. We define two types of trampolines: base trampolines and mini-trampolines (see Figure 5). There is one base trampoline per point with active instrumentation. Base trampolines have slots for calling mini-trampolines both before and after the relocated instruction. One slot before and one slot after the instruction are used to call global primitives (ones that are inserted into all processes in an application). The other slots are used to call local primitives (ones that are specific only to this process). This structure makes it easier to use hardware broadcast facilities to install global requests.

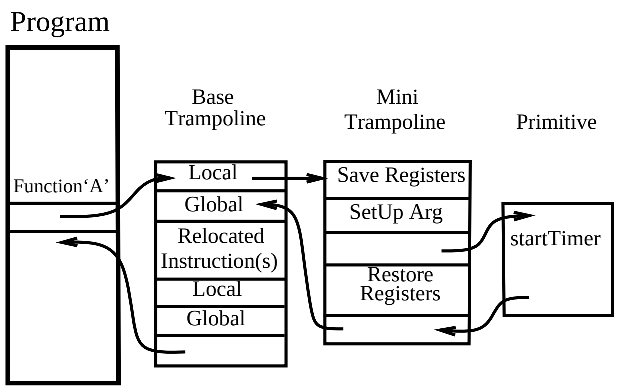

Figure 5. Inserting Instrumentation into a Program.

Identifying the points where instrumentation can be inserted is accomplished by analyzing the instructions in the application. This is one area where compiler writers could help make our task easier. It would be helpful if additional symbol table information were available that indicated exactly what is code and what is data in a program. Many compilers place read-only data into a program's text (code) segment. This creates a problems for post-linker tools (correctness debuggers and performance tools). Ball[1] has also noted this problem.

Mini-trampolines contain predicate and primitive specific code and there is one for each primitive at each point. A sample instrumentation point with trampolines installed appears in Figure 5.

Creating a mini-trampoline requires generating appropriate machine instructions for the primitives and predicates requested by the Metric Manager. The predicate language is simple, requiring only a handful of instruction types. The instructions are assembled by the Instrumentation Manager, and then transferred to the application process using a variation of the UNIX ptrace interface (described in the next section). In addition to code requested by the Metric Manager, code must also be generated to save and restore any registers that the generated code (or called primitives) might overwrite. When we extend the available points to include basic blocks and individual statements, additional processor state (such as condition codes) must also be saved and restored. Figure 6 shows two sample mini-trampolines. The first example (a) shows a simple counter increment that has been generated as inline code. This is the code that would be generated for the first primitive in Figure 4. The second example (b) shows the instrumentation generated for the predicate and primitive at the entry point to the `SendMsg` procedure in Figure 4. The `if` statement has been translated into a branch based on the value of the `fooFlg` counter. If the counter is non-zero, the `stopTimer` predicate is called from the mini-trampoline.

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

(a)

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

(b)

Figure 6. Two Sample Mini-Trampolines.

To implement our dynamic instrumentation, we needed several efficient operating system services. These include a data transport mechanism for moving performance data off of the parallel machine with little perturbation of the program under study, and a scalable version of Unix ptrace. Unfortunately, the operating system of our target machine included neither of these features, and we were forced to construct them.

The data transport is responsible for moving performance data off the parallel machine to some other location where it can be analyzed. However, if performance data moves through the parallel machine during program execution, it can compete for resources with the program under study, and thus perturb it. We take advantage of the CM-5 synchronous gang scheduling to avoid this problem. We developed something similar to a signal handler, that we term a "timeslice handler". This handler is a piece of user-level code that gets called on each node at the end of each timeslice, after the normal program context has been saved by the operating system.

On top of this timeslice handler mechanism, we built the data transport. Each node buffers performance data in memory while the application program runs, and then transfers the data at the end of each timeslice. Since the program has already been stopped by the operating system and its execution context has been saved, the program execution state is not affected and the timeslice handlers do not contend with the program for machine resources.

This technique will cause some dilation of wall-clock execution time, but the data transport itself has virtually no perturbing effect on the program's execution†. On machines without a synchronous scheduling model, such as the Intel Paragon, we use this model of data tran-

† The one exception is when the data buffer fills up on the nodes; then it is necessary to either force the timeslice to end early, or wait for it to end. Either option has a perturbing effect.

sport but it is more difficult to factor out the cost.

The Instrumentation Manager may make frequent changes to the instrumentation during the application's execution. To be scalable, the cost of instrumentation operations must not grow with the number of nodes in the parallel computer. When we insert instrumentation into all instances of a procedure on all nodes, the cost should be the same as inserting instrumentation into a single node. To achieve this goal, we have developed a parallel (broadcast based) version of the ptrace function on the CM-5. Ptrace allows the application address space to be read and written. We use our broadcast ptrace to insert global instrumentation into all nodes in unit time.

