Chapter 4

Measurement and Analysis of Parallel Program Performance Using TAU and HPCToolkit

Allen D. Malony
University of Oregon

John Mellor-Crummey
Rice University

Sameer S. Shende
University of Oregon

4.1 Introduction ........................................ 50
4.2 Terminology ........................................ 52
4.3 Measurement Approaches ........................... 53
   4.3.1 Instrumentation ................................. 53
   4.3.2 Asynchronous Sampling ....................... 54
      4.3.2.1 Event-Based Sampling (EBS) .......... 55
      4.3.2.2 Instruction-Based Sampling (IBS) .... 55
   4.3.3 Contrasting Measurement Approaches ........ 56
   4.3.4 Performance Measurement in Practice ....... 57
4.4 HPCToolkit Performance Tools ..................... 58
   4.4.1 Design Principles .............................. 58
      4.4.1.1 From Principles to Practice .......... 60
   4.4.2 Measurement ................................... 61
   4.4.3 Analysis ....................................... 63
      4.4.3.1 Correlating Performance Metrics with
             Optimized Code ............................ 63
      4.4.3.2 Computed Metrics ....................... 64
      4.4.3.3 Identifying Scalability Bottlenecks in
             Parallel Programs ......................... 65
   4.4.4 Presentation .................................. 66
      4.4.4.1 hpcviewer ................................ 66
      Using hpcviewer ................................ 68
      4.4.4.2 hpctraceview ............................ 70
   4.4.5 Summary and Ongoing Work ................... 71
4.5 TAU Performance System ........................... 72
4.1 Introduction

Over the last several decades, platforms for high performance computing (HPC) have become increasingly complex. Today, the largest systems consist of tens of thousands of nodes. Each node is equipped with one or more multicore microprocessors. Individual processor cores support additional levels of parallelism including pipelined execution of multiple instructions, short vector operations, and simultaneous multithreading. Microprocessor-based nodes rely on deep multi-level memory hierarchies for managing latency and improving data bandwidth to processor cores. Subsystems for interprocessor communication and parallel I/O add to the overall complexity of these platforms.

Achieving high efficiency on such HPC systems with sophisticated applications is imperative for tackling “grand challenge” problems in engineering and science. Without careful attention, data-intensive scientific applications typically achieve less than 10% of peak performance on the sort of microprocessor-based parallel systems common today [264]. As a result, there is an urgent need for effective and scalable tools that can pinpoint performance and scalability bottlenecks in sophisticated applications running on HPC platforms. A challenge for tools is that bottlenecks may arise from a myriad of causes both within and across nodes. A performance analysis tool should enable one to pinpoint performance bottlenecks and identify their underlying causes.
The performance of a parallel program on a large-scale parallel system depends primarily on how its work, load balance, communication, and I/O costs scale with the size of its input data and the number of processors used. Additionally, one must consider how well an application makes use of the resources available on the target platform. Understanding the performance of a parallel program on a particular system requires measurement and analysis of many different aspects of program behavior. Some aspects worthy of attention include work (i.e., operation counts), resource consumption (e.g., memory bandwidth, communication bandwidth, I/O bandwidth), and inefficiency (e.g., contention, delays, and insufficient parallelism).

For performance tools to analyze aspects of a parallel program’s execution, they must be observable. For any given parallel system, not all aspects of interest will be measurable. For those that are, performance tools must gather metrics and correlate them to the program contexts in which they occur. It is also useful for tools to gather data about how execution behavior unfolds over time. While some aspects of program performance (e.g., load balance) can be diagnosed with simple summary information, others (e.g., serialization) require temporal or ordering information to identify them.

Performance tools must balance the need for information against the cost of obtaining it. Too little information leads to shallow and inaccurate analysis results; very detailed information is difficult to measure reliably and cumbersome to analyze. Unfortunately, there is no formal approach to determine, given a parallel performance evaluation problem, how to best observe a parallel execution to produce only the information essential for analysis. Furthermore, any parallel performance methodology ultimately is constrained by the capabilities of the mechanisms available for observing performance.

A parallel program execution can be regarded as a sequence of concurrent operations reflecting aspects of its computational activities. While the execution of instructions by processor cores is of course critical to high performance, other aspects of an execution are equally important, including activities in the memory system, interconnection network, and I/O subsystem. Operations of interest must be measured to be observed. Events are operations made visible to the measurement system. Not all operations can be monitored, and thus events only represent operations that are observable. In addition to encoding information about an operation, an event can be associated with the execution context in which it occurred and when it occurred in time.

A fundamental challenge for performance tools is how to gather detailed information to identify performance problems without unduly distorting an application’s execution. The nature of performance measurements, the cost of collecting them, and the analyses that they support follow directly from the approach used to gather performance data. In this chapter, we discuss several different approaches for measuring the performance of parallel programs and then describe in detail two leading performance toolkits that use rather different approaches for performance measurement and analysis. The next section provides some precise definitions of terms that one must understand.
to appreciate the commonality and differences between the measurement approaches described in this chapter. The chapter continues with descriptions of the HPCToolkit performance tools and the TAU Performance System. The section about each toolkit describes its features, outlines its capabilities, and demonstrates how it can be used to analyze the performance of large-scale applications. The chapter concludes with a brief discussion of open issues and challenges ahead for performance measurement and analysis on next generation parallel systems.

4.2 Terminology

Performance measurement relies on access to events. The terms synchronous and asynchronous are useful to distinguish the way an event is measured. A synchronous measurement of an event is one that is collected by code embedded in a program's control flow. An asynchronous measurement of an event is one that is collected by a signal handler invoked in response to an event occurrence.

Two orthogonal techniques used by performance tools are instrumentation and sampling. Performance tools for parallel systems typically augment programs with additional instructions (i.e., instrumentation) to monitor various aspects of program behavior. Most commonly, when people think of instrumentation, they think of using it to measure performance characteristics; this use of instrumentation is described in Section 4.3.1. However, instrumentation is also used by performance tools to observe certain significant operations, such as creation and destruction of processes and threads. Instrumentation of such operations is used by tools to ensure that necessary tool control operations, such as initialization or finalization of performance data for a thread, can be performed at appropriate times.

Monitoring all events in a program execution can be costly. To reduce the cost of monitoring, performance tools often use sampling to observe a fraction of program events. Most commonly, when people think of sampling in the context of performance tools, they think of asynchronous sampling, in which program control flow is periodically interrupted to associate an event or cost with the context in which it occurred. However, sampling need not be asynchronous. One can use sampling in conjunction with instrumentation injected into a program to make operations observable. For example, when instrumentation is encountered during program execution, the measurement code may choose to collect data only for some subset of the times it is executed. This style of synchronous sampling was used by Photon MPI [341], which used instrumentation of message passing routines, but used sampling to collect measurements for only a subset of messages transmitted. In general, this points
out the important distinction between how events are made observable and what is measured about the events.

There are two basic ways to record performance measurements of a program execution: as a stream of events, known as a trace, or a statistical summary, known as a profile. Each of these approaches has strengths and weaknesses. Traces can provide detailed insight into how an execution unfolds over time. While this is a powerful capability, it comes at a cost: for large-scale parallel systems, detailed traces can be enormous. In contrast, profiles are relatively compact—their size does not grow linearly with program execution time—and they can be recorded with low overhead. As a statistical summary, a profile by definition lacks detail about how events unfold over time in an execution; for this reason, profiles cannot yield insight into a program’s transient execution behavior. Despite their limitations, profiles can be surprisingly useful; for example, Section 4.4.3.3 describes how profiles can be used to pinpoint and quantify scalability bottlenecks with low overhead.

4.3 Measurement Approaches

In this section, we review two different measurement approaches used by performance tools: measurement based on instrumentation, and measurement based on asynchronous sampling.

4.3.1 Instrumentation

A straightforward approach to observing the performance of each thread in a parallel program execution is to directly instrument the program at multiple points with measurement code, known as a probe or caliper. Probes can be added to an application prior to compile time using source-level instrumentation, during compilation, or afterward using static [206] or dynamic [169] binary rewriting. The instrumentation process causes execution of probes to become part of the normal control flow of the program.

Probes can be used to measure different aspects of program performance. For instance, one can measure the time between a pair of probes inserted at procedure entry and exit to determine the duration of a procedure call. Modern microprocessors also contain hardware performance counters that can be used by probes to gather information. Using these counters, one can count events between probes that represent work (e.g., execution of instructions or floating-point operations), resource consumption (e.g., cycles or memory bus transactions), or inefficiency (e.g., cache misses or pipeline stall cycles). Typically, microprocessors support multiple counters so that several events can be monitored by probes in a single execution. Another use for probes is
to directly observe semantic information of interest associated with software operations (e.g., bytes allocated, messages sent, or bytes communicated).

While probes measuring software activity such as messages or data allocation can provide exact information, using timers or hardware counters to measure activity between pairs of probes is subject to perturbation caused by execution of the probes themselves. It is important to understand that when relying on probes for measurement, only program code instrumented with probes is actually observed and measured. To avoid incomplete measurement of an execution, binary rewriting can be used to insert probes everywhere they are needed, even in libraries for which no source code is available.

