RICE UNIVERSITY

Performance Analysis for Parallel Programs

From Multicore to Petascale

by

Nathan Russell Tallent

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

[illegible]

John Mellor-Crummey, Chair
Professor of Computer Science and
Electrical & Computer Engineering

Vivek Sarkar

Vivek Sarkar
E.D. Butcher Professor of Computer Science
and Electrical & Computer Engineering

[illegible]

Peter Varman
Professor of Electrical & Computer
Engineering and Computer Science

[illegible]

Robert Fowler
Chief Domain Scientist, High Performance
Computing, Renaissance Computing Institute

HOUSTON, TEXAS

MARCH 2010

Abstract

Performance Analysis for Parallel Programs

From Multicore to Petascale

by

Nathan Russell Tallent

Cutting-edge science and engineering applications require petascale computing. Petascale computing platforms are characterized by both extreme parallelism (systems of hundreds of thousands to millions of cores) and hybrid parallelism (nodes with multicore chips). Consequently, to effectively use petascale resources, applications must exploit concurrency at both the node and system level — a difficult problem. The challenge of developing scalable petascale applications is only partially aided by existing languages and compilers. As a result, manual performance tuning is often necessary to identify and resolve poor parallel and serial efficiency.

Our thesis is that it is possible to achieve unique, accurate, and actionable insight into the performance of fully optimized parallel programs by measuring them with asynchronous-sampling-based call path profiles; attributing the resulting binary-level measurements to source code structure; analyzing measurements on-the-fly and postmortem to highlight performance inefficiencies; and presenting the resulting context-sensitive metrics in three complementary views. To support this thesis, we have developed several techniques for identifying performance problems in fully optimized serial, multithreaded and petascale programs. First, we describe how to attribute very precise (instruction-level) measurements to source-level static and dynamic contexts in fully optimized applications — all for an average run-time overhead of a few percent. We then generalize this work with the development of logical call path profiling and apply it to work-stealing-based applications. Second, we describe techniques for pinpointing and quantifying parallel inefficiencies such as parallel idleness, parallel overhead and lock contention in multithreaded executions. Third, we show how to diagnose scalability bottlenecks in petascale applications by scaling our our measurement, analysis and presentation tools to support large-scale executions. Finally, we provide a coherent framework for these techniques by sketching a unique and comprehensive performance analysis methodology. This work forms the basis of Rice University's HPCTOOLKIT performance tools.

Acknowledgments

This dissertation represents more than just my past few years of Computer Science graduate study. Seemingly by accident, I became involved in the early stages of the HPCToolkit performance tools project (née HPCView), inaugurated by John Mellor-Crummey. Consequently, before beginning any work toward this dissertation, I had helped build most of what became the proto HPCToolkit.

Nevertheless, I must highlight this dissertation's profound debt to others. The most generous share of credit goes to my advisor, John Mellor-Crummey, whose guidance and insight inform and infuse this work. I must also acknowledge several additional collaborators (in alphabetical order):

- Laksono Adhianto, who is the primary implementer of HPCToolkit's presentation tool, `hpcviewer`.

- Mike Fagan, who contributed to Chapter 3's on-the-fly binary analysis for unwinding call stacks and whose continual questions uncover weaknesses in our thinking.

- Mark Krentel, whose efforts and commitment to correctness have vastly improved HPCToolkit's ability to dynamically and statically monitor processes and threads.

- Allan Porterfield, who helped develop Chapter 6's blame shifting.

Additionally, I am grateful to (in chapter order):

- Chapter 3: Mark Charney and Robert Cohn of Intel who assisted with XED2 [38].

- Chapter 6: Robert Fowler for focusing our attention on MADNESS; Robert Harrison for helping us with his MADNESS code; and William Scherer for reminding us of Bacon's prior work on dual-representation locks and pointing out the similarity to STM contention managers.

- Chapter 7: Anshu Dubey and Chris Daley of the FLASH team; and Peter Lichtner, Glenn Hammond and other members of the PFLOTRAN team. Both teams graciously provided us with a copy of their respective code, configuration advice, and a test problem of interest.

Finally, I would like to acknowledge Robert Fowler, who was deeply involved with HPCToolkit while at Rice; Gabriel Marin, who was part of the original HPCToolkit team; Nathan Froyd, who worked on an early version of what is now HPCToolkit's measurement tool; and Cristian Coarfa, who first explored the scalability analysis technique used in Chapter 7.

* * *

Development of the HPCToolkit performance tools would not have been possible without without

- support from the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762;
- equipment purchased in part with funds from NSF Grant CNS-0421109;
- resources at the Argonne 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;
- resources at 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.

* * *

While academic supervision and financial support are necessary for dissertation research, they are not sufficient. To my parents, who lived like sojourners for their children; and to my grandfather Jack, who wanted to see this day: this dissertation is dedicated to you. To my wife, two sons and baby: we let the wind sweep away the world's wisdom and, despite a shoestring budget and some competition between midnight baby sitting and midnight paper writing, have been the happier for it. And finally, would science be possible without a starting point?

> For all knowledge proceeds from faith of whatever kind. You lean on God, you proceed from your own ego, or you hold fast to your ideal. The person who does not believe does not exist. At the very least, one who had nothing standing immediately firm before him could not find a point for his thinking to even begin. And how could someone whose thinking lacked a starting point ever investigate something *scientifically*?

Abraham Kuyper, October 20, 1880. [24, p. 486]

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

High performance computers have become enormously complex. Today's largest systems consist of tens of thousands of nodes and current plans call for a hundred thousand. Nodes themselves are equipped with one or more multicore microprocessors. Often these processor cores support additional levels of parallelism, such as hardware threads, short vector operations and pipelined execution of multiple instructions. 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. Recently, accelerators such as graphics chips and other co-processors have started to become more common on nodes. As the complexity of high performance computing (HPC) systems has grown, the complexity of applications has grown as well. Multi-scale and multi-physics applications are increasingly common, as are coupled applications.

Because HPC computing resources are limited and therefore precious, achieving top performance on leading-edge systems is critical. Unfortunately, existing compilers and other automatic techniques often fail to achieve top performance. The inability to harness such machines efficiently limits their ability to tackle the largest problems of interest. As a result, there is an urgent need for effective and scalable tools that can pinpoint a variety of performance and scalability bottlenecks in complex applications.