On machines that do not provide hardware broadcast, such at the IBM SP/1, we can construct a software message spanning tree. The spanning tree technique achieves logarithmic time, instead of unit time cost. We are currently experimenting with these software mechanisms. Most new machines come with some form of broadcast facility, including the Intel Paragon (though it is currently not accessible to application software), Cray T3D, and Meiko CS-2. We believe that unit time operations such as these are crucial to dealing with the issue of scale in tools for large parallel machines.

4. Performance of Implementation

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

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

4.1. Micro Benchmarks

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

The first aspect of the performance of our instrumentation system we studied was the overhead of the trampolines that get inserted any time a primitive is called. Figure 7 shows the cost involved in trampolines. The time required for a base trampoline is 4 clock cycles‡. This time provides four no-op slots to call the mini-trampolines. The second row shows the time required for the smallest mini-trampoline. Two clock cycles are required for the SPARC save and restore register instructions (for case when no register window overflow occurs). Another source of delay in the trampolines (8 clock cycles long) is for the branches into and out of the trampolines. The final component of the trampoline overhead is the time required to setup the parameters to call the predicate. This time ranges from 1 to 5 clock cycles depending on the type of the parameter.

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

Figure 7. Time to execute trampolines.

The other intrinsic cost of our instrumentation is the time to execute our primitives. Figure 8 shows each of our six primitives and their execution time in clock cycles. In addition, the time to execute a simple predicate that checks if a counter is positive is included. The measured times ranged from 7 clock cycles for the predicate to 71 clock cycles to stop a process timer. These results were collected by invoking each primitive 1,000 times in a tight loop, and recording the elapsed time. For each call, the counter or timer passed to the primitive was the same, so the effects of the machine's memory hierarchy has been factored out of these results. To get a feel for the relative cost of these numbers, we compared the cost of our primitives to simple procedure calls on the machine. A one-argument procedure call that returned its argument took 15 clock cycles.

To look for bottlenecks in our timers, we isolated the time required to execute each part of each timer primitive. Figure 9 shows each of the four timer primitives and 5 different operations used to implement the timer.

‡ It's a 33Mhz clock, so one cycle is about 30 nano-seconds.

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

Figure 8. Cost of Primitive Operations.

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

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

Figure 9. Primitive times by component.

Our breakdown of the cost of timer operations shows that the largest time component in any primitive (and the majority of time in two cases) is spent reading the clock. This time could be substantially reduced if processors provided a clock that was readable by user level code at register access speed. Few RISC processors, with the exception of the DEC Alpha[12], provide such a clock. We feel this type of clock is a critical architectural feature to permit building efficient performance tools.

The numbers presented above reflect the time to execute our primitives after we made a slight modification to the CM-5 operating system. Originally, the operating system required a system call to read the clock, and the overhead of this system call dominated the cost of our timer primitives. Figure 10 shows the time to execute our four timer primitives both with and without the mapped clock. Using this system call slowed down our timer primitives by a factor ranging from 2 to 9. We overcame this bottleneck by adding a new system call to the operating system that maps the kernel's clock data structure into user address space so that we could read the clock at memory speed. Another option to implement our timers would be to use the timer library routines supplied by Thinking Machines. The results of using their timer calls also appears in Figure 10. Our timers are a factor of 6 to 12 times faster than the vendor-supplied ones. However, the Thinking Machine timers provide both a process and a wall timer in one. While this may be useful for some applications, for dynamic instrumentation we only need one timer type at a time. We feel it is vitally important when building performance tools that accessing clocks be as fast as possible. Requiring a kernel call to read a clock is unacceptably slow.

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

Figure 10. Timer overheads on a TMC CM-5.

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

We also measured the performance of several components of our scalable ptrace implementation to determine the performance of each piece. Figure 11 outlines the structure of the communication paths in our implementation. First, ptrace requests are sent from the Instrumentation Controller over a pipe to the Timeslice Handler running on the front end processor. These requests are then broadcast over the CM-5 network to the nodes where they are processed. Ptrace requests are aggregated into batches by the Instrumentation Controller, and sent as a group to amortize the per-operation cost of the pipe.

For each of the two communication paths shown in Figure 11 we divided the overhead into three component costs. First, there is a fixed startup latency associated with sending any request. This round-trip-time is represented in the first row of Figure 12. Second, there is a cost associated with interpreting and executing each request; this is shown in the second row. Last, there is a cost for each byte transferred, shown in the third row.