Probe-based measurement is a robust approach that has provided the foundation for many parallel performance tools, including Scalasca [373], Vampir [251], mpiP [342], and IPM [307]. It is the primary technique used by the TAU performance system discussed in Section 4.5.

### 4.3.2 Asynchronous Sampling

An effective, unobtrusive way to monitor a program’s execution is to use periodic sampling based on a recurring sample trigger. When a sample trigger fires, the operating system sends a signal to the executing program. When the program receives the signal, a signal handler installed in the program by a performance tool will record the program counter location where the signal was received and perhaps the current calling context and/or detailed information associated with the sample event (e.g., the effective address of a load or store). For parallel applications, one can set up sample triggers to monitor and signal each thread independently.

The recurring nature of a sample trigger means that typically a thread’s execution will be sampled many times, producing a histogram of program contexts for which samples were received. Asynchronous sampling can measure and attribute detailed performance information at a fine grain accurately as long as (1) code segments are executed repeatedly, (2) the execution is sufficiently long to collect a large number of samples, and (3) the sampling frequency is uncorrelated with a thread’s behavior. If these conditions are met, the distribution of samples will approximate the true distribution of the thread activity that the sample triggers are measuring. For modest sampling frequencies, the overhead and distortion introduced by asynchronous sampling is typically much lower than that introduced by probe-based instrumentation [143].

Sample-based measurement has proven to be a useful technique that is used by many performance measurement tools, including PerfSuite [201] and Sun’s Performance Analyzer [321]. It is also the primary technique used in the HPCToolkit system discussed in Section 4.4.

There are two principal types of asynchronous sampling: event-based sampling and instruction-based sampling. The difference between these approaches
lies in how samples are triggered. We briefly describe these two methods and the nature of their differences.

4.3.2.1 Event-Based Sampling (EBS)

Event-based sampling uses the occurrence of specific kinds of program events to trigger samples. Sample triggers based on different kinds of events measure different aspects of program performance. The classical approximation used by event-based sampling is that the program context in which a sample event is received is charged for all of the activity since the last sample event.

The most common type of event trigger is an interval timer. One can configure an interval timer to periodically interrupt a program execution with a specified sample period. The result of sampling an execution based on an interval timer is a quantitative record of where the program was observed to have spent time during its execution. For decades, this approach has been used by the Unix prof [333] and gprof [152] tools.

For over a decade, microprocessors have routinely provided hardware performance counters that can be used as event triggers. Using these counters, one can count events that represent work, resource consumption, or inefficiency. One can configure a counter to trigger an interrupt when a particular count threshold is reached. Typically, processors support multiple performance counters and each counter can be configured independently as a recurring event trigger; this enables multiple events to be monitored in a single execution.

A problem for event-based sampling using hardware performance counters on modern out-of-order microprocessors is that long pipelines, speculative execution, and the delay between when an event counter reaches a pre-determined threshold and when it triggers an interrupt make it difficult for performance monitoring hardware to precisely attribute an event to the instruction that caused it. As a result, information gathered using conventional event-based sampling on out-of-order processors is too imprecise to support fine-grain analysis of a program execution at the level of individual instructions.

An approach developed to address this problem on Intel processors is precise event-based sampling (PEBS) [315]. PEBS uses a special microassist and an associated microcode-assist service routine to precisely attribute an event sample. On today’s Intel Core microarchitecture, only nine events support PEBS and PEBS can only be used with one of the programmable hardware performance counters [179]. Because of the imprecision of conventional event-based sampling, unless PEBS is used, one should not expect instruction-level accuracy in the attribution of sample events. Despite the fact that conventional event-based samples are inaccurate at the instruction level, event-based samples without instruction-level accuracy are not useless: they can provide meaningful aggregate information at the loop level and above, except for loops that execute only a trivial number of instructions each time they are encountered.
4.3.2.2 Instruction-Based Sampling (IBS)

To provide a more comprehensive solution for performance monitoring on out-of-order processors, Dean et al. [110] developed an approach for the DEC Alpha 21264 that they called ProfileMe, which is based on sampling instructions. Periodically, an instruction is selected for monitoring before it is dispatched. As a monitored instruction moves through the execution pipeline, special hardware records a detailed record of significant events (e.g., whether the instruction suffered misses in the instruction cache, data cache, or TLB), latencies associated with pipeline stages, the effective address of a branch target or memory operand, and whether the instruction completed successfully or aborted. As a sampled instruction completes, an interrupt is triggered and the details of the instruction’s execution history are available for inspection. Support for exploiting ProfileMe’s instruction-based sampling was incorporated into DEC’s Dynamic Continuous Profiling Infrastructure, known as DCPI [32].

Today, AMD’s Family 10h Opteron processors support two flavors of instruction-based sampling: fetch sampling and op sampling [125]. For any sampled fetch, fetch sampling will log the fetch address, whether a fetch completed or not, associated instruction cache and TLB misses, the fetch latency, and the page size of the address translation [125]. Knowing when speculative fetches abort late can provide insight into an important source of speculation overhead. Op sampling provides information about instructions that complete. Information recorded about an instruction in flight includes the latency from when the instruction is tagged until it retires, the completion to retire latency, status bits about a branch’s behavior and the accuracy of branch predictions, the effective address (virtual and physical) for a load/store and details about the memory hierarchy response (e.g., cache and TLB miss; misalignment, and page size of address translation). This wealth of information can be used for detailed analysis of pipeline and memory hierarchy utilization.

Instruction-based sampling is used to diagnose processor performance issues by AMD’s CodeAnalyst performance tool [25]. In the past, HPCToolkit exploited instruction-based sampling measurements collected using ProfileMe on the Alpha 21264. In recent years, HPCToolkit’s focus has been exploiting event-based sampling because it is ubiquitously available whereas instruction-based sampling is not. At this writing, HPCToolkit is being extended to capture and analyze precise performance information gathered using instruction-based sampling.

4.3.3 Contrasting Measurement Approaches

In general, the aspect of program behavior or performance being studied defines the requirements for observation. No single measurement approach is inherently superior. Asynchronous sampling and instrumentation have different observational capabilities and each has their place. Asynchronous sampling is great for capturing and attributing detailed performance information at a
very fine grain with low overhead. However, only instrumentation is capable of capturing semantic information at particular points in a program’s execution. Detailed information about a communication event or how many times a procedure was invoked, or measuring costs within an interval defined by a pair of events, requires instrumented measurement.

When collecting performance measurements, measurement accuracy is always a concern. Accuracy is influenced by several factors, including both overhead and granularity. Any performance measurement, whether based on instrumentation or asynchronous sampling, incurs overhead when executing measurement code. Measurement overhead not only slows a program execution, it also can perturb behavior. Measurement overhead is generally linearly proportional to the total number of events measured. For asynchronous sampling, this depends upon the sampling period. For instrumentation, overhead depends upon the execution frequency of instrumented operations (e.g., procedures). Monitoring overhead also depends upon whether measurements are recorded in a profile or a trace.

A critical difference between asynchronous sampling and instrumentation is the relationship between measurement granularity and accuracy. When instrumentation is used to measure fine-grained events (i.e., where the quantity being measured is on the order of the measurement overhead) accuracy is compromised. Unfortunately, omitting instrumentation for fine-grain events introduces distortion of another kind: if an instrumentation-based performance measurement tool attributes each unit of cost (e.g., time), omitting instrumentation for fine-grain events has the effect of redistributing their costs to other measured events. In contrast, asynchronous sampling can measure and attribute detailed performance information at a fine grain accurately as long as the conditions for accurate sampling described in Section 4.3.2 are met.

Pragmatic concerns also influence the measurement approach used by performance tools. Measurement tools based on source-level instrumentation require little in the way of operating system support; for that reason, they can be available on new platforms immediately. However, source-level instrumentation can interfere with compiler optimizations as well as fail to accurately attribute costs to libraries that exist in binary only form. In contrast, asynchronous sampling can be applied to an entire executable, even dynamically-linked shared libraries. However, asynchronous sampling requires more operating system support; in the past, this dependency has delayed deployment of sampling-based tools on leading edge parallel systems such as Cray XT and IBM Blue Gene/P [324]. No approach is without drawbacks.

4.3.4 Performance Measurement in Practice

The nature of an execution under study and the types of performance problems suspected ultimately determines how performance tools are applied in practice. The goal of a performance tool is to collect measurements that yield insight into where a parallel program suffers from scalability losses and where
Performance losses accumulate due to a failure to make efficient use of processors, communication, or I/O. While performance bottlenecks may be identified with asynchronous sampling, fixing their causes may require understanding in detail the pattern of interactions between processes and threads, which can only be captured with instrumentation. For this reason, asynchronous sampling and instrumentation-based measurement are complementary. The next section provides an overview of the HPCTOOLKIT performance tools, which focuses on performance measurement and analysis based on asynchronous sampling; Section 4.5 describes the TAU Performance System, which focuses on performance measurement and analysis based on instrumentation.

4.4 HPCToolkit Performance Tools

This section provides an overview of the HPCTOOLKIT performance tools [22, 284] developed at Rice University. The principal design goals for HPCTOOLKIT were that it be simple to use and provide fine-grain detail about performance bottlenecks in complex parallel applications. Both of these goals have been achieved.