Our thesis is that it is possible to achieve unique, accurate, and actionable insight into the performance of fully optimized parallel programs by (1) measuring them with asynchronous-sampling-based call path profiles; (2) attributing the resulting binary-level measurements to source code structure; (3) analyzing measurements on-the-fly and post-mortem to highlight performance inefficiencies; and (4) presenting the resulting context-sensitive metrics in three complementary views. By actionable insight, we refer to insight into an application's performance that justifies concrete actions, such as determining how to resolve a performance bottleneck or deciding that there are no significant and worthwhile opportunities for performance improvement. By program performance, we refer to characterizing the performance of a particular execution. This is in contrast to constructing analytical models of a program that can be used for performance prediction on different inputs or architectures. Although we only focus on obtaining performance insight from a particular execution, it is often the case that fixing a bottleneck in a representative execution improves performance on different inputs and architectures.

To support this thesis, we have developed several techniques for identifying performance problems in fully optimized serial, multithreaded and petascale programs and have shown how these techniques form a coherent methodology. This work forms the basis of Rice University's HPCToolkit performance tools [119].

Methodology. To lay a foundation for our work, Chapter 2 sketches a performance analysis methodology. This methodology is based on a set of complementary principles that, while not novel in themselves, form a coherent synthesis that is greater than the constituent parts. Our methodology is *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation); *scalable*, because it can be used to effectively analyze the performance of a single thread or a large parallel execution; and *actionable*, because it associates insightful performance metrics (such as parallel inefficiency, scalability loss or memory bandwidth consumed) with important source code abstractions (such as loops) in their full dynamic calling contexts [1,95,96,138].

Measurement & Attribution. Chapters 3 and 4 present the measurement and attribution technology that serves as a foundation for the analysis techniques described in later chapters. In particular, we describe how to attribute very precise measurements to source-level static and dynamic contexts in fully optimized applications — all for an average run-time overhead of a few percent [141].

Modern programs frequently employ sophisticated modular designs. As a result, performance problems cannot be identified from metrics attributed to procedures in isolation; understanding code performance requires information about a procedure's calling context. Performance tools that attribute performance metrics to their full calling context are called *call path profilers* [67]. Current strategies for attributing calling-context-sensitive performance at the source level for fully optimized applications either compromise measurement accuracy, remain too close to the binary, or require custom compilers. Many tools measure using *instrumentation*, i.e., special instructions inserted directly into an application. Tools based on general instrumentation incur large overheads — often *factors* of at least two — that compromise accuracy. To avoid large overheads, we use asynchronous sampling. Sampling-based call path profilers must be able to unwind a program thread's call stack and then attribute the result back to source code. Existing sampling-based call path profilers are unable to reliably do this for fully optimized applications.

To understand the performance of fully optimized modular code, Chapter 3 describes two novel binary analysis techniques for asynchronous-sampling-based call path profilers: (1) *on-the-fly* analysis of optimized machine code to enable minimally intrusive measurements qualified by their full dynamic calling contexts; and (2) post-mortem analysis of optimized machine code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code [141]. By combining the recovered static program structure with dynamic calling context information, HPCToolkit can accurately attribute performance metrics to procedures, loops, and inlined instances of procedures in their full calling contexts.

Over the past decade, high-level multithreaded programming models such as Cilk [58] have emerged to simplify the development of multithreaded programs. These programming models raise the level of abstraction of parallel programming by partitioning the problem into two parts: the programmer is responsible for expressing the logical concurrency in a program and a run-time system is responsible for partitioning and mapping parallel work efficiently onto a pool of threads for execution.

To apply our work on call path profiling to parallel programming models such as Cilk, Chapter 4 generalizes the notion of call path profiling to *logical call path profiling* [140,142]. For many high-level programming models, using call path profiling to associate costs with the context in which they are incurred is not as simple as it sounds. Standard call path profiling assumes a thread's call stack can be used as a proxy for the full source-level calling context of a particular point in its execution. However, for applications written in Cilk, which uses a work-stealing scheduler to partition and map work onto a thread pool, the stack of native procedure frames active within a thread represents only a suffix of the calling context. Moreover, Cilk's work-stealing scheduler causes calling contexts to become separated in space and time as procedure frames migrate between threads as work is distributed (stolen). As a result, a standard call path profile of a Cilk execution shows fragments of call paths mapped to each of the threads in the scheduler's thread pool. In contrast, a logical call path profile attributes arbitrary performance metrics to source-level contexts for a Cilk application, even in the presence of work distribution (stealing). Accomplishing this requires bridging the gap between the expression of logical concurrency in a program and its realization at run time as the program's work is dynamically partitioned and scheduled onto a pool of threads. A later chapter uses these results to attribute metrics that reflect parallel inefficiency to source-level calling contexts in Cilk.

Analysis of Multithreaded Executions. Chapters 5 and 6 focus on performance analysis of multithreaded executions. Understanding why the performance of a multithreaded program does not improve linearly with the number of cores in a shared-memory multicore node is increasingly important. For instance, nodes on the Department of Energy's 'leadership class' machines currently contain 4-12 cores and nodes on less-balanced large-scale systems will soon contain scores of threaded cores. To address these issues, we developed techniques for pinpointing and quantifying parallel inefficiencies in work-stealing-based and lock-based multithreaded applications.

To understand the performance of work-stealing-based programs, Chapter 5 develops techniques for quantifying parallel idleness and overhead and pinpointing them to their logical calling context [140, 142]. *Parallel idleness* occurs when threads are stalled and unable to work, whereas *parallel overhead* occurs when a thread performs miscellaneous work other than the user's computation. These metrics enable one to identify areas of an application where concurrency should be increased (to reduce idleness), decreased (to reduce overhead), or where the present parallelization is hopeless (where idleness and overhead are both high). By basing our techniques on asynchronous sampling, we can measure and attribute parallel idleness for minimal overhead ($< 5\%$). By using a combination of compiler support and post-mortem binary analysis, we can measure parallel overhead without any measurement cost beyond normal profiling. These techniques apply broadly to high-level programming models such as Cilk and OpenMP. Our results provide unique insight into the performance of complex modular code where existing techniques fail.

Although higher-level parallel programming models are attractive, most multithreaded codes use locks to coordinate access to shared data. Indeed, fine-grain locking remains the gold standard for performance. In addition, locks are used to implement higher-level abstractions such as software transactional memory [50]. The chief cause of parallel inefficiency in lock-based programs is lock contention. Being able to quantify and attribute lock contention is important for understanding how to improve a multithreaded program's scalability.

Chapter 6 proposes and evaluates three strategies for gaining insight into performance losses due to lock contention [144]. First, we consider using a straightforward strategy based on call path profiling to attribute idle time and show that it fails to yield insight into lock contention. Second, we consider an approach that builds on the strategy of Chapter 5 for analyzing idleness in work stealing computations; we show that this strategy does not work well for understanding lock contention. Finally, we propose a new technique for measurement and analysis of lock contention that uses data associated with locks to blame lock holders for the idleness of spinning threads. Our approach incurs less than 5% overhead for a non-trivial execution of a quantum chemistry code that makes extensive use of locking (65M distinct locks, a maximum of 340K live locks, and an average of 30K lock acquisitions per second per thread) and attributes lock contention to its full static and dynamic calling contexts. Our strategy is distributed and should scale well to systems with larger core counts.

