Optimization of Instrumentation in Parallel Performance Evaluation Tools

Sameer Shende, Allen D. Malony, and Alan Morris

Performance Research Laboratory,
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA,
{sameer,malony,amorris}@cs.uoregon.edu

Abstract. Tools to observe the performance of parallel programs typically employ profiling and tracing as the two main forms of event-based measurement models. In both of these approaches, the volume of performance data generated and the corresponding perturbation encountered in the program depend upon the amount of instrumentation in the program. To produce accurate performance data, tools need to control the granularity of instrumentation. In this paper, we describe developments in the TAU performance system aimed at controlling the amount of instrumentation in performance experiments. A range of options are provided to optimize instrumentation based on the structure of the program, event generation rates, and historical performance data gathered from prior executions.
Keywords: Performance measurement and analysis, parallel computing, profiling, tracing, instrumentation optimization.

1 Introduction

The advent of large scale parallel supercomputers is challenging the ability of tools to observe application performance. As the complexity and size of these parallel systems continue to evolve, so must techniques for evaluating the performance of parallel programs. Profiling and tracing are two commonly used techniques for evaluating application performance. Tools based on profiling maintain summary statistics of performance metrics, such as inclusive and exclusive time or hardware performance monitor counts [1], for routines on each thread of execution. Tracing tools generate time-stamped events with performance data in a trace file. Any empirical measurement approach will introduce overheads in the program execution, the amount and type depending on the measurement method and the number of times it is invoked. However, the need for performance data must be balanced with the cost of obtaining the data and its accuracy. Too much data runs the risk of measurement intrusion and perturbation, yet too little data makes performance evaluation difficult.

Performance evaluation tools either employ sampling of program state based on periodic interrupts or direct instrumentation of measurement code. Sampling generally introduces a fixed overhead based on the inter-interrupt sampling interval. Thus, sampling is often considered to be generate less perturbation on performance. Unfortunately, sampling suffers from lack of event specificity and an inability to observe inter-event actions. For these reasons, sampling is less viable approach for robust parallel performance analysis. Here, we consider direct measurement-based techniques where instrumentation hooks are inserted in the code at locations of relevant events. During execution, events occur as program actions and the measurement code is activated to inspect performance behavior. Because event generation does not occur as the result of an interrupt, the number of events generated is instead tied to how often the program code executes. In some cases, this could be significant. Furthermore, the time between event measurements is not fixed, which can affect measurement accuracy. Whereas direct instrumentation and measurement can produce robust performance data, care must be taken to maintain overhead and accuracy. We use the term "instrumentation optimization" to describe this objective.

In this paper, we discuss our work in optimizing program instrumentation in the TAU performance system [2]. Section §2 gives further background and motivation for the problem. Sections §3 and Sections §4 describe how we can limit the instrumentation based on selective instrumentation and runtime measurement control. Section §5 discusses our future plans.

2 Motivation

Given a performance evaluation problem, certain performance data must be observed to address it. How should the program be instrumented and measurements made to capture the data? If the measurements cost nothing, the degree of instrumentation is of no consequence. However, measurements introduce overhead in execution time and overhead results in intrusion on the execution behavior. Intrusion can cause the performance of an application to change (i.e., to be *perturbed*). Performance perturbation directly affects the accuracy of the measurements. Accuracy is also affected by the resolution of the performance data source (e.g., real time clock) relative to the granularity of the metric being measured (e.g., execution time of a routine). Thus, it is not enough to know what performance data needs to be observed. One must understand the cost (overhead, intrusion, perturbation, accuracy) of obtaining the data.

Optimization of instrumentation is basically a trade-off of performance data detail and accuracy. If a measurement is being requested at a granularity too fine for the measurement system, the performance data will be faulty. Clearly, instrumentation should be configured to prevent the measurement from being made at all. At the other end of the spectrum, there can be situations where more data is gathered than necessary for a certain level of accuracy. Here, instrumentation optimization would be used to limit unnecessary overhead.