HPCTOOLKIT consists of tools for measuring the performance of fully optimized executables using asynchronous sampling, analyzing application binaries to understand the structure of optimized code, correlating measurements with both static and dynamic program structure, and presenting the resulting performance data in a top-down fashion to facilitate rapid analysis. Section 4.4.1 outlines the design principles that shaped HPCTOOLKIT’s development and provides an overview of some of HPCTOOLKIT’s key components. Sections 4.4.2 and 4.4.3, respectively, describe HPCTOOLKIT’s measurement and analysis components in more detail. Section 4.4.4 describes two user interfaces for analyzing performance data collected with HPCTOOLKIT. We illustrate HPCTOOLKIT’s capabilities for analyzing the performance of complex scientific applications using two codes: one that employs structured adaptive mesh refinement to model astrophysical thermonuclear flashes and another that uses a particle-in-cell method to simulate turbulent plasma in a tokamak. Section 4.4.5 offers some final thoughts and sketches our plans for enhancing HPCTOOLKIT for emerging parallel systems.

4.4.1 Design Principles

HPCTOOLKIT’s design is based on a set of complementary principles that form a coherent synthesis that is greater than their constituent parts. Although we have outlined these principles in earlier work [22, 235], it is useful to review them here so that the reader can understand the motivation behind HPCTOOLKIT’s approach to performance measurement and analysis.
Be language independent. Modern parallel scientific programs typically consist of an application core, along with a collection of libraries for communication, I/O, and mathematical operations. It is common for components to be written in different programming languages. To support measurement and attribution of program performance across components independent of the programming language in which they are expressed, HPCTOOLKIT works directly with application binaries.

Avoid instrumentation for measuring performance. The overhead of instrumentation can distort application performance through a variety of mechanisms [250]. In prior work [143, 327], we found that approaches that add instrumentation to every procedure to collect call graphs can slow execution by a factor of two to four across the SPEC integer benchmarks. To reduce instrumentation overhead, tools such as TAU intentionally refrain from instrumenting certain procedures [299]. However, the more this approach reduces overhead, the more it reduces precision. To avoid the high overhead common with routine-level instrumentation, as well as to provide loop and line-level performance details with low measurement overhead, HPCTOOLKIT uses asynchronous sampling as its principal method of measurement.

Avoid blind spots. Applications frequently link against libraries for which source code is not available. For instance, a compiler’s implementation of OpenMP typically relies on a runtime support library of this sort. To avoid systematic error, one must measure and attribute costs for routines in all such libraries. To handle routines in binary-only libraries, HPCTOOLKIT performs several types of binary analysis.

Context is essential for understanding modern software. Parallel applications and libraries often call communication primitives in many different contexts. Similarly, C++ programs may use the same algorithm or data structure template in many different contexts. In both cases, understanding performance requires understanding it in context. For this reason, HPCTOOLKIT supports call path profiling to attribute costs to the full calling contexts in which they are incurred.

Any one performance measure produces a myopic view. Measuring time or only one species of event seldom diagnoses a correctable performance problem. One set of metrics may be necessary to identify a problem and others may be necessary to diagnose its causes. For this reason, HPCTOOLKIT supports collection, correlation and presentation of multiple metrics.

Derived performance metrics are essential for effective analysis. Typical metrics such as elapsed time are useful for identifying program hot spots. However, tuning a program usually requires a measure of not where resources are consumed, but where they are consumed inefficiently. For this purpose, derived metrics, such as total aggregate losses accumulated as a result of differences between peak and actual performance, are far more useful than raw metrics such as operation counts. HPCTOOLKIT supports computation
of user-defined derived metrics and enables users to rank and sort program scopes using such metrics.

**Performance analysis should be top-down.** It is unreasonable to require users to wade through mountains of data to hunt for evidence of important problems. To make analysis of large programs tractable, performance tools should present measurement data in a hierarchical fashion, prioritize what appear to be important problems, and support a top-down analysis methodology that helps users quickly locate bottlenecks. HPCTOOLKIT’s user interface supports hierarchical presentation of performance data according to both static and dynamic contexts, along with ranking and sorting based on metrics.

**Hierarchical aggregation is vital.** Instruction-level parallelism in processor cores can make it difficult or expensive for hardware counters to precisely attribute particular events to specific instructions. However, even if fine-grain attribution of events is flawed, aggregate event counts within loops or procedures are typically accurate. HPCTOOLKIT’s hierarchical attribution and presentation of measurement data deftly addresses this issue; loop level information available with HPCTOOLKIT is particularly useful.

**Measurement and analysis must be scalable.** For performance tools to be useful on large-scale parallel systems, measurement and analysis techniques must scale to tens and even hundreds of thousands of threads. HPCTOOLKIT’s sampling-based measurements are compact and the data for large-scale systems is not unmanageably large. As we describe later, HPCTOOLKIT supports a novel approach for quantifying and pinpointing scalability bottlenecks conveniently on systems independent of scale.

### 4.4.1.1 From Principles to Practice

From these principles, we have devised a general methodology embodied by the workflow depicted in Figure 4.1. The workflow is organized around four principal activities: *measurement* of performance metrics while an application executes; *analysis* of application binaries to recover program structure; *correlation* of performance measurements with source code structure; and *presentation* of performance metrics and associated source code.

![HPCToolkit Workflow Diagram](image)

**FIGURE 4.1:** HPCToolkit workflow.

To use HPCToolkit to measure and analyze an application’s performance, one first compiles and links the application for a production run, using *full* optimization. Second, one launches an application with HPCToolkit’s
measurement tool, hpcrun, which uses asynchronous sampling to collect a performance profile. Third, one invokes hpcstruct, HPCTOOLKIT’s tool to analyze an application binary and recover information about files, functions, loops, and inlined code. Fourth, one uses hpcprof to combine information about an application’s structure with performance measurements to produce a performance database. Finally, one explores a performance database with HPCTOOLKIT’s hpcviewer graphical user interface.

At this level of detail, much of the HPCTOOLKIT workflow approximates other performance analysis systems, with the most unusual step being binary analysis. In the following sections, we outline how the methodological principles described above suggest several novel approaches to both accurate measurement (Section 4.4.2) and effective analysis (Section 4.4.3).

Our approach is accurate, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation), and effective, because it associates useful performance metrics (such as parallel idleness or memory bandwidth) with important source code abstractions (such as loops) as well as dynamic calling context.

4.4.2 Measurement

This section highlights the ways in which the methodological principles from Section 4.4.1 are applied to measurement. Without accurate performance measurements for fully optimized applications, analysis is unproductive. Consequently, one of the chief concerns for HPCTOOLKIT has been designing an accurate measurement approach that exposes low-level execution details while avoiding systematic measurement errors that come with high measurement overhead. For this reason, HPCTOOLKIT uses sampling (both synchronous and asynchronous) for low overhead measurement.

Asynchronous event triggers. Different kinds of event triggers measure different aspects of program performance. HPCTOOLKIT initiates asynchronous samples using either an interval timer, hardware performance counter events, or hardware support for instruction-based sampling. Hardware performance counters enable HPCTOOLKIT to profile events such as cache misses and issue-stall cycles. Instruction-based sampling enables precise fine-grained attribution of costs such as stall cycles, which is problematic with event-based sampling.

Synchronous event triggers. Synchronous triggers are generated via direct program action. Examples of interesting events for synchronous monitoring are lock acquisition and release, memory allocation, I/O, and inter-

\footnote{For the most detailed attribution of application performance data using HPCTOOLKIT, one should ensure that the compiler includes line map information in the object code it generates. While HPCTOOLKIT does not require this information, it helps users by improving the correlation between measurements and source code. Since compilers can usually provide line map information for fully optimized code, this need not require a special build process.}
Performance Tuning of Scientific Applications

process communication. For such events, one can measure quantities such as lock waiting time, bytes allocated, bytes written, or bytes communicated. To reduce the overhead of instrumentation-based synchronous measurements, HPCTOOLKIT uses sampling to record data for only a fraction of the monitored events.

Maintaining control over parallel applications. To control measurement of an executable, HPCTOOLKIT intercepts certain process control routines including those used to coordinate thread/process creation and destruction, signal handling, dynamic loading, and MPI initialization. To support measurement of unmodified, dynamically linked, optimized application binaries, HPCTOOLKIT uses the library preloading feature of modern dynamic loaders to preload a measurement library as an application is launched. With library preloading, process control routines defined by HPCTOOLKIT are called instead of their default implementations. To measure the performance of statically linked executables, HPCTOOLKIT provides a script for use at link time that arranges for process control routines to be intercepted during execution.

Call path profiling and tracing. Experience has shown that comprehensive performance analysis of modern modular software requires information about the full calling context in which costs are incurred. The calling context for a sample event is the set of procedure frames active on the call stack at the time the event trigger fires. The process of monitoring an execution to record the calling contexts in which event triggers fire is known as call path profiling. To provide insight into an application’s dynamic behavior, HPCTOOLKIT also offers the option to collect call path traces.