Analysis & Presentation of Petascale Executions. Finally, Chapter 7 focuses on the performance analysis and presentation of petascale executions. The first petascale systems became available in 2009. To compute at the petaflop level — a thousand trillion floating point operations per second — petascale systems have hundreds of thousands of processor cores. Because of the challenge of using petascale computing platforms effectively, there is an acute need for application scientists to resolve scaling bottlenecks. To help address these issues, we develop new features for HPCToolkit to support the low-overhead (1–2%) collection of precise measurements on emerging petascale platforms [2, 56, 143]. Additionally, we show how to scalably analyze and present data from large-scale runs, including how to scalably compute a large set of derived metrics in parallel. With these new features, we show how to use HPCToolkit's call path sampling to pinpoint and quantify both scaling and node performance bottlenecks. By applying this method to several emerging petascale applications on the Cray XT and IBM BlueGene/P platforms, HPCToolkit identifies specific source lines — in their full calling context — associated with performance bottlenecks in these codes. This information is exactly what application developers need to know to improve their applications to take full advantage of the power of petascale systems.

* * *

The principal goal of performance analysis is to determine if a production application has any performance bottlenecks and, if so, to provide actionable insight into what should be done next. This at least involves highlighting, within source code, bottlenecks that are both profitable and worthwhile to resolve. However, achieving such actionable insight is difficult. Since performance measurement typically occurs within a program's execution space, the very act of measuring disturbs a program's execution. Consequently, there is a natural tension between measurement precision and accuracy: the more precise measurements are, the more difficult to obtain accuracy. Yet, both precise and accurate measurements are usually prerequisites for actionable insight into program performance.

As a result, one of the principal focuses of this thesis has been the design and implementation of techniques for providing accurate fine-grain measurements of production applications running at scale. For measurements to be accurate, performance tools must avoid introducing measurement error, including error from overhead. For tools to be useful on production applications, they cannot significantly increase execution time by inducing large overhead. HPCTOOLKIT is able to attribute precise measurements — statements in their full static and dynamic calling context — with low, controllable overhead and high accuracy.

A second overriding theme has been constructing insightful metrics from these accurate fine-grain measurements. We have accomplished this in two ways. First, we have highlighted sources of inefficiency in a program rather than where it spends its time. Second, we have developed ways to blame sections of source code for causing inefficient computation rather than reporting where that inefficiency is manifested.

Chapter 2

A Methodology for Performance Analysis

2.1 Introduction

In this chapter, we sketch HPCTOOLKIT's unique and comprehensive methodology for analyzing the performance of parallel programs [1,95,96,138]. The methodology is based on a set of complementary principles that, while not novel in themselves, form a coherent synthesis that is greater than the constituent parts. This methodology is (1) *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation); (2) *scalable*, because it can be used to effectively analyze the performance of a single thread or a large parallel code; and (3) *actionable*, because it associates insightful performance metrics (such as parallel inefficiency, scalability loss or memory bandwidth) with important source code abstractions (such as loops) in their full calling context. These emphases have resulted in measurement techniques that incur low overhead, preserve low-level detail, and scale to large systems; metrics that highlight inefficiency rather than simply resource usage; and attribution, analysis and presentation techniques that yield insight by projecting low-level measurements to much higher levels of abstraction.

The methodology we describe is a significant development of prior work with Mellor-Crummey, Fowler and Marin [93] and Froyd [60, 61]. Since this prior work, HPCTOOLKIT's measurement, attribution, analysis, and presentation abilities have been radically advanced and its ability to effectively analyze multithreaded and large-scale parallel executions is entirely new. Accordingly, we now present a full-orbed methodology for performance analysis of parallel programs. As a companion to our methodology, Appendix A presents an analysis, the first to our knowledge, of statistical sampling as a means of obtaining a thread-based profile.

This chapter is organized as follows. Section 2.2 enumerates several principles of performance analysis and then Section 2.3 applies those principles to form a methodology based on accurate measurement, source-level attribution, effective analysis and insightful presentation. Finally, Section 2.4 discuses related work and Section 2.5 discusses the chapter's main themes.

2.2 Principles of Performance Analysis

The following principles form the basis of our methodology.

The goal is actionable insight.

The goal of performance analysis is actionable insight. By actionable insight, we refer to insight into an application's performance that justifies concrete actions such as determining how to resolve a performance bottleneck or deciding that there are no significant and worthwhile opportunities for performance improvement. Although obtaining insight requires accurate and scalable measurement techniques, such techniques are only a means to an end.

One way of stating this principle more concretely is to observe that the role of performance tools is not so much to highlight program hot spots, but to pinpoint and diagnose bottlenecks. For instance, the most important thing to know for a parallel application is whether there are parallel scaling bottlenecks at any architectural level. If both inter-node and intra-node parallelism are good, the next step is to determine if the application making the most of a processor core. What are the rate-limiting factors for the application? Is there a mismatch between the application's needs and the computing system's capabilities? Finally, when a bottleneck is identified, it is important to know two things about it: What the expected benefit of resolving the bottleneck is and what level of effort will be necessary to do so.

Be language independent.

Modern parallel scientific programs, on one hand, often have a numerical core written in some modern dialect of Fortran, but on the other hand, leverage frameworks and communication libraries written in C or C++. For this reason, the ability to analyze multi-lingual programs is essential. To provide language independence, HPCToolkit works directly with application binaries rather than source code.

Avoid code instrumentation for measurement.

We define instrumentation to be any addition to a program that is directly and synchronously invoked during the course of normal program execution; it can be contrasted with the indirect execution of an asynchronous signal handler. Although instrumentation can take several forms — source code, compiler-inserted or binary — all forms can distort application performance through a variety of mechanisms [109]. The most common problem with instrumentation is overhead, which distorts measurements. The classic tool Gprof [65], which uses compiler-inserted instrumentation, induced an average overhead of over 100% on the SPEC 2000 integer benchmarks [60]. Intel's VTune [77], which uses static binary instrumentation, claims an average overhead of a factor of eight. Intel's Performance Tuning Utility (PTU) [7] includes a call graph profiler based on Pin's dynamic binary instrumentation [88]; we found that it yielded an average overhead of over 400% on the SPEC 2006 integer benchmarks [141].

Another problem with instrumentation is the trade-off between accuracy and precision. While all measurement approaches must address this trade-off, the problem is particularly acute for instrumentation. For example, tools such as TAU [128] may intentionally refrain from instrumenting certain procedures to avoid large overheads. A common selective instrumentation technique is to ignore small frequently executed procedures. The more this approach reduces overhead, the more it reduces precision. Moreover, the ignored procedures may be just the synchronization library routines that are critical performance bottlenecks.