Most users are not so sophisticated in their performance measurement practices. Therefore, performance tools must provide mechanisms that enable users to understand instrumentation effects and accuracy trade-offs, and adjust their performance experiments accordingly. Balancing the volume of performance data produced and the accuracy of performance measurements is key to optimizing the instrumentation. Techniques for improving performance observability fall into three broad categories:

– *Instrumentation* – Techniques that *reduce* the number of instrumentation points inserted in the program.
– *Measurement* – Techniques that *limit* and *control* the amount of information emitted by the tool at the instrumentation points, and
– *Analysis* – Techniques that *scale* the number of processors involved in processing the performance data, and techniques that reduce and reclassify the performance information.

In this paper, we will limit our discussion to instrumentation and measurement based approaches.

3 Instrumentation

There are two fundamental aspects to optimizing instrumentation: deciding which events to instrument and deciding what performance data to measure. Event selection can occur prior to execution or at runtime. However, what is possible in practice depends entirely on the instrumentation tools available. Different types of events also determines the complexity of the instrumentation problem. The following discusses the possible instrumentation approaches and optimization issues that arise.

3.1 Event Types

Because direct instrumentation inserts measurement instructions in the program code, events are most often defined with respect to program flow of control. *Standard events* include the begin and end of routines and basic blocks. Other events may be defined at arbitrary code locations by the user. Events can also be defined with respect to program state. These events are still instrumented for with code insertion, but are 'enabled' depending on the value of state variables or parameters. We also distinguish between paired *entry/exit* events and events that are *atomic*.

The different types of events represent, in a sense, the range of possible event instrumentation scenarios. This could range from having all routines in a program code instrumented to having only a few specific routines instrumented, such as in a library. Clearly, the more events instrumented for, the more measurements will be active. Also, it is important to distinguish events that occur on individual threads (processes) of execution. Hence, the more threads executing, the more concurrent events are possible.

3.2 Instrumentation Mechanisms

Coinciding with the types of events to instrument are the mechanisms for instrumentation. There are five common approaches:

– *Compiler* – Instrumentation occurs as the program is being compiled. The choice of events is determined by the compiler, but options may be provided. Instrumentation for *gprof*-style profiling is generally done.
– *Library* – A library has been pre-instrumented and this instrumentation can be invoked by a program just by relinking. The MPI library is a special case of this since it also provides a "profiling interface" (PMPI) for tool developers to build their own instrumentation.
– *Source (automatic)* – This is an instrumentation approach based on source rewriting. The Opari [3] and TAU instrumentor [2] tools work in this manner.
– *Source (manual)* – A manual instrumentation API often accompanies measurement tools to allow users to create their own events anywhere in the program. For non-trivial applications, the effort to do so becomes quite cumbersome.
– *Binary* – Instead of working at the source level, some tools can instrument binary code, a form of binary rewriting. For the most part, the events are the same except lower-level code features may be targeted. Binary rewriting is hard and ISA specific.
– Dynamic – Some tools work at runtime to instrument executable code. For instance, DyninstAPI [5] can dynamically instrument running parallel executions based on code trampolining techniques.

The choice of which instrumentation mechanism to use is based on different factors. One factor is the accessibility (visibility) of events of interest. Another is the flexibility of event creation and instrumentation. These factors affect whether the mechanism can meet the event requirements. However, mechanisms are sometimes chosen based on their perceived overheads. For instance, source instrumentation has been criticized for its effects on code optimization, while binary and dynamic instrumentation purport to work with optimized code. On the other hand, source-instrumented measurement code also undergoes optimization, and can generate more efficient code than dynamically-instrumented measurements inserted under pessimistic assumptions of register usage and other factors.

3.3 Selective Instrumentation

Given the above discussion, instrumentation optimization with respect to events reduces essentially to a question of selective instrumentation. Put another way, we want to conduct performance experiments that capture performance data only for those events of interest, and nothing more. If a mechanism does not allow instrumentation of some event of interest, it is less useful than one that does. When there are many events that can be instrumented, a means to select events will allow only those events desired to be instrumented. Each of the mechanisms above can support event selection, but not all tools based on these mechanisms support it.