When synchronous or asynchronous event triggers fire, hpcrun records the full calling context for each event. A calling context collected by hpcrun is a list of instruction pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program address at which the event occurred. The rest of the list contains the return address for each active procedure frame. Rather than storing the call path independently for each sample event, hpcrun represents all of the call paths for events as a calling context tree (CCT) [31]. In a calling context tree, the path from the root of the tree to a node corresponds to a distinct call path observed during execution; a count at each node in the tree indicates the number of times that the path to that node was sampled. Since the calling context for a sample may be completely represented by a leaf node in the CCT, to record a complete trace of call path samples hpcrun simply records a sequence of (CCT node id, time stamp) tuples along with a CCT.

Coping with fully optimized binaries. Collecting a call path profile or trace requires capturing the calling context for each sample event. To capture the calling context for a sample event, hpcrun must be able to unwind the call stack at any point in a program’s execution. Obtaining the return address for a procedure frame that does not use a frame pointer is challenging since the frame may dynamically grow (i.e., when space is reserved for
the caller’s registers and local variables, the frame is extended with calls to `alloca`, or arguments to called procedures are pushed) and shrink (as space for the aforementioned purposes is deallocated) as the procedure executes. To cope with this situation, hpcrun uses a fast, on-the-fly binary analyzer that examines a routine’s machine instructions and computes how to unwind a stack frame for the procedure [327]. For each address in a routine, there must be a recipe for how to unwind. Different recipes may be needed for different intervals of addresses within the routine. Each interval ends in an instruction that changes the state of the routine’s stack frame. Each recipe describes (1) where to find the current frame’s return address, (2) how to recover the value of the stack pointer for the caller’s frame, and (3) how to recover the value that the base pointer register had in the caller’s frame. Once hpcrun computes unwind recipes for all intervals in a routine, it memorizes them for later reuse.

To compute unwind recipes with binary analysis, hpcrun must know where each routine starts and ends. When working with applications, one often encounters partially stripped libraries or executables that are missing information about function boundaries. To address this problem, hpcrun uses a binary analyzer to infer routine boundaries [327].

HPCTOOLKIT’s use of binary analysis for call stack unwinding has proven to be very effective, even for fully optimized code. At present, HPCTOOLKIT provides binary analysis for stack unwinding on the x86_64, Power, and MIPS architectures. A detailed study of HPCTOOLKIT’s x86_64 unwinder on versions of the SPEC CPU2006 benchmarks optimized with several different compilers showed that the unwinder was able to recover the calling context for all but a vanishingly small number of cases [327]. For other architectures (e.g., Itanium), HPCTOOLKIT uses the libunwind library [249] for unwinding.

Handling dynamic loading. Modern operating systems such as Linux enable programs to load and unload shared libraries at run time, a process known as dynamic loading. Dynamic loading presents the possibility that multiple functions may be mapped to the same address at different times during a program’s execution. During execution, hpcrun ensures that all measurements are attributed to the proper routine in such cases.

4.4.3 Analysis

This section describes HPCTOOLKIT’s general approach to analyzing performance measurements, correlating them with source code, and preparing them for presentation.

4.4.3.1 Correlating Performance Metrics with Optimized Code

To enable effective analysis, performance measurements of fully optimized programs must be correlated with important source code abstractions. Since measurements are made with reference to executables and shared libraries, for analysis it is necessary to map measurements back to the program source.
To correlate sample-based performance measurements with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure.\(^2\) HPCToolkit’s \texttt{hpceststruct} constructs this mapping using binary analysis; we call this process \textit{recovering program structure}.

\texttt{hpceststruct} focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, \texttt{hpceststruct} parses a load module’s machine instructions, reconstructs a control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and account for transformations to loops [327].\(^3\)

Several benefits naturally accrue from this approach. First, HPCToolkit can expose the structure of and assign metrics to what is actually executed, \textit{even if source code is unavailable}. For example, \texttt{hpceststruct}’s program structure naturally reveals transformations such as loop fusion and scalarized loops implementing Fortran 90 array notation. Similarly, it exposes calls to compiler support routines and wait loops in communication libraries of which one would otherwise be unaware. Finally, \texttt{hpcrun}’s function discovery heuristics identify procedures within stripped binaries.

HPCToolkit’s \texttt{hpcestprof} tool is used to correlate measurements collected using \texttt{hpcrun} with program structure information gathered by \texttt{hpceststruct}. The resulting performance database relates performance metrics to dynamic program structure (call chains) as well as static program structure (load modules, procedures, loops, inlined code, and source lines). A unique strength of this tool is that it can integrate both static and dynamic context information and relate performance to call paths that include representations for loops and inlined procedures. We recently developed an MPI [238] version of \texttt{hpcestprof} which scalably analyzes, correlates, and summarizes call path profiles from all threads and processes from an execution on a system with large-scale parallelism.

\subsection*{4.4.3.2 Computed Metrics}

Identifying performance problems and opportunities for tuning may require synthetic performance metrics. To identify where an algorithm is not effectively using hardware resources, one should compute a metric that reflects \textit{wasted} rather than consumed resources. For instance, when tuning a floating-point intensive scientific code, it is often less useful to know where the majority of the floating-point operations occur than where opportunities for executing floating-point operations were squandered, such as waiting for pipeline stalls or long-latency loads. A measure of how much floating point

\(^2\)This object to source code mapping should be contrasted with the binary’s line map, which (if present) is typically fundamentally line based.

\(^3\)Without line map information, \texttt{hpceststruct} can still identify procedures and loops, but is not able to account for inlining or loop transformations.
resources are underutilized can be computed as (cycles in a scope × peak flop/cycle − actual flops in that scope) and displaying this measure for loops and procedures. Many other useful quantities such as memory bandwidth available can be computed similarly.

### 4.4.3.3 Identifying Scalability Bottlenecks in Parallel Programs

![Figure 4.2: Differencing call path profiles to pinpoint scalability bottlenecks when weak scaling from $q$ to $p$ processors.](image)

If an execution of an SPMD program is sufficiently balanced and symmetric, the behavior of the program as a whole can be characterized by examining the scaling performance of any one of the processes or threads. If behavior is sufficiently stable over time, then an entire run can be characterized by analyzing profiles that are integrated over any appropriate time interval.

Differential profiling [231] is a strategy for analyzing multiple executions of a program by combining their execution profiles mathematically. A comparison of profiles from two executions yields information about where and how much the costs in the two executions differ. To pinpoint and quantify scalability bottlenecks for parallel programs and attribute them to calling contexts, we use HPCToolkit to compare call path profiles from a pair of executions using different levels of parallelism [95]. Consider two parallel executions of an application, one executed on $q$ processors and the second executed on $p$ processors, where $p > q$. In a weak scaling scenario, each processor in both executions computes on an identical amount of local data. If the application exhibits perfect weak scaling, then the total cost (e.g., execution time) should be the same in both executions. If every part of the application scales uniformly, then this equality should hold in each scope of the application.

Figure 4.2 pictorially shows an analysis of weak scaling by comparing two representative calling context trees (CCTs)\(^4\)—one tree from a process in each execution. For instance, the difference in cost incurred in the subtrees highlighted in the CCTs for $p$ and $q$ processors represents parallel overhead when scaling from $q$ to $p$ processors. This difference in cost for the subtree can be converted into percent scalability loss by dividing its cost by the inclusive

---

\(^4\)A CCT is a weighted tree in which each calling context is represented by a path from the root to a node and a node's weight represents the cost attributed to its associated context.
cost of the root node in the CCT for \( p \) processors and multiplying by 100. Computing inclusive scalability losses for each node in a CCT enables one to easily locate scalability bottlenecks in a top-down fashion by tracing patterns of losses down paths from the root.

In addition, we recently developed general techniques for effectively analyzing multithreaded applications using measurements based on both synchronous and asynchronous sampling [326, 328]. Using them, HPCToolkit can attribute precise measures of lock contention, parallel idleness, and parallel overhead to user-level calling contexts—even for a multithreaded language such as Cilk [142], which uses a work-stealing run-time system.

### 4.4.4 Presentation

This section describes hpcviewer and hpctraceview, HPCToolkit’s two presentation tools. We illustrate the functionality of these tools by applying them to analyze measurements of parallel executions of FLASH [126], a code that uses adaptive mesh refinement to model astrophysical thermonuclear flashes, and the Gyrokinetic Toroidal Code (GTC) [217], a particle-in-cell (PIC) code for simulating turbulent transport in fusion plasma in devices such as the International Thermomuclear Experimental Reactor.

#### 4.4.4.1 hpcviewer

![hpcviewer example](image)

**Figure 4.3**: Using HPCToolkit and differential profiling to analyze scaling losses (microseconds) for FLASH when running at 256 and 8192 cores on an IBM Blue Gene/P.
FIGURE 4.4: A Caller’s (bottom-up) view of scaling loss (wallclock) for FLASH when running at 256 and 8192 cores on an IBM Blue Gene/P.