Tools that rely on source code instrumentation can distort application performance in even more subtle ways. Because instrumentation often has side effects, it interferes with inlining and template optimization [139]; some compiler-based instrumentation also disables compiler optimizations. Additionally, source code instrumentation is fundamentally unable to measure procedures for which source is unavailable, such as from binary-only libraries. This results in blind spots.

To avoid instrumentation's pitfalls, HPCToolkit uses statistical sampling to measure performance. When possible, we prefer using asynchronous signals to generate sample events. However, in some cases an event is fundamentally and synchronously tied to program execution. For example, our analysis of lock contention (Chapter 6) requires intercepting *every* invocation of `lock` and `unlock` — potentially frequent events. To minimize the distorting overhead of instrumentation in these cases, HPCToolkit applies sampling to instrumentation, i.e., it uses very lightweight instrumentation to periodically switch to short periods of heavyweight instrumentation. Another example of a fundamentally synchronous event is an application thread's entry and exit point. HPCToolkit intercepts these entry and exit points to initialize and finalize statistical sampling.

Avoid blind spots.

Production applications frequently link against fully optimized and even partially stripped binaries, e.g., math and communication libraries, for which source code is not available. To avoid systematic error, one must measure costs for routines in these libraries. However, fully optimized binaries create challenges for asynchronous-sampling-based call path profiling and hierarchical aggregation of performance measurements. To deftly handle optimized and stripped binaries, HPCToolkit performs several types of binary analysis that are summarized in Sections 2.3.1 and 2.3.2.

Context is essential for understanding modular software.

Modern software design emphasizes modularity through layers of functional abstraction, generics and object-orientation. In such programs, it is important to attribute the costs incurred by each procedure to the different contexts in which the procedure is called. The costs incurred for calls to communication primitives (e.g., `MPI_Wait`) or for code that results from instantiating C++ data structure templates can vary widely depending upon their calling context. When considering how to implement a set, different choices may be appropriate for different contexts. For instance, a bit vector can be a good implementation where a dense set is needed, but other representations are preferable for sparse sets. Because there are often layered implementations within applications and libraries, it is insufficient either to measure at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCToolkit supports call path profiling [67] to attribute performance metrics to the full calling contexts in which they are incurred.

Although we focus on calling contexts, it is possible to collect other forms of contextual information. If calling context represents inter-procedural control flow, it is also possible to additionally collect intra-procedural context representing the path of flow within a procedure's control flow graph [53, 71]. Yet another piece of context is the value of a state variable or a particular procedure's input. For example, one may wish to distinguish communication calls by message size. Finally, it is possible to distinguish between context *instances* by qualifying all measurements by time, or more generally, by any monotonically increasing resource. This is also known as tracing.

While more contextual information theoretically produces more fine-grained measurement data, this is only true if there is a reasonable balance between accuracy and the desired level of measurement (precision). We have focused on calling context because it is very useful and becuase we have developed highly accurate low-overhead techniques for gathering it. Moreover, we have developed fully post-mortem techniques for fusing static program structure — including loop nests — with dynamic calling contexts. Such information enables HPCToolkit to expose the most important aspect of intra-procedural flow without any measurement overhead.

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 another set may be necessary to diagnose its causes. For example, counts of cache misses indicate problems only if both the miss *rate* is high and the latencies of the misses are not hidden. HPCToolkit supports collection, correlation and presentation of multiple metrics.

Metrics pinpointing inefficiency 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 *not* of where resources are consumed, but where they are consumed *inefficiently*. For this purpose, derived measures such as the difference between peak and actual performance are far more useful than raw data such as operation counts. HPCToolkit supports the computation of user-defined derived metrics and enables users to rank and sort program scopes using such metrics. In addition, HPCToolkit can compute metrics that blame sections of source code for *causing* inefficient computation rather than simply reporting where that inefficiency is manifested.

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 without the need to wade through irrelevant details. HPCToolkit's presentation tool 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.

The amount of 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, total event counts within loops or procedures will typically be accurate. Moreover, in most cases, it is the balance of operations within loops that matters — for instance, the ratio between floating point arithmetic and memory operations. HPCToolkit's

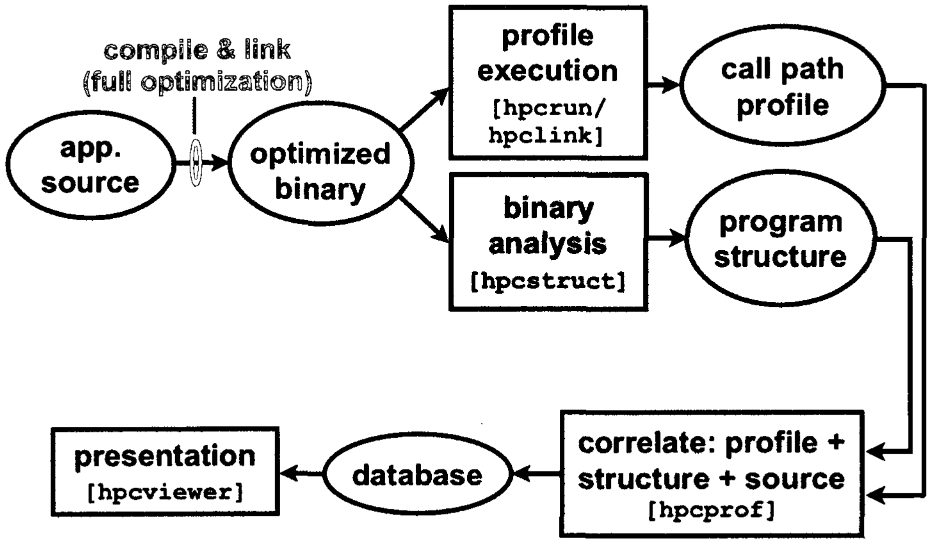


Figure 2.1: Overview of HPCToolkit tool's workflow.

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.

Large parallel systems may have tens of thousands of nodes, each equipped with one or more multicore processors. For performance tools to be useful on these systems, measurement and analysis techniques must scale to tens to hundreds of thousands of threads. HPCToolkit's sampling-based measurements are compact and the data for large-scale executions is not unmanageably large.

2.3 From Principles to Practical Methods

From these principles, we have devised a general methodology summarized by the workflow depicted in Figure 2.1. The workflow is organized around four principal capabilities:

1. *measurement* of context-sensitive performance metrics while an application executes;
2. *binary analysis* to recover program structure from application binaries;
3. *attribution* of performance metrics by correlating dynamic performance metrics with static program structure; and
4. *presentation* of performance metrics and associated source code.