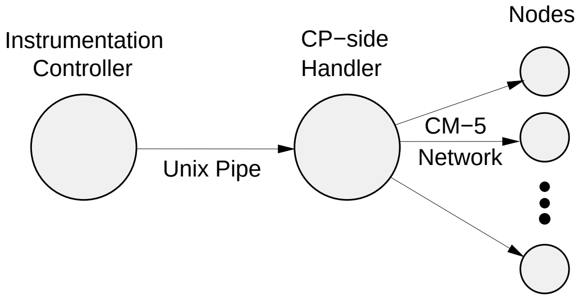

Figure 11. Communication Paths for Scalable Ptrace.

Cost Component	Time (in micro-seconds) Controller to Handler	Handler to Nodes
Startup Cost	3027.0	35.9
Per-operation Cost	8.3	14.9
Per-byte Cost	0.3	1.8

Figure 12. Component Costs of Broadcast Ptrace.

The results show several bottlenecks in our current ptrace implementation. There is a high cost, 3 milliseconds, associated with each batch transferred due to UNIX context-switching. The impact of this bottleneck can be mitigated by using large batch sizes to amortize the startup cost across many operations. Although the CM-5 network provides better bandwidth than a Unix pipe, the per-byte cost in the handler to node case is higher than for the pipe. This is because each byte sent to a node is received into a temporary buffer, and then copied into the desired memory location. This extra copying reduces the effective bandwidth of the broadcast network.

We also measured the performance of our data transport layer, which moves performance data from the nodes to the front end processor. The data path for the transport is the reverse of that shown in Figure 11. Our data transport layer, implemented using timeslice handlers, achieves throughput of approximately 2 MB/sec from the nodes to the front end. The throughput is bounded by hardware architecture; although the network is capable of greater bandwidth (40 MB/s theoretical physical limit), the front-end processor can only receive from the network when the appropriate process is scheduled by the SunOS kernel. Although 2MB/sec can arrive on the front-end, we are limited to about 1.5 MB/sec because we pass data through a Unix pipe.

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

4.2. Macro Benchmarks

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

For the dynamic tests, we ran two parallel applications on the CM-5. The first application does a domain-decomposition method for optimizing large-scale linear models. The second program is a database simulator that implements a parallel Grace Hash join algorithm in a simulated shared-nothing environment[11]. Both programs are written in C. We compared the execution time of an un-instrumented application to the execution time of an instrumented application. We measured the overall cost of our instrumentation and the overhead of various components of the instrumentation. For this study, we used the Performance Consultant[6], a system that takes full advantage of dynamic instrumentation, to drive our instrumentation.

The results for these tests are shown in Figure 13. The second column, Original Execution Time, shows the time to run the application without any instrumentation. One program ran for about an hour, and the other one for a couple of minutes. The third column, Execution Time w/Instrumentation, shows the total execution time with dynamic instrumentation. To understand where this time is spent, we have divided the overhead for instrumentation into three categories: Null Handler time, Actual

Handler time, and Direct Perturbation. The first two types of overhead represent the time spent in the end-of-timeslice handlers. Null Handler time is the cost of running the end-of-timeslice code without any instrumentation enabled. The Actual Handler time is the time to process ptrace requests and collect data from the nodes. The differences between the two applications for Actual Handler time is due to the larger volume of performance data transferred for the database application. Since the CM-5 uses synchronous gang scheduling, overhead time in the handlers does not directly perturb the performance of the program. Instead, it dilates execution similar to the way that other time-shared jobs impact an application's execution. The third type of overhead represents the direct perturbation of the application due to the instrumentation inserted. It is a modest amount (6 & 8%) for these two applications, since the Performance Consultant requests only the instrumentation it requires.

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

The results of running dynamic instrumentation, prof, and gprof on the two sequential applications is summarized in Figure 14. The first application is MultiComm which is a math programming application that solves a multi-commodity network flow problem with mutual capacity constraints. It is written in a mixture of C and Fortran. For this application, dynamic instrumentation has only 4% overhead, but prof and gprof have overheads of 40% and 79% respectively. This is because prof and gprof instrument the internals of the C library, and this application makes heavy use of integer divide which is a library function on most SPARC machines.

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

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

Application	Original Execution Time	Execution Time w/Instrumentation	Components of Overhead		
			Null Handlers	Actual Handlers	Direct Perturbation
Decomp	59:22	1:05:29 (10%)	01:26 (2%)	00:06 (< 1%)	04:34 (8%)
Hash-join	01:22	0:02:09 (57%)	00:10 (12%)	00:32 (39%)	00:05 (6%)

Figure 13. Cost of ptrace operations.
This table shows the overhead of using dynamic instrumentation with the Performance Consultant for two applications running on the CM-5. All times in minutes:seconds.

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

Figure 14. Overhead of different Sequential CPU Profilers.