HPCTOOLKIT’s hpcviewer user interface [23] presents performance metrics correlated to program structure (Section 4.4.3.1) and mapped to a program’s source code, if available. Figure 4.3 shows a snapshot of the hpcviewer user interface displaying data from several parallel executions of FLASH. The top pane of the hpcviewer interface shows program source code. The bottom pane associates a table of performance metrics with static or dynamic program contexts. hpcviewer provides three different views of performance measurements collected using call path profiling.

Calling context view. This top-down view associates an execution’s dynamic calling contexts with their costs. Using this view, one can readily see how much of an application’s cost was incurred by a function when called from a particular context. If finer detail is of interest, one can explore how the costs incurred by a call in a particular context are divided between the callee itself and the procedures it calls. HPCTOOLKIT distinguishes calling contexts precisely by individual call sites; this means that if a procedure \( g \) contains calls to procedure \( f \) in different places, each call represents a separate call-
ing context. Figure 4.3 shows a calling context view. This view is created by integrating static program structure (loops and inlined code) with dynamic calling contexts gathered by hpcrun. Analysis of the scaling study shown in the figure is discussed in the next section.

**Callers view.** This bottom-up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are called in more than one context. For instance, a message-passing program may call communication routines in many different calling contexts. The cost of any particular call will depend upon its context. Figure 4.4 shows a caller’s view of processes from two parallel runs of FLASH. Analysis of the example shown in this figure is discussed in the next section.

**Flat view.** This view organizes performance data according to an application’s static structure. All costs incurred in any calling context by a procedure are aggregated together in the flat view. This complements the calling context view, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

**hpcviewer** can present an arbitrary collection of performance metrics gathered during one or more runs, or compute derived metrics expressed as formulae with existing metrics as terms. For any given scope, **hpcviewer** computes both *exclusive* and *inclusive* metric values. Exclusive metrics only reflect costs for a scope itself; inclusive metrics reflect costs for the entire subtree rooted at that scope. Within a view, a user may order program scopes by sorting them using any performance metric. **hpcviewer** supports several convenient operations to facilitate analysis: revealing a *hot path* within the hierarchy below a scope, flattening out one or more levels of the static hierarchy, such as to facilitate comparison of costs between loops in different procedures, and zooming to focus on a particular scope and its children.

**Using hpcviewer**

We illustrate the capabilities of **hpcviewer** by using it to pinpoint scaling bottlenecks in FLASH by using differential profiling to analyze two (weak scaling) simulations of a white dwarf explosion by executing 256-core and 8192-core simulations on an IBM Blue Gene/P. For the 8192-core execution, both the input and the number of cores are 32x larger than the 256-core execution. With perfect weak scaling, we would expect identical run times and process call path profiles in both configurations.

Figure 4.3 shows a portion of the residual calling context tree, annotated with two metrics: “scaling loss” and “% scaling loss.” The former quantifies the scaling loss (in microseconds) while the latter expresses that loss as a percentage of total execution time (shown in scientific notation). The top-most entry in each metric column gives the aggregate metric value for the whole execution. A percentage to the right of a metric value indicates the magnitude of that particular value relative to the aggregate. Thus, for this execution,
there was a scaling loss of $1.65 \times 10^8 \mu$s, accounting for 24.4% of the execution.\footnote{A scaling loss of 24.4\% means that FLASH is executing at 75.6\% parallel efficiency on 8192 cores relative to its performance on 256 cores.} By sorting calling contexts according to metric values, we immediately see that the evolution phase (\texttt{Driver\_evolve\_Flash}) of the execution (highlighted with a black rectangle) has a scaling loss that accounts for 13.9\% of the total execution time on 8192 cores, which represents 57.1\% of the scaling losses in the execution.

To pinpoint the source of the scalability bottleneck in FLASH’s simulation, we use the “hot path” button to expand automatically the unambiguous portion of the hot call path according to this metric. Figure 4.3 shows this result.\texttt{HPCToolkit} identifies a loop (beginning at line 213), within its full dynamic calling context—a unique capability—that is responsible for 13.7\% of the scaling losses. This loop uses a ring-based all-to-all communication pattern known as a \textit{digital orrery} to update a processor’s knowledge about neighboring blocks of the adaptive mesh. Although FLASH only uses the orrery pattern to set up subsequent communication to fill guard cells, looping over all processes is inherently unscalable. Other scalability bottlenecks can be identified readily by repeating the “hot path” analysis for other subtrees in the computation where scaling losses are present. The power of \texttt{HPCToolkit}’s scalability analysis is apparent from the fact that it immediately pinpoints and quantifies the scaling loss of a key loop deep inside the application’s layers of abstractions.

To quickly understand where the scaling losses for the initialization and simulation phases are aggregated, we turn to the bottom-up caller’s view. The caller’s view apportions the cost of a procedure (in context) to its callers. We sort the caller’s view by the exclusive scaling loss metric, thus highlighting the scaling loss for each procedure in the application, exclusive of callees. Two routines in the BG/P communication library immediately emerge as responsible for the bulk of the scaling loss: \texttt{DCMF::Protocol::MultiSend::Tree::Allreduce\_Short\_Recv\_Post\_Message::advance} (\texttt{advance} for short) and \texttt{DCMF::BG\_Lock\_Manager::global\_Barrier\_Query\_Done} (\texttt{Query\_Done} for short). When we use \texttt{hpcviewer}’s caller’s view to look up the call chain from \texttt{advance}, we find routines that represent a call to \texttt{MPI\_Allreduce}. The first call, which accounts for 57\% of the scaling loss (14.1\% of run time), is highlighted in blue; the others, which are inconsequential, are hidden by an image overlay indicated by the thick horizontal black line. As the corresponding source code shows, this call to \texttt{MPI\_Allreduce} is a global max reduce for a scalar that occurs in code managing the adaptive mesh. \texttt{HPCToolkit} is uniquely able to pinpoint this one crucial call to \texttt{MPI\_Allreduce} and distinguish it from several others that occur in the application. Next, we peer up the hot path of call chains leading to \texttt{Query\_Done}. We hone in on one call to a barrier that disproportionately affects scaling. The barrier call site is within \texttt{Grid\_fill\_Guard\_Cells} and is visible at the bottom of Figure 4.4; it accounts for 13.6\% of the scaling loss (or 3.31\% of the run time). In a few simple steps, \texttt{HPCToolkit} has
enabled us to quickly pinpoint exactly two calls that account for about 70% of FLASH’s scaling loss on BG/P.

This brief study of FLASH shows how the measurement, analysis, attribution, and presentation capabilities of HPCTOOLKIT make it straightforward to pinpoint and quantify sources of scaling losses between different parallel configurations of an application.

4.4.4.2 hpctraceview

![hpctraceview image]

**FIGURE 4.5:** hpctraceview showing part of an execution trace for GTC. (See color insert.)

hpctraceview is a prototype visualization tool that was recently added to HPCTOOLKIT. hpctraceview renders space-time diagrams that show how a parallel execution unfolds over time. Figure 4.5 shows a screen snapshot from the space-time view pane of hpctraceview displaying an interval of execution for a hybrid MPI+OpenMP version of the Gyrokinetic Toroidal Code (GTC) [217] running on 64 processors. GTC is a particle-in-cell code for simulating turbulent transport in fusion plasma in devices such as the International Thermonuclear Experimental Reactor. The execution consists of 32 MPI processes with two OpenMP threads per process. The figure shows a set of time lines for threads. A thread’s activity over time unfolds left to right. Time lines for different threads are stacked top to bottom. Even numbered threads (starting from 0) represent MPI processes; odd-numbered threads represent OpenMP slave threads. Although hpctraceview’s space-time visualizations appear similar to those by many contemporary tools, the nature of its visualizations and the data upon which they are based is rather different.

Other performance tools, e.g., TAU and Paraver [276], render space-time diagrams based on data collected by embedded program instrumentation that *synchronously* records information about the entry and exit of program procedures, communication operations, and/or program phase markers. In contrast, hpctraceview’s traces are collected using *asynchronous* sampling. Each time line in hpctraceview represents a sequence of asynchronous samples taken over the life of a thread (or process) and each colored band represents a procedure frame that was observed by an asynchronous sample. While that
difference is important, it alone does not justify construction of a new engine for rendering space-time diagrams. A new viewer is necessary because \texttt{hpctraceview}'s samples are multi-level: each sample for a thread represents the entire call stack of procedures active when the sample event occurred. \texttt{hpctraceview} can in fact render traces at different levels of abstraction by displaying them at different call stack depths at the user's request.

Despite the fact that \texttt{hpctraceview}'s space-time diagrams are based on asynchronous samples, they can support analysis of transient behavior similar to their counterparts based on synchronous traces. Casual inspection of the space-time view in Figure 4.5 shows three complete repetitions of a “pattern” and part of a fourth. A closer look reveals that the first and third repetitions are somewhat different in nature than the second and fourth: the aforementioned patterns contain a band of yellow bars,\footnote{In a black and white copy, the yellow bars are those of lightest shade.} whereas the latter do not. These yellow bars represent a procedure that is called only in alternate iterations of the simulation. Space-time visualizations are good for spotting and understanding such time varying behavior. To relate such visual feedback back to the source program, one can use a pointing device to select a colored sample in the space-time view. Another pane in the \texttt{hpctraceview} GUI (not shown) will display the complete call path for a selected sample.