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 statistical sampling to collect a performance profile. Third, one invokes `hpcstruct`, HPCToolkit's tool for analyzing an application binary to recover information about files, procedures, loops, and inlined code.[1] Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's `hpcviewer` graphical presentation tool.

At this level of detail, much of the HPCToolkit workflow approximates other performance analysis tools, with the most unusual step being binary analysis. However, the high level of the workflow discussion masks several novel aspects of HPCToolkit's methodology. In the following sections, we (1) sketch how the principles described above inform our methodology and (2) highlight several novel approaches to accurate measurement (Section 2.3.1), source-level attribution (Section 2.3.2), effective analysis (Section 2.3.3) and insightful presentation (Section 2.3.4).

[1]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 need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully optimized code, this requirement need not require a special build process.

2.3.1 Measurement

Without accurate performance measurements for fully optimized applications, analysis is unproductive. Consequently, one of our chief concerns has been designing an accurate measurement approach that simultaneously exposes low-level execution details while avoiding systematic measurement error, either through large overheads or through systematic dilation of execution. For this reason, HPCToolkit avoids instrumentation and favors *statistical sampling.*

Statistical sampling

Statistical sampling is a method for estimating performance metrics for a whole execution from a sample of that execution. There are two basic technique types for sampling a program's execution: asynchronous and synchronous.

HPCToolkit primarily relies on asynchronous sampling for measurement. Asynchronous sampling uses a recurring event trigger to send signals to the program being profiled. When an event trigger occurs, a signal is sent to the program. A signal handler then records the context where the sample occurred. The recurring nature of the event trigger means that the program counter and context is sampled many times, resulting in a histogram of program contexts. As long as the number of samples collected during execution is sufficiently large (and is not correlated with certain program features), their distribution is expected to approximate the true distribution of the costs that the event triggers are intended to measure.

The second form of statistical sampling is synchronous sampling. Sometimes it is necessary to monitor fundamentally synchronous events such as lock acquisitions. To minimize the overhead typically associated with synchronously monitoring frequently occurring synchronous events, HPCToolkit samples them. In effect, this involves switching between lightweight and heavyweight instrumentation.

Event triggers

Different kinds of event triggers measure different aspects of program performance. From the perspective of a program, event triggers can be either asynchronous or synchronous, corresponding to asynchronous and synchronous sampling, respectively. Asynchronous triggers are external to the monitored program and are not initiated by direct program action. HPCToolkit initiates asynchronous samples using either an interval timer or hardware performance counter events. Hardware performance counters enable HPCToolkit to statistically profile events such as cache misses and issue-stall cycles. Synchronous triggers, on the other hand, are generated via direct program action. Examples of interesting events for synchronous profiling are memory allocation, I/O, and inter-process communication. For such events, one might measure bytes allocated, written, or communicated, respectively. Another example of a synchronous trigger is lightweight instrumentation that samples heavyweight instrumentation.

Unless there is a compelling need for a synchronous event trigger, we prefer an asynchronous one. Asynchronous triggers use easily controllable sampling periods, require no direct change to an application, and, assuming the sampling period is not correlated with program behavior, cannot contribute to a blind spot.

Maintaining control over parallel applications

To manage profiling 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

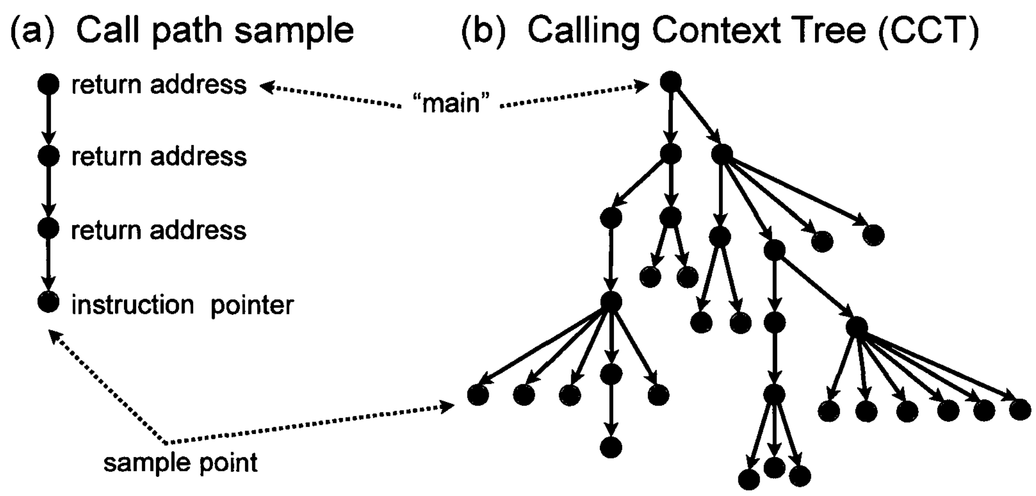


Figure 2.2: An asynchronous-sampling-based call path profiler (a) collects a call path for each sample point; and (b) several call paths form a calling context tree.

a profiling library as an application is launched.[2] For statically linked executables, a script arranges to intercept process control routines at link time.[3] In either case, HPCToolkit is able to execute its own code both before and after the intercepted routine executes.

Call path profiling

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. We refer to the process of monitoring an execution to record the calling contexts in which event triggers fire as *call path profiling* [67].

When synchronous or asynchronous events occur, `hpcrun` records the full calling context for each event. A calling context collected by `hpcrun` is a list of instruction

[2]On Linux, see the loader's special environment variable `LD_PRELOAD`.
[3]On Linux, see the linker's special `--wrap` option.

pointers, one for each procedure frame active at the time the event occurred; an example is shown in Figure 2.2(a). The last 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, we represent all of the call paths for events as a calling context tree (CCT) [9]. In a calling context tree, shown in Figure 2.2(b), 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.

Coping with fully optimized binaries

Collecting a call path profile 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 (as space is reserved for the caller's registers and local variables; as the frame is extended with calls to `alloca`; as 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, we developed a fast, on-the-fly binary analyzer that examines a procedure's machine instructions and computes how to unwind a stack frame for the procedure [141]. For each address in the 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 procedure'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 frame pointer register had in the caller's frame. Once we compute unwind recipes for all intervals in a routine, we memoize them for later reuse.

To apply our binary analysis to compute unwind recipes, we must know where each routine begins 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, we developed a binary analyzer that infers routine boundaries by noting instructions that are reached by call instructions or instructions following unconditional control transfers (jumps and returns) that are not reachable by conditional control flow.

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 the 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 [141].

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.