In the TAU project, a variety of instrumentation techniques are used: source pre-processing with PDT [4], MPI library interposition, binary re-writing and dynamic instrumentation with DyninstAPI [5], and manual. For each of these mechanisms, TAU allows an event selection file to be provided to control what events are to be 'included' in and 'excluded' from instrumentation. The specification format also allows for instrumentation to be enabled and disabled for entire source files.

Unfortunately, it is common for naive performance tool users to ignore such support and ask for all events to be instrumented. There are two downsides of this. First, it is probably true that not all of the events are really needed. Second, some events may be generated that are very small, resulting in poor measurement accuracy, or high-frequency, causing excessive buildup of overhead. Of course, a user could use TAU's selective instrumentation to disable such events, but they might not be aware of them.

TAU's selective instrumentation file also allows rules for instrumentation control to be specified. TAU provides a tool, *tau_reduce*, to analyze the profiles and apply the instrumentation rules. Effectively, the output is a list of routines that should be excluded from instrumentation. Naive instrumentation of parallel programs can easily include lightweight routines that perturb the application significantly when measured. What rules should the user then write?

If the user does not specify the rules for removing instrumentation using *tau_reduce*, TAU applies a default set (e.g., the number of calls must exceed one million and the inclusive time per call for a given routine must be less than 10 microseconds to exclude the routine). The program is then re-instrumented using the *exclude list* emitted by *tau_reduce*. To ensure that other routines that were above the threshold for exclusion before do not qualify for exclusion after re-instrumentation (due to removal of instrumentation in child routines), the user may re-generate the exclude list by re-running the program against the same set of rules. When any two instrumented executions generate no new exclusions, we say that the instrumentation *fixed-point* is reached for a given set of execution parameters (processor size, input, and so on) and instrumentation rules. The instrumentation is sufficiently coarse-grained to produce accurate measurements.

The selective, rule-based instrumentation approach implemented in TAU is a powerful methodology for performance experimentation. Users can create multiple event selection files and apply them depending on their experimentation purposes. However, there is still an issue of optimization with respect to the amount of performance data generated. This is discussed in the next section.

4 Measurement

Event instrumentation coupled with measurement code produces a "ready-to-run" performance experiment. Profile and trace measurements are the standard types used to generate performance data. Issues of instrumentation optimization regarding choice of measurement trade off detail for overhead. That is one part of the story. The other part has to do with the number of events generated versus overhead and measurement accuracy.

4.1 Measurement Choice

The overhead to generate the performance data during profiling and tracing is roughly comparable. However, because tracing produces more data, it runs the risk of additional overhead resulting from trace buffer management. Extremely high volume trace data can be produced. When this is unacceptable, one alternative is to switch the measurement method to profiling. The general point here is that the choice of measurement method is an effective means to control overhead effects, but with ramifications on the type of data acquired.

Once events have been specified for an experiment, TAU allows users to chose between profiling and tracing at runtime. The details of data produced for each event are decided both at link time and through environment settings.

4.2 Runtime Event Control

However, let us assume for the moment that only ‘null’ measurements are made, that is, no performance data is created and stored, but the ‘instrumented’ events are still detected. Since events are, in general, defined by their code location in direct instrumentation, the number of times an event occurs depends on how many times control passes through its code location. The event count is an important parameter in deciding on measurement optimization, regardless of whether profiling or tracing is used.

As the count for a particular event increases, the measurement overhead (from either profiling or tracing) for that event will accumulate. Since not all events will have the same count, the intrusions due to the overheads are distributed unevenly, in a sense, across the program and during execution. The intrusions may be manifested in different ways, and may lead to performance perturbations.

The only way to control the degree of measurement overhead is to control the event generation. That is, mechanisms must be used to *enable* and *disable* events at runtime. We call this technique *event throttling*. During program execution, instrumentation may be disabled in the program based on spatial, context, or location constraints. Spatial constraints deal with event count and frequency, context constraints consider program state, and location constraints involve event placement.