At present, \texttt{hpctraceview} is a proof-of-concept prototype. At this writing, enhancements are under way to enable it to render long traces from huge numbers of threads based on out-of-core data. Not only is the underlying data displayed by \texttt{hpctraceview} statistical in nature, its rendering is as well. If one tries to render more samples than will fit horizontally or more threads than will fit vertically, a rule is used to determine what color should be rendered for a pixel. Paraver has shown that with appropriate rules, such low-resolution visualizations can still convey important information about transient program behavior. We expect that \texttt{hpctraceview} will provide similar insights based on asynchronous call stack samples.

### 4.4.5 Summary and Ongoing Work

The key metric for parallel performance is scalability; this is especially true as parallel systems grow beyond the petascale systems of today. Consequently, there is an acute need for application scientists to understand and address scaling bottlenecks in their codes. \texttt{HPCToolkit}'s sampling-based performance measurements make it possible to pinpoint and quantify scaling bottlenecks both within and across nodes. Measurement overhead is low, analysis is rapid, and results are actionable.

In addition to understanding scaling losses, gaining insight into node performance bottlenecks on large-scale parallel systems is a problem of growing importance. Today's parallel systems typically have between four and sixteen cores per node. IBM's forthcoming Blue Waters system will increase...
the core count per node by packing four multithreaded eight-core processors into a multi-chip module [257]. By using event- or instruction-based sampling on hardware performance counters, one can distinguish between core performance bottlenecks caused by a variety of factors including inadequate instruction-level parallelism, memory latency, memory bandwidth, and contention. One can also use sampling to understand load imbalance [326] or lock contention [328] among the threads on a multiprocessor node.

Ongoing work in the HPCToolkit project is focused on making it easier to analyze performance data from huge numbers of cores in a top down fashion. HPCToolkit’s new MPI version of hpcprof [325] is capable of analyzing in parallel a set of execution profiles from many separate cores. Output of the tool provides summary statistics for all cores and a visual means for assessing similarities and differences in performance across the cores in a parallel system. HPCToolkit delivers these summaries along with access to fine-grain detail in the way that it always has: mapped back to the source code with full dynamic calling context augmented with information loops and inlined code. To cope with the volume of measurement data from very large-scale parallel systems, HPCToolkit’s presentation tools are being enhanced to support access to out-of-core data.

Key challenges ahead for HPCToolkit are to help diagnose causes of performance loss in parallel systems rather than just identifying where the losses occur, to provide deeper insight into how performance losses arise at the node level, and to offer with suggestions for how to tune your program to improve its performance. Ongoing work in this area includes a partnership with researchers at the University of Texas at Austin to build PerfExpert [70], an expert system for performance analysis, which aims to provide such guidance based on performance measurements collected by HPCToolkit.

4.5 TAU Performance System

The TAU Performance System® [225,300,301,339] is the product of over sixteen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. Although the University of Oregon is home to the TAU project, its success represents the combined efforts of researchers at the University of Oregon and colleagues at the Research Centre Jülich and Los Alamos National Laboratory, especially in TAU’s formative years. The following gives an overview of TAU’s architecture and current suite of tools. Best practices for its use are discussed and examples drawn from work highlighting some of TAU’s recent features.
4.5.1 TAU Performance System Design and Architecture

TAU is designed as a tool framework, wherein tool components and modules integrate and coordinate their operations using well-defined interfaces and data formats. The TAU framework architecture, shown in Figure 4.6, is organized into three primary layers: instrumentation, measurement, and analysis. Multiple modules exist within each layer that can be configured in a flexible manner by the user. The following sections discuss the layers in more detail, but first the overall design decisions that governed TAU’s development are presented.

**FIGURE 4.6:** TAU framework architecture.

TAU is a performance systems based on probe-based observation of events in the application, library, or system code. For any performance experiment using TAU, performance events of interest must be decided and the program must be instrumented. The role of the instrumentation layer in TAU is to insert calls to the TAU measurement system at code locations where the events occur. Since performance events of interest can be found at different places in the code, TAU provides a range of instrumentation capabilities to gather events from all these locations.

TAU supports both *atomic* events and *interval* events. An atomic event denotes a single action. When it occurs, the measurement system has the opportunity to obtain the performance data associated with that action at that time. In contrast, an interval event is really a pair of events: *begin* and *end*. The measurement system uses performance data obtained from each event to determine a performance result.

TAU uses probe-based measurement to support both *profiling* and *tracing*. Profiling methods compute performance statistics at runtime based on measurements of atomic or interval events. Tracing, on the other hand, records the measurement information for each event (including when it occurred) in a file for future analysis. In profiling, the number of recorded events is fixed, whereas tracing will generate a record for every event occurrence. The TAU
Performance Tuning of Scientific Applications

performance system includes tools for parallel profile analysis and targets parallel trace analysis tools from other research groups, principally Vampir [251] and Jumpshot [385].

Overall, TAU’s design has proven to be robust, sound, and highly adaptable to generations of parallel systems. In recent years, we have extended the TAU performance system architecture to support kernel-level performance integration [254], performance monitoring [253, 256], and collaboration through the TAU Portal [340]. The extensions have been moderate, mainly in the development of new interfaces for performance data access and reporting.

4.5.2 TAU Instrumentation

From TAU’s perspective, the execution of a program is regarded as a sequence of significant performance events. As events are triggered during execution, the instrumented probes engage the TAU performance measurement infrastructure to obtain the performance data. Logically, instrumentation is separated from measurement in TAU. The measurement options will determine what performance data is recorded and the type of measurement made for the events, whether profile or trace. Instrumentation is focused primarily on how the events are created and how code gets placed in the program.

4.5.2.1 Event Interface

The TAU event interface allows events to be defined, their visibility controlled, and their runtime data structures to be created. Each event has a type (atomic or interval), a group, and a unique event name. The event name is a character string and is a powerful way to encode event information. At runtime, TAU maps the event name to an efficient event ID for use during measurement. Events are created dynamically in TAU by providing the event interface with a unique event name. This makes it possible for runtime context information to be used in forming an event name (context-based events), or values of routine parameters to be used to distinguish call variants, (parameter-based events). TAU also supports phase and sample events.

The purpose of event control in TAU is to enable and disable a group of events at a coarse level. This allow the focus of instrumentation to be refined at runtime. All groups can be disabled and any set of groups can be selectively enabled. Similarly, all event groups can be enabled initially and then selectively disabled. It is also possible to individually enable and disable events. TAU uses this support internally to throttle high overhead events during measurement.

4.5.2.2 Instrumentation Mechanisms

Instrumentation can be introduced in a program at several levels of the program transformation process. In fact, it is important to realize that events and event information can be between levels and a complete performance view may
require contribution across levels [298]. For these reasons, TAU supports several instrumentation mechanisms based on the code type and transformation level: source (manual, preprocessor, library interposition), binary/dynamic, interpreter, component, and virtual machine. There are multiple factors that affect the choice of what level to instrument, including accessibility, flexibility, portability, concern for intrusion, and functionality. It is not a question of what level is “correct” because there are trade-offs for each and different events are visible at different levels. The goal in the TAU performance system is to provide support for several instrumentation mechanisms that might be useful together.

**Source Instrumentation**

Instrumenting at the source level is the most portable instrumentation approach. Source-level instrumentation can be placed at any point in the program and it allows a direct association between language-and program-level semantics and performance measurements. Using cross-language bindings, TAU implements the event interface in C, C++, Fortran, Java, and Python languages, and provides a higher-level specification in SIDL [194,302] for cross-language portability and deployment in component-based programming environments [54].

Programmers can use the event API to manually annotate the source code of their program. For certain application projects, this may be the preferred way to control precisely where instrumentation is placed. Of course, manual instrumentation can be tedious and error prone. To address these issues, we have developed an automatic source instrumentation tool, `tau_instrumentor`, for C, C++, and Fortran, based on the program database toolkit (PDT) [218]. The TAU source instrumentor tool can place probes at routine and class method entry/exit, on basic block boundaries, in outer loops, and as part of component proxies. PDT’s robust parsing and source analysis capabilities enable the TAU instrumentor to work with very large and complex source files and inserts probes at all possible points. We have translated the technology behind the TAU instrumentor into a more generic instrumentation component recently [148].

In contrast to manual instrumentation, automatic instrumentation needs direction on what performance events of interest should be instrumented in a particular performance experiment. TAU provides support for selective instrumentation in all automatic instrumentation schemes. An event specification file defines which of the possible performance events to instrument by grouping the event names in include and exclude lists. Regular expressions can be used in event name specifiers and file names can be given to restrict instrumentation focus. Selective instrumentation in TAU has proven invaluable as a means to customize performance experiments and to easily “select out” unwanted performance events, such as high frequency, small routines that may generate excessive measurement overhead.
Library wrapping is a form of source instrumentation whereby the original library routines are replaced by instrumented versions which in turn call the original routines. The problem is how to avoid modifying the library calling interface. Some libraries provide support for interposition, where an alternative name-shifted interface to the native library is provided and weak bindings are used for application code linking. Like other tools, TAU uses MPI’s support for interposition (PMPI [238]) for performance instrumentation purposes. A combination of atomic and interval events are used for MPI. The atomic events allow TAU to track the size of messages in certain routines, for instance, while the interval events capture performance during routine execution. TAU provides a performance instrumented PMPI library for both MPI-1 and MPI-2. In general, automatic library wrapping, with and without interposition, is possible with TAU’s instrumentation tools.