2.3.2 Attribution

To enable effective analysis, measurements of fully optimized programs must be correlated with important source code abstractions. Since HPCToolkit measures with reference to instructions in executables and shared libraries, for analysis it is necessary to attribute these low-level measurements back to program source. To do this, we need a mapping between a load module's object code and its associated source code. Most load modules contain such mappings in the form of a 'line map.' However, to accurately attribute measurements to interesting source-level structure such as loop nests, it is necessary to have much richer information than can typically be obtained from the line map, which is fundamentally line based. Moreover, the line map for fully optimized programs often contains ambiguities resulting from inlining. Consequently, HPCToolkit's `hpcstruct` tool constructs such a mapping using a binary analysis technique that we call *recovering program structure.*

`hpcstruct` focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, `hpc-struct` (1) parses a load module's machine instructions; (2) reconstructs a control flow graph; and (3) 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 [141].[4]

Several benefits naturally accrue from this approach. First, HPCToolkit can expose the structure of what is actually executed and assign metrics to it, *even if source code is unavailable.* For example, `hpcstruct`'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. `hpc-run`'s function discovery heuristics expose distinct logical procedures within stripped binaries.

[4]Without line map information, `hpcstruct` can still identify procedures and loops, but is not able to account for inlining, which can affect loops in the vicinity of inlined code.

2.3.3 Analysis

Derived metrics

Identifying performance problems and opportunities for tuning may require synthesizing performance metrics from others. To identify where an algorithm is not effectively using hardware resources, one should compute a metric that reflects *inefficiency* rather than accomplishment; *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 floating-point performance is low. Knowing where the most cycles are spent doing things other than floating-point computation hints at opportunities for tuning. Such a metric can be directly computed by taking the difference between the cycle count on one hand and, on the other hand, the floating point operations (FLOPs) count divided by a target FLOPs-per-cycle value, and displaying this measure for loops and procedures. Our experiences with using multiple computed metrics such as miss ratios, instruction balance, and 'lost cycles' underscore the power of this approach.

Third-party metrics

For multithreaded applications, critical inefficiency occurs when threads idle waiting for work. In contrast to serial code, idleness in one thread is usually caused by another thread. For example, if one thread holds a lock that another thread needs, the latter's execution must be delayed. Or, if threads who are responsible for generating parallel work fail to do so, then other threads will be starved of work. To attribute the idleness in one thread to its cause in another thread, we have developed techniques for efficiently blaming the offending thread for the idleness it causes [140, 144]. We call these metrics *third-party* because in contrast to *first-party* metrics, they require some knowledge of the execution state of other threads and the interactions between those threads.

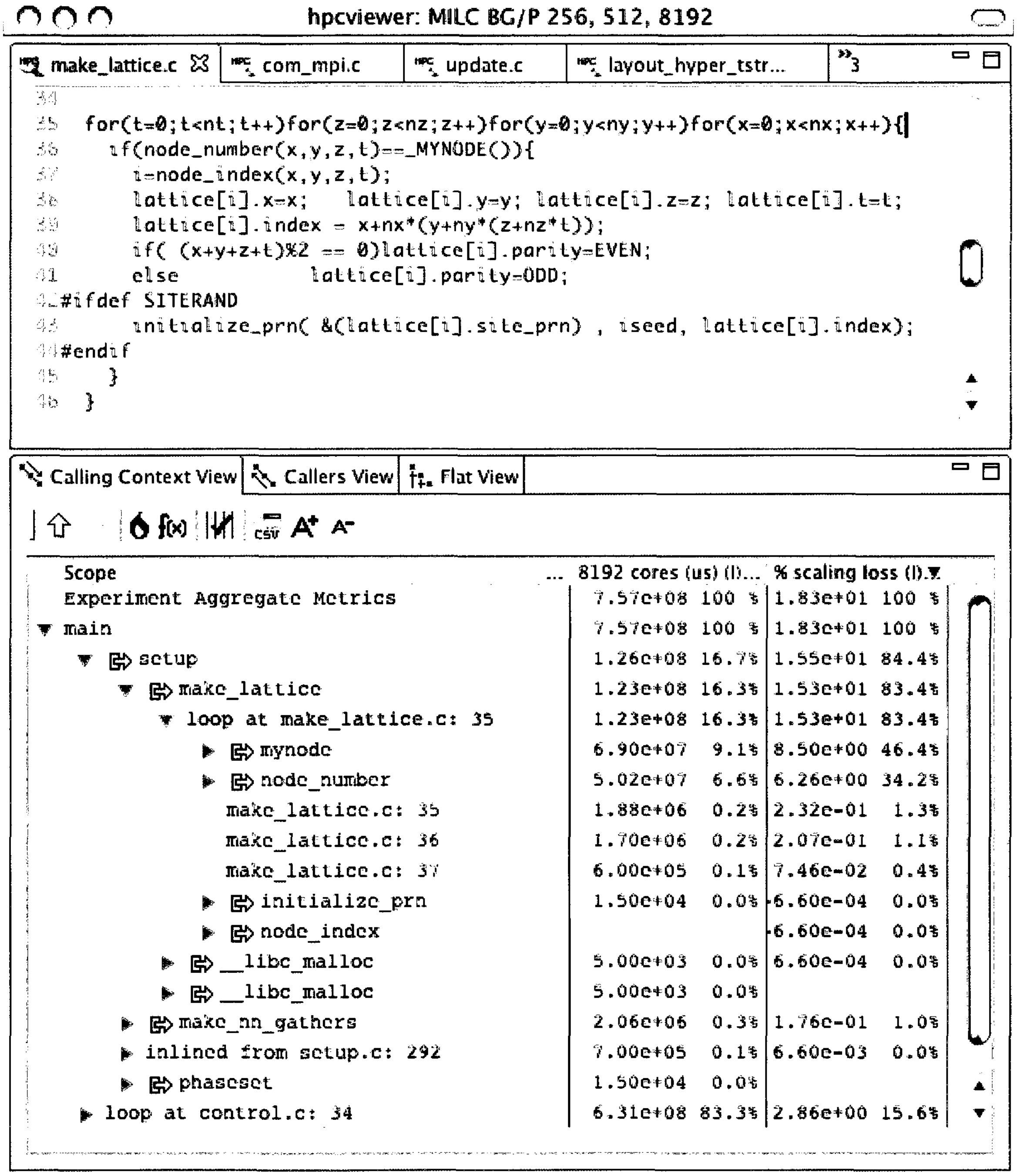
Scalably identifying scalability bottlenecks in parallel programs

We have developed scalable versions of `hpcprof` and `hpcviewer` for scalably analyzing, attributing and presenting call path profiles from large-scale executions. One novel application of HPCToolkit's call path profiles is to use them to pinpoint and quantify scalability bottlenecks in emerging petascale SPMD parallel executions [143]. In particular, with HPCToolkit's scalable analysis and presentation, it is possible to apply *differential profiling* [41, 92] to compare two whole executions instead of, as with non-scalable techniques, two 'representative' threads. Combining execution-wide call path profiles with program structure information, HPCToolkit can use an *excess work* metric to quantify scalability losses and attribute them to the full calling context in which these losses occur.