TAU allows the user to disable the instrumentation at runtime based on rules similar to the ones employed by the offline analysis of profiles using *tau_reduce*. For instance, the number of calls to each event can be examined at runtime. and when it exceeds a given user specified threshold (e.g., 100000 calls), it can be disabled [6]. This is an example of a count threshold. Disabling decisions can additionally consider measurement accuracy. For instance, if the per-call execution value for an event is below a certain threshold (e.g., 10 microseconds per call), the event is disabled. TAU disables events at runtime by adding them to the profile group (TAU_DISABLE). Subsequent calls to start or stop that event incur a minimal overhead of masking two bitmaps to determine enabled state.

Profilers based on measured profiling have timers that track routines of groups of one or more statements. A timer has a name and a profile group associated with it. A routine may belong to one or more profile groups. Performance analysis tools such as Vampir[7] and ParaProf[8] organize timer-based performance data by groups.

4.3 Group Based Control

During program execution, instrumentation may be disabled in the program based on spatial, context or location based constraints imposed. TAU provides an API for controlling instrumentation at runtime.

Logically related timers or phases may be grouped together by classifying these in a common profile group. Directory or file based association of routines is common. TAU provides a mechanism for enabling or disabling the program instrumentation based on groups. The top-level timer that is associated with main in C or C++, and the program unit in Fortran 90 belongs to a special group (TAU_DEFAULT) that is always enabled. The user may annotate the program at special points in the program based on certain conditions, to enable and/or disable instrumentation belonging to certain groups. These groups may be optionally specified on the command-line of the program as a set of groups that should be instrumented for the entire program. Limiting instrumentation based on groups, however, has the same disadvantages as knowing during program instrumentation which files or sets of routines to exclude from instrumentation.

4.4 Full Program Instrumentation Control

TAU allows the user to enable or disable all program instrumentation using the above instrumentation control API. This is useful for limiting the instrumentation (and generation of trace records) in parts of the program based on program dynamics. For regular iterative parallel applications where a program executes a sequence of iterations, it might be helpful to enable the instrumentation in a given subset. For instance, instead of enabling the instrumentation for tracing a million iterations of the program, it may be sufficient to trace the first and the last thousand iterations. The user may choose to disable the instrumentation based on the rank of a MPI process. For instance, it may be useful to limit the instrumentation for a large number of processors to only generating trace records for only one out of a hundred processors. This technique is similar to *sampling by space*[9]. TAU's trace merging and conversion utilities do not require all tasks to generate trace data. This technique cannot be applied effectively for MIMD applications where each task may have potentially different performance characteristics.

4.5 Context Based Control

TAU provides a unique depth limited instrumentation control option. A user can specify that a routines instrumentation be turned off when it executes beyond a given callpath depth. The limit may be specified as a runtime parameter. When this depth is specified as one, only the top level routine is active; at a depth of two, only the top level routine and the instrumented routine called directly by it are active and so on. When a routine executes below this threshold at some point in execution, and beyond this threshold at other points, only the former instance is recorded in the trace files. At the expense of truncating the performance information for those routines that execute beyond the given threshold, we can limit the performance data to the top few routines. The message communication events are not affected by this option.

4.6 Callpath Based Control

The KOJAK toolkit [3] includes the Expert tool that automates performance bottleneck diagnosis by examining communication events. In the analysis phase, it ignores routines that do not directly call MPI routines along a calling stack. To generate traces for Expert, it is useful to limit the instrumentation to only those routines that call an MPI routine. This is done by first configuring TAU to generate callpath profiles[2]. TAU allows a user to specify a callpath depth as a runtime parameter. All callpaths originating from a given instrumented routine, and extending to its parents are truncated when these exceed the threshold. So, the user sets a sufficiently high threshold of callpath depth so that every callpath reaches the top level routine. Then, a script parses the profile files and extracts the names of routines that directly or indirectly called an MPI routine. This list is then fed to the instrumentor as an include list, and it instruments only routines that had a calling path to the MPI routines. This technique can dramatically reduce the trace size for Expert. The drawback is that if a routine calls an MPI routine at some instances in its execution and does not invoke MPI calls at others, all of its instances are recorded, although Expert ignores those instances where it does not invoke MPI routines. This can potentially increase the trace file size and it requires a re-execution of the program with callpath profiling enabled.