Source instrumentation can also be provided in source-to-source translation tools. TAU uses the Opri tool [245] for instrumenting OpenMP applications. Opri rewrites OpenMP directives to introduce instrumentation based on the POMP/POMP-2 [245] event model. TAU implements a POMP-compatible interface that allows OpenMP (POMP) events to be instantiated and measured.

Binary / Dynamic Instrumentation

Source instrumentation is possible only if the source code is available. To remove any dependencies on source access or compiler support, TAU can use the Dyninst package from the University of Wisconsin and the University of Maryland [69] to re-write the binary with instrumentation at routine boundaries. This takes place prior to code execution using the tau_run tool to inject the TAU measurement library as a shared object and conduct code instrumentation. TAU can also use Dyninst for online instrumentation of a running program. In each case, it is possible to use selective instrumentation for routine events.

Compiler-Based Instrumentation

Compilers from Intel, PGI, IBM, and GNU provide hooks for instrumentation of C, C++, and Fortran codes at the routine level. TAU works with all of these compilers automatically through environment parameters and compiler scripts. The compiler scripts can examine a selective instrumentation file to determine what source files to process. However, special care is needed for TAU to properly work with statically-linked and dynamically-linked shared objects. The problems arise in identifying routine names and involves the mapping of names to runtime addresses. All of the issues are correctly handled by TAU.
Interpreter-Based Instrumentation

TAU also supports the measurement of the runtime systems of both Java and Python. For these languages, TAU can interface dynamically at runtime with the virtual machine or interpreter to inject the TAU hooks into the context of the running program and capture information about events that take place. In these cases, no source code is modified by TAU and there is no need to re-compile the source code to introduce the instrumentation calls in the application. However, TAU also provides an instrumentation API for both Java and Python so that the user can set calipers around regions of interest.

4.5.2.3 Instrumentation Utilities

To deliver the richness of instrumentation TAU provides for performance observation, it helps to have utilities to reduce the impact on users. Where this is most evident is in building applications with source instrumentation. TAU provides a set of compiler scripts that can substitute for standard compilers in order to take control of the instrumentation process with little modification to application build environments. TAU comes with a PDT-based wrapper generator, tau_wrap, which allows library header files to be read and a new library wrapper header file created with preprocessor DEFINE macros to change routine names to TAU routines names. In this way, a new wrapped library is created with these routines instrumented. This is very similar to what is done by TAU for C malloc/free and I/O wrapping, except that tau_wrap can be used for any library.

4.5.3 TAU Measurement

The measurement system is the heart and soul of TAU. It has evolved over time to a highly robust, scalable infrastructure portable to all HPC platforms. The instrumentation layer defines which events will be measured and the measurement system selects which performance data metrics to observe. Performance experiments are created by selecting the key events of interest and by configuring measurement modules together to capture desired performance data [118]. TAU’s measurement system provides portable timing support, integration with hardware performance counters, parallel profiling and parallel tracing, runtime monitoring, and kernel-level measurement.

4.5.3.1 Measurement System Design

As shown in Figure 4.6, the design of the measurement system is flexible and modular. It is responsible for creating and managing performance events, making measurements from available performance data sources, and recording profile and trace data for each thread in the execution.

TAU’s measurement system has two core capabilities. First, the event management handles the registration and encoding of events as they are cre-
ated. New events are represented in an event table by instantiating a new event record, recording the event name, and linking in storage allocated for the event performance data. The event table is used for all atomic and interval events regardless of their complexity. Event type and context information are encoded in the event names. The TAU event management system hashes and maps these names to determine if an event has already occurred or needs to be created. Events are managed for every thread of execution in the application.

Second, a runtime representation called the event callstack captures the nesting relationship of interval performance events. It is a runtime measurement abstraction for managing the TAU performance state for use in both profiling and tracing. In particular, the event callstack is key for managing execution context, allowing TAU to associate this context with the events being measured.

The TAU measurement system implements another novel performance observation feature called performance mapping [298]. The ability to associate low-level performance measurements with higher-level execution semantics is important to understanding parallel performance data with respect to the application’s structure and dynamics. Performance mapping provides a mechanism whereby performance measurements, made for one instrumented event, can be associated with another (semantic) event at a different level of performance observation. TAU has implemented performance mapping as an integral part of its measurement system and uses it to implement sophisticated capabilities not found in other tools.

4.5.3.2 Parallel Profiling

Profiling characterizes the behavior of an application in terms of its aggregate performance metrics. Profiles are produced by calculating statistics for the selected measured performance data. Different statistics are kept for interval events and atomic events. For interval events, TAU computes exclusive and inclusive metrics for each event. Exclusive metrics report the performance when only the interval event is active, excluding nested events. Inclusive metrics report the exclusive performance of the interval event and all its nested events. Typically one source is measured (e.g., time), but the user may configure TAU with an option for using multiple counters and specify which are to be tracked during a single execution. For atomic events, the statistics measured include maxima, minima, mean, standard deviation, and the number of samples. When the program execution completes, a separate profile file is created for each thread of execution.

The TAU profiling system supports several profiling variants. The most basic and standard type of profiling is called flat profiling, which shows the exclusive and inclusive performance of each event, but provides no other performance information about events occurring when an interval is active (i.e., nested events). In contrast, TAU’s event path profiling can capture performance data with respect to event nesting relationships. In general, an event
path identifies a dynamic event nesting on the event stack of a certain length. An event path profile of length one is a flat profile. An event path profile of length two is often referred to as a callgraph profile. It provides performance data for all parents and children of an interval event. By specifying a path length in an environment variable, TAU can make a performance measurement for any event path of any length. Again, exclusive and inclusive performance information is kept in the measurement. The objective of callpath profiling is to gain more insight in how performance is distributed across the program’s execution.

While callpath profiling can reveal the distribution of performance events based on nesting relationships, it is equally interesting to observe performance data relative to an execution state. The concept of a phase is common in scientific applications, reflecting how developers think about the structural, logical, and numerical aspects of a computation. A phase can also be used to interpret performance. Phase profiling is an approach to profiling that measures performance relative to the phases of execution [226]. TAU supports an interface to create phases (phase events) and to mark their entry and exit. Internally in the TAU measurement system, when a phase, \( P \), is entered, all subsequent performance will be measured with respect to \( P \) until it exits. When phase profiles are recorded, a separate parallel profile is generated for each phase. Phases can be nested, in which case profiling follows normal scoping rules and is associated with the closest parent phase obtained by traversing up the callstack. When phase profiling is enabled, each thread of execution in an application has a default phase corresponding to the top level event. When phase profiling is not enabled, phases events acts just like interval events.

TAU also has support for recording of the current values of parallel profile measurements while the program is being executed in a form of a parallel profile snapshot [248]. The objective is to collect multiple parallel profile snapshots to generate a time-sequenced representation of the changing performance behavior of a program. In this manner, by analyzing a series of profile snapshot, temporal performance dynamics are revealed.

### 4.5.3.3 Tracing

Parallel profiling aggregates performance metrics for events, but cannot highlight the time varying aspect of the parallel execution. TAU implements robust parallel tracing support to log events in time-ordered tuples containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event, event-specific information, and other performance-related data (e.g., hardware counters). All performance events are available for tracing and a trace is created for every thread of execution. TAU will write these traces in its modern trace format as well as in VTF3 [296], OTF [191], and EPILOG [246] formats. TAU writes performance traces for post-mortem analysis, but also supports an interface for online trace access. This includes mechanisms for online and hierarchical trace merging [66, 68].
4.5.3.4 Measurement Overhead Control

TAU is a highly-engineered performance system and delivers very high measurement efficiencies and low measurement overhead. However, it is easy to naively construct an experiment that will significantly perturb the performance effect to be measured. TAU implements support to help the user manage the degree of performance instrumentation as a way to better control performance intrusion. The approach is to find performance events that have either poor measurement accuracy (i.e., they are small) or a high frequency of occurrence. Once these events are identified, the event selection mechanism described above can be used to reduce the instrumentation degree, thereby reducing performance intrusion in the next program run.

In addition, TAU offers two runtime techniques for profiling to address performance overhead. The first is event throttling. Here TAU regulates the active performance events by watching to see if performance intrusion is excessive. Environment variables `TAU_THROTTLE_PERCALL` and `TAU_THROTTLE_NUMCALLS` can be set to throttle when thresholds of per call overhead or number of calls are exceeded. The second is overhead compensation. Here TAU estimate how much time is spent in various profiling operations. TAU will then attempt to compensate for these profiling overheads while these events are being measured. This is accomplished by subtracting the estimated amount of time dedicated to profiling when calculating time spent for an event. TAU can also compensate for metrics besides time (e.g., floating-point operations). Overhead compensation requires small experiments to be run prior to execution. Its accuracy in practice will depend on how stable the estimated overhead is during the program run.