5. Related Work

Several systems have been built that defer instrumentation until after compilation. Both QPT[1] and Mtool[4] use binary re-writing to insert instrumentation into an object file after it has been compiled and assembled. These systems require data collection decisions to be made prior to program execution. One system that defers instrumentation until the program has started to execute is the TAM facility[10] provided by Intel for the Paragon. TAM uses a static set of performance instrumentation profiles (i.e. prof style sampling, or full event tracing) to insert instrumentation into a program after it has been loaded into memory but prior to execution. Their method of inserting instrumentation is similar to ours, but their data collection is not as dynamic.

The other major area of related work are techniques for modifying a program once it has started execution. A number of correctness debuggers have been built that modify an executing program for assertion checking and conditional breakpoints. Jeff Brown[2] developed a debugger for Cray Computers that provided this feature. Kessler at Xerox Parc[7] built a system that used dynamic modification of a program to insert breakpoints. Work has also been done by Wahbe, et. all on fast data breakpoints[13]. They employ sophisticated program analysis techniques to minimize the overhead of instrumentation for data breakpoints. Massalin and Pu[8] have a novel application of code patchup in their Synthesis operating system. They modify a program to automatically schedule another thread when it is about to block.

6. Conclusion

We have described the design and implementation of a new data collection strategy that meets ours goals of being detailed, frugal, and scalable. Our current implementation runs on a Thinking Machine CM-5 using explicit message passing programs written in C, C++, or Fortran. The cost of the primitive operations is only a few instruction times, and overall application perturbation was under 10% for several real applications tested. Our system also provides a simple interface, metrics and resources, that makes it easy to use our instrumentation system with higher level performance tools.

In addition, we have provided several specific recommendations to compiler writers, operating systems designers, and machine architects about the features necessary to monitor large parallel systems. Compilers need to provide additional symbol table information for post-linker tools to be able to correctly relate machine activity back to the source program. Operating systems need to provide scalable versions of services like ptrace, and make clock access as cheap or cheaper than memory accesses. Architects should provide high resolution clocks that are easy to access by user-level programs.

This work is part of an ongoing project, called Paradyn, to design and build a complete performance measurement environment for large-scale parallel machines. In the future, we plan to conduct a detailed case study of both the Performance Consultant[6], dynamic instrumentation, and their interactions using several real applications. We are also in the process of studying and refining our predicated cost model. Work is underway to provide dynamic instrumentation on the Intel Paragon and clusters of workstations (using PVM[3]).

7. Acknowledgements

We thank Adam Greenberg of TMC for helping to benchmark the CMOS timers, and Babak Falsafi for providing the memory mapped timers for the CM-5. We also thank the authors of the applications used in our study: Mark Hill and Madhusudhan Talluri (Tycho), Spyros Kontogiorgis (Decomp), Tia Newhall (Hash-join), and Gary Schultz (MultiComm).

References

1. T. Ball and J. R. Larus, Optimally Profiling and Tracing Programs, in *19th ACM Symposium on Principles of Programming Languages*, (Albuquerque, NM, January 19-22, 1992), 1992, 59-70.
2. J. S. Brown, *The Application of Code Instrumenation Technology in the Los Alamos Debugger*, Los Alamos National Laboratory, October 1992.
3. J. Dongarra, G. A. Geist, R. Manchek and V. S. Sunderam, "Integrated PVM framework supports heterogeneous network computing.", *Computers in Physics 7*, 2 (March-April 1993), pp. 166-74.
4. A. J. Goldberg and J. L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL", *Supercomputing'91*, Albuquerque, NM, Nov. 1991, pp. 481-490.
5. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool", *1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, April 1991, pp. 189-200.
6. J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", *7th ACM International Conf. on Supercomputing*, Tokyo, July 1993, pp. 185-194.
7. P. B. Kessler, "Fast Breakpoints: Design and Implementation", *ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, White Plains, NY, June 20-22, 1990, pp. 78-84.
8. H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel", *ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Dec 3-6 1989, pp. 191-201.
9. J. K. Osterhout, *USENIX Winter Conf.*, Jan. 1990, pp. 133-146.
10. B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards and W. Smith, "The Paragon Performance Monitoring Environment", *Supercomputing'93*, Portland, OR, Nov 15-19, 1993, pp. 850-859.
11. D. A. Schneider and D. J. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, Tech. Report 836, Dept. of Comp. Sci., University of Wisconsin, April 1989.
12. R. L. Sites, ed., *Alpha Architecture Reference Manual*, Digital Press, 1992.
13. R. Wahbe, L. Lucco and S. L. Graham, "Practical data breakpoints: design and implementation", *ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM, June 23-25, 1993, pp. 1-12.