4.7 Trace Based Control

It is possible to address the above problem by keeping track of all calls in an event buffer. When an MPI routine is executed, we need to examine the buffer and move trace records by eliminating those records that do not directly call the given MPI routine. This problem has a drawback that this scheme cannot work effectively with fixed size buffers that are commonly found in trace generation libraries. When a buffer overflow event takes place, all records are to be flushed to the trace file. However, if an MPI event has not taken place, it is unknown whether one will take place in the future or not. So, to preserve the trace information, we must increase the size of the trace buffer and keep processing the trace records. When it does encounter an MPI event, the trace buffer can be examined again and un-necessary instances of routines removed at runtime, and the buffers flushed to disk. This scheme does not sufficiently address the concerns as, the program could run out of memory in expanding the trace buffers and be forced to write the records to disk.

4.8 Callstack Based Control

To better address the previous requirement, TAU has introduced a callstack based runtime instrumentation control option for tracking only those instances of a routine that directly or indirectly invoke an MPI call. Trace records are generated for routines on the calling stack when an entry into an MPI routine (all MPI routines belong to a special group) is detected. When a routine entry takes place, we store the exact time it occurred on the callstack. Each routine on the callstack has a flag that indicates if it has been recorded in the trace file. When an MPI routine is started, we traverse the callstack recursively from the given routine and generate trace records if the routine has not been recorded. We stop when we encounter a routine that has been recorded. This limits the trace file to just those instances of events that are ancestors of an MPI call. By using elements of profiling and tracing together, we can better address efficient trace generation.

5 Conclusion

Parallel performance systems strive to build measurement systems as efficiently as possible. However, users can make poor instrumentation and measurement choices that lead to performance data proliferation and inaccuracies. Performance tools should support users in effective performance experimentation by providing mechanisms for optimizing instrumentation. This is true for specifying events and measurements to meet the objectives of the experiment, as well as controlling the degree of overhead and data accuracy.

The TAU performance system implements a robust set of instrumentation optimization methods. Some are discussed here. Other techniques implemented in TAU included compensation of instrumentation overhead, APIs for event grouping and control, context-based control based on callpath depths, and callstack-based control. It should be understood that all of the techniques work in parallel execution.

6 Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science.

References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A Portable Programming Interface for Performance Evaluation on Modern Processors,” *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.
2. S. Shende and A. D. Malony, “The TAU Parallel Performance System,” International Journal of High Performance Computing Applications, SAGE Publications, 20(2), pp. 287–331, Summer 2006.
3. B. Mohr, F. Wolf, “KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs,” *Euro-Par 2003 conference*, August 2003.
4. K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, “A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates,” SC 2000 conference, 2000.
5. B. Buck and J. Hollingsworth, “An API for Runtime Code Patching”, *Journal of High Performance Computing Applications*, pp. 317–329, 14(4), 2000.
6. A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, N. Trebon, “Advances in the TAU Performance System,” Chapter, “Performance Analysis and Grid Computing,” (Eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller), Kluwer, Norwell, MA, pp. 129-144, 2003.
7. H. Brunst, D. Kranzmüller, W. Nagel, “Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz”, *DAPSYS conference*, Kluwer, pp. 93–102, 2004.
8. R. Bell, A. D. Malony, and S. Shende, “A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis,” Proc. EUROPAR 2003 conference, LNCS 2790, Springer, pp. 17–26, 2003.
9. C. Mendes, and D. Reed, “Monitoring Large Systems via Statistical Sampling,” LACSI Symposium, Oct. 2002.