We have also developed techniques for effectively analyzing scalability bottlenecks in multithreaded applications [140, 144]. Using them, HPCToolkit can attribute precise measures of lock contention, parallel idleness, and parallel overhead to *source-level* calling contexts — even for a multithreaded language such as Cilk [58], which uses a work-stealing scheduler.

2.3.4 Presentation

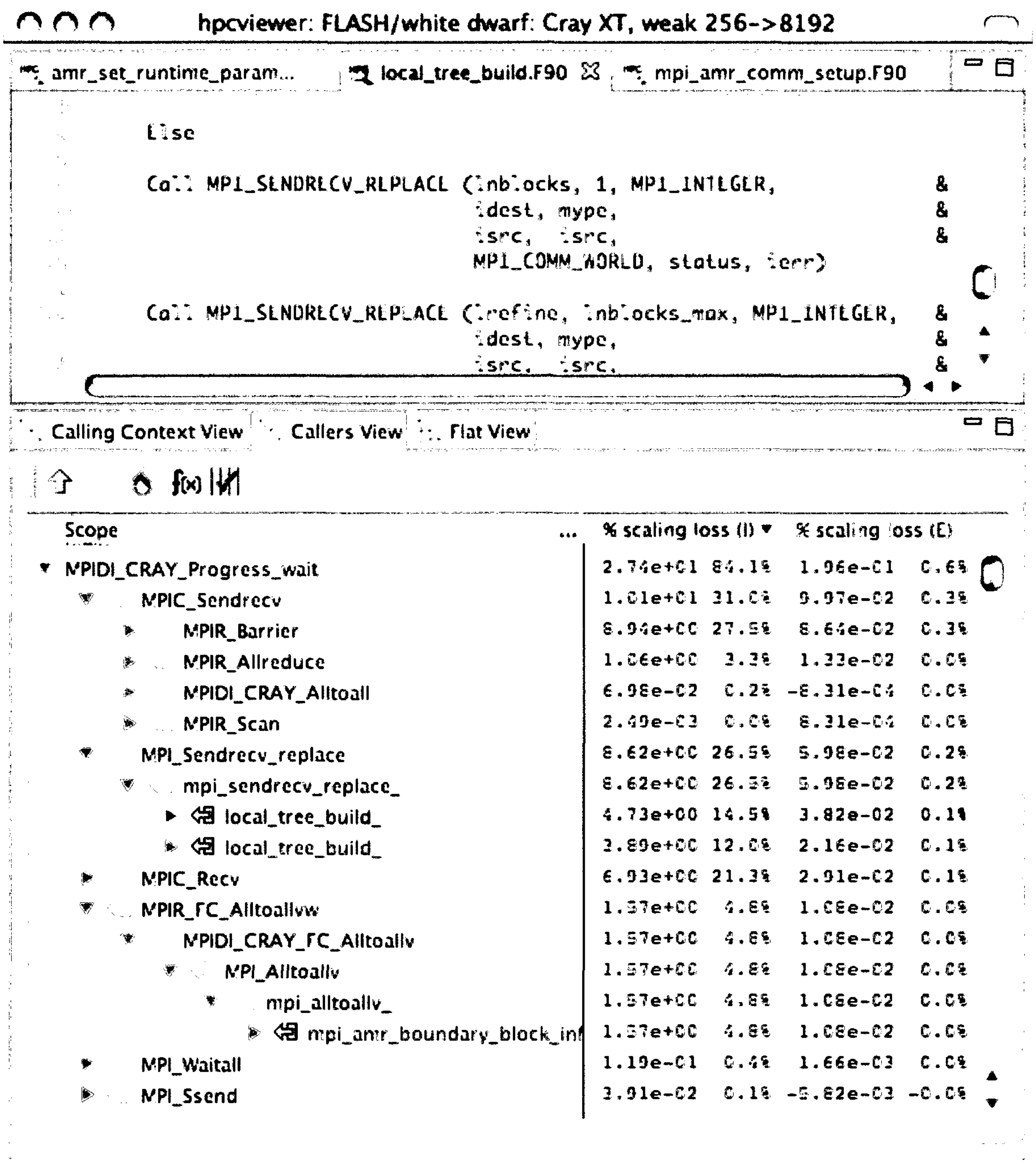
HPCToolkit's presentation tool, `hpcviewer`, interactively presents context-sensitive performance metrics correlated to program structure (Section 2.3.2) and mapped to a program's source code, if available. Figure 2.3 shows a snapshot of `hpcviewer`'s user interface presenting a call path profile. The user interface is composed of two principal panes. The top pane displays program source code. The bot-



```
34
35  for(t=0;t<nt;t++)for(z=0;z<nz;z++)for(y=0;y<ny;y++)for(x=0;x<nx;x++){
36    if(node_number(x,y,z,t)==_MYNODE()){
37      i=node_index(x,y,z,t);
38      lattice[i].x=x;   lattice[i].y=y; lattice[i].z=z; lattice[i].t=t;
39      lattice[i].index = x+nx*(y+ny*(z+nz*t));
40      if( (x+y+z+t)%2 == 0)lattice[i].parity=EVEN;
41      else          lattice[i].parity=ODD;
42 #ifdef SITERAND
43      initialize_prn( &(lattice[i].site_prn) , iseed, lattice[i].index);
44 #endif
45    }
46  }
```

```
Scope                                       ... 8192 cores (us) (I)... % scaling loss (I)
Experiment Aggregate Metrics                    7.57e+08 100 %  1.83e+01 100 %
▼ main                                          7.57e+08 100 %  1.83e+01 100 %
   ▼ setup                                      1.26e+08 16.7%  1.55e+01 84.4%
      ▼ make_lattice                            1.23e+08 16.3%  1.53e+01 83.4%
         ▼ loop at make_lattice.c: 35           1.23e+08 16.3%  1.53e+01 83.4%
            ▶ mynode                            6.90e+07  9.1%  8.50e+00 46.4%
            ▶ node_number                       5.02e+07  6.6%  6.26e+00 34.2%
              make_lattice.c: 35                1.88e+06  0.2%  2.32e-01  1.3%
              make_lattice.c: 36                1.70e+06  0.2%  2.07e-01  1.1%
              make_lattice.c: 37                6.00e+05  0.1%  7.46e-02  0.4%
            ▶ initialize_prn                    1.50e+04  0.0% -6.60e-04  0.0%
            ▶ node_index                                       -6.60e-04  0.0%
         ▶ __libc_malloc                        5.00e+03  0.0%  6.60e-04  0.0%
         ▶ __libc_malloc                        5.00e+03  0.0%
      ▶ make_nn_gathers                         2.06e+06  0.3%  1.76e-01  1.0%
      ▶ inlined from setup.c: 292               7.00e+05  0.1%  6.60e-03  0.0%
      ▶ phaseset                                1.50e+04  0.0%
   ▶ loop at control.c: 34                      6.31e+08 83.3%  2.86e+00 15.6%
```