4.5.4 TAU Analysis

As the complexity of measuring parallel performance increases, the burden falls on analysis and visualization tools to interpret the performance information. If measurement is the heart and soul of the TAU performance system, the analysis tools bring TAU to life. As shown in Figure 4.6, TAU includes sophisticated tools for parallel profile analysis and leverages existing trace analysis functionality available in robust external tools, including the Vampir [251] and Expert/CUBE [313, 372] tools. This section focuses on TAU’s parallel profile analysis and parallel performance data mining.

4.5.4.1 Parallel Profile Management

The TAU performance measurement system is capable of producing parallel profiles for tens to hundreds of thousands of threads of execution consisting of hundreds of events. Scalable analysis tools are required to handle this large amount of detailed performance information. Figure 4.7 shows TAU’s parallel profile analysis environment. It consists of a framework for managing parallel
profile data, PerfDMF [174], and TAU’s parallel profile analysis tool, ParaProf [49]. The complete environment is implemented entirely in Java.

PerfDMF provides a common foundation for parsing, storing, and querying parallel profiles from multiple performance experiments. It supports the importing of profile data from tools other than TAU through the use of embedded translators. These are built with PerfDMF’s utilities and target a common, extensible parallel profile representation. Many profile formats are supported in addition to TAU profiles. The profile database component is the center of PerfDMF’s persistent data storage and builds on robust SQL relational database engines. Once imported to the database, profile data can also be exported to a common format for interchange between tools.

To facilitate performance analysis development, the PerfDMF architecture includes a well-documented data management API to abstract query and analysis operation into a more programmatic, non-SQL form. This layer is intended to complement the SQL interface, which is directly accessible by the analysis tools, with dynamic data management and higher-level query functions. Analysis programs can utilize this API for their implementation. Access to the SQL interface is provided using the Java Database Connectivity (JDBC) API.
4.5.4.2 Parallel Profile Analysis

The ParaProf parallel profile analysis tool [49] included in TAU is capable of processing the richness of parallel profile information produced by the measurement system, both in terms of the profile types (flat, callpath, phase, snapshots) as well as scale. ParaProf provides users with a highly graphical tool for viewing parallel profile data with respect to different viewing scopes and presentation methods. Profile data can be input directly from a PerfDMF database and multiple profiles can be analyzed simultaneously.

To get a brief sense of what ParaProf can produce, consider the AORSA (All Orders Spectral Algorithm) [183] simulation code, one of the major applications of DOE’s fusion modeling program. There are a 2-D and 3-D versions of AORSA. AORSA-2D provides a high-resolution, two-dimensional solutions for mode conversion and high harmonic fast wave heating in tokamak plasmas. AORSA-3D provides fully three-dimensional solutions of the integral wave equation for minority ion cyclotron heating in three dimensional stellarator plasmas. We ran AORSA-2D on the Cray XT5 (Jaguar) system at Oak Ridge National Laboratory to investigate its performance scaling behavior. The code was instrumented at routine boundaries and the outer loop level to evaluate the effect of multi-core CPUs on the overall code performance and to identify inefficiencies. Figure 4.8 shows a parallel profile from a 256-core execution. Here ParaProf has analyzed the full performance data and generated the three-dimensional view shown. This view is useful for highlighting performance features that reflect the relationships between events (left-side axis) and execution threads (bottom-side axis). In this case, the patterns reflect aspects of the model being simulated. Exclusive time is the performance metric (right-side axis).

![FIGURE 4.8: ParaProf displays of AORSA-2D performance on the ORNL Cray XT5 (Jaguar) system. (See color insert.)](image-url)
This example shows just a part of ParaProf’s capabilities. ParaProf can produce parallel profile information in the form of bargraphs, callgraphs, scalable histograms, and cumulative plots. ParaProf is also capable of integrating multiple performance profiles for the same performance experiment, but using different performance metrics for each. Phase profiles are also fully supported in ParaProf. Users can navigate easily through the phase hierarchy and compare the performance of one phase with another.

ParaProf is able to extend its analysis functionality in two ways. First, it can calculate derived statistics from arithmetic expressions of performance metrics. A simple example of this is “floating point operations per second” derived from two metrics, “floating point counts” and “time.” Second, ParaProf analysis workflows can be programmed with Python, using the Jython scripting interface.

4.5.4.3 Parallel Performance Data Mining

To provide more sophisticated performance analysis capabilities, we developed support for parallel performance data mining in TAU. PerfExplorer [172, 173] is a framework for performance data mining motivated by our interest in automated performance analysis and by our concern for extensible and reusable performance tool technology. PerfExplorer is built on PerfDMF and targets large-scale performance analysis for single experiments on thousands of processors and for multiple experiments from parametric studies. PerfExplorer addresses the need to manage large-scale data complexity using techniques such as clustering and dimensionality reduction, and the need to perform automated discovery of relevant data relationships using comparative and correlation analysis techniques. Such data mining operations are enabled in the PerfExplorer framework via an open, flexible interface to statistical analysis and computational packages, including WEKA [371], the R system [332], and Octave [128].

Figure 4.9 shows the architecture of TAU’s PerfExplorer system. It supports process control for scripting analysis processes, persistence for recording results of intermediate analysis, provenance mechanisms for retaining analysis results and history, metadata for encoding experiment context, and reasoning/rules for capturing relationships between performance.

PerfExplorer is able to analyze multi-experiment performance data and we have used it to understand scaling performance of the S3D application on the IBM BG/P (Intrepid) system at Argonne National Laboratory. S3D [320] is a turbulent combustion simulation system developed at the Combustion Research Facility at Sandia National Laboratories. We collected S3D parallel profile data on Intrepid for jobs ranging up to 12,000 cores. Figure 4.10(a) is a plot produced by PerfExplorer showing how major S3D events scale. Figure 4.10(b) is a plot showing the correlation of these events to total execution time. Events more strongly correlated with total execution time have a positive slope. These analyses were fully automated by PerfExplorer.
4.5.5 Summary and Future Work

The TAU Performance System® has undergone several incarnations in pursuit of its objectives: flexibility, portability, integration, interoperability, and scalability. The outcome is a robust technology suite that has significant coverage of the performance problem solving landscape for high-end computing. TAU follows an instrumentation-based performance observation methodology. This approach allows performance events relevant to a parallel program’s execution to be measured as they occur and the performance data obtained to be interpreted in the context of the computational semantics. However, issues of instrumentation scope and measurement intrusion have to be addressed. Throughout TAU’s lifetime, the project has pursued these issues aggressively and enhanced the technology in several ways to reduce the effects of the probe.

TAU is still evolving. We have added support for performance monitoring to TAU built on the Supermon [253] and MRNet [256] scalable monitoring infrastructures. The goal here is to enable opportunities for dynamic performance analysis by allowing global performance information to be accessed at runtime. We have extended our performance perspective to include observation of kernel operation and its effect on application performance [255]. This perspective will broaden to include parallel I/O and other sub-systems. TAU is also being extended to support performance analysis of heterogeneous parallel systems, in particular systems with GPU acceleration. In this regard, we are developing a CUDA performance measurement infrastructure [230]. Our current and long-term vision for TAU is targeted to whole performance evaluation for extreme scale optimization. Here we are working to integrate TAU in the system software stack and parallel application development environments being created for petascale and exascale systems.
Measurement and Analysis of Parallel Program Performance

Figure 4.10: PerfExplorer analysis of S3D performance on IBM BG/P. (a) S3D events scale. (b) correlation of events to execution time. (See color insert.)

4.6 Summary

Over the last two decades, research in the area of performance tools has delivered important ideas, approaches, and technical solutions. HPCToolkit and the TAU Performance System are representative of advances in the field. The impending emergence of extreme-scale parallel systems that incorporate heterogeneity will drive the development of next-generation tools. Performance
tools for extreme-scale systems will need new measurement capabilities to support new architectural features, new strategies for data management to cope with the torrent of information from these systems, and automated parallel analysis capabilities to sift through reams of performance data effectively and efficiently. With the growing complexity of emerging systems, it will be increasingly important for next generation performance tools to increasingly focus on providing higher-level guidance for performance tuning rather than just pinpointing performance problems.

Acknowledgments

The HPCToolkit project is supported by the Department of Energy’s Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762. Principal contributors to HPCToolkit include Laksono Adhianto, Michael Fagan, Mark Krentel, and Nathan Tallent.

The TAU project is supported by the U.S. Department of Energy, Office of Science, under contracts DE-FG03-01ER25501 and DE-FG02-03ER25561. It is also supported by the U.S. National Science Foundation, Software Development for Cyberinfrastructure (SDCI), under award NSF-SDCI-0722072. The work is further supported by U.S. Department of Defense, High-Performance Computing Modernization Program (HPCMP), Programming Environment and Training (PET) activities through Mississippi State University under the terms of Agreement No. #GSO4TO1BFC0060. The opinions expressed herein are those of the author(s) and do not necessarily reflect the views of the DoD or Mississippi State University.

This chapter reports results obtained using resources at both Argonne’s Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; and the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.