Figure 2.3: hpcviewer's Calling Context view of scaling losses (cycles).



hpcviewer: FLASH/white dwarf: Cray XT, weak 256->8192

amr_set_runtime_param... | local_tree_build.F90 | mpi_amr_comm_setup.F90

```
        Else

        Call MPI_SENDRECV_REPLACE (lnblocks, 1, MPI_INTEGER,            &
                                   idest, mype,                          &
                                   isrc,  isrc,                          &
                                   MPI_COMM_WORLD, status, ierr)

        Call MPI_SENDRECV_REPLACE (lrefine, lnblocks_max, MPI_INTEGER,  &
                                   idest, mype,                          &
                                   isrc,  isrc,                          &
```

Calling Context View | Callers View | Flat View

```
Scope                                       ...  % scaling loss (I) ▼  % scaling loss (E)
▼ MPIDI_CRAY_Progress_wait                       2.74e+01 84.1%   1.96e-01   0.6%
    ▼ MPIC_Sendrecv                              1.01e+01 31.0%   9.97e-02   0.3%
        ▶ MPIR_Barrier                           8.94e+00 27.5%   8.64e-02   0.3%
        ▶ MPIR_Allreduce                         1.06e+00  3.3%   1.33e-02   0.0%
        ▶ MPIDI_CRAY_Alltoall                    6.98e-02  0.2%  -8.31e-04   0.0%
        ▶ MPIR_Scan                              2.49e-03  0.0%   8.31e-04   0.0%
    ▼ MPI_Sendrecv_replace                       8.62e+00 26.5%   5.98e-02   0.2%
        ▼ mpi_sendrecv_replace_                  8.62e+00 26.5%   5.98e-02   0.2%
            ▶ local_tree_build_                  4.73e+00 14.5%   3.82e-02   0.1%
            ▶ local_tree_build_                  3.89e+00 12.0%   2.16e-02   0.1%
    ▶ MPIC_Recv                                  6.93e+00 21.3%   2.91e-02   0.1%
    ▼ MPIR_FC_Alltoallvw                         1.57e+00  4.8%   1.08e-02   0.0%
        ▼ MPIDI_CRAY_FC_Alltoallv                1.57e+00  4.8%   1.08e-02   0.0%
            ▼ MPI_Alltoallv                      1.57e+00  4.8%   1.08e-02   0.0%
                ▼ mpi_alltoallv_                 1.57e+00  4.8%   1.08e-02   0.0%
                    ▶ mpi_amr_boundary_block_inf 1.57e+00  4.8%   1.08e-02   0.0%
    ▶ MPI_Waitall                                1.19e-01  0.4%   1.66e-03   0.0%
    ▶ MPI_Ssend                                  3.91e-02  0.1%  -5.82e-03  -0.0%
```

Figure 2.4: hpcviewer's Callers view of scaling losses (cycles).

tom pane associates a table of performance metrics with static or dynamic program structure. `hpcviewer` provides three different views of performance measurements collected using call path profiling. We briefly describe the three views and their corresponding purposes.

- **Calling Context view**. Figure 2.3 shows a Calling Context view. This top-down view associates metrics with the full calling context in which they were incurred. Indentation in the lower pane shows dynamic nesting of calls, loops and inlined code. Using this view, one can readily see how much of the application's cost was incurred by a procedure 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 context precisely by individual call sites; this means that if a procedure f contains calls to procedure g in different places, each call represents a separate calling context. The Calling Context view is created by integrating dynamic calling contexts gathered by `hpcrun` with static program structure (e.g., loops) gathered by `hpcstruct`. Loops appear explicitly in the call chains shown in Figure 2.3.

- **Callers view**. This bottom-up view enables one to look upward along call paths. Because the Callers view apportions metrics of a callee on behalf of its caller, 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 `MPI_Wait` in many different calling contexts. The cost of any particular call will depend upon its context. Serialization or load imbalance may cause long waits in some calling contexts but not others. Figure 2.4 shows `hpcviewer` presenting a Callers view of a call path profile.

When several levels of the Callers view are expanded, saying that the Callers view apportions metrics of a callee on behalf of its caller can be ambiguous: what is the caller and what is the callee? To resolve this ambiguity we can say that the Callers view apportions the metrics of a particular procedure *in its various calling contexts* on behalf of that context's caller. Alternatively but equivalently, the Callers view apportions the metrics of a particular procedure on behalf of its various *calling contexts*. For example, notice that the highlighted line in Figure 2.4 shows a (partially collapsed) callers chain ending with `local_tree_build` that is four levels deep. The metrics at `local_tree_build` are actually formed by attributing the metrics at the chain's root (`MPIDI_CRAY_Progress_wait`) up its call chain to `local_tree_build`.

- **Flat view**. This view organizes performance data according to an application's static structure. That is, all costs incurred by a procedure, in any calling context, are aggregated together to form the Flat view. This view 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 in these three views, `hpcviewer` computes both *inclusive* and *exclusive* metric values. For the moment, consider the Calling Context view. Inclusive metrics reflect costs for the entire subtree rooted at that scope. Exclusive metrics are of two flavors, depending on the scope. For a procedure, exclusive metrics reflect all costs within that procedure but excluding callees. In other words, for a procedure, costs are exclusive with respect to dynamic call chains. For all other scopes, exclusive metrics reflect costs for the scope itself; i.e., costs are exclusive with respect to static structure. The Callers and Flat views contain inclusive and exclusive metric values that are relative to the Calling Context view. This means, e.g., that inclusive metrics for a particular scope in the Callers or Flat view are with respect to that scope's subtree in the Calling Context view.

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* one or more levels of the static hierarchy, e.g., to facilitate comparison of costs between loops in different procedures; and *zooming* to focus on a particular scope and its children.

2.4 Related Work

Here, we primarily discuss work related to HPCToolkit's measurement methodology. We defer detailed discussion of attribution, analysis and presentation to later chapters.

Tools that permit monitoring of unmodified executables are critical for applications with long build processes or for attaching to an existing production run. Although different performance tools measure the same dimensions of an execution, they may differ with respect to their measurement methodology. These different methodologies determine whether a tool can analyze the performance of unmodified applications. TAU [129], OPARI [102], and Pablo [117], among others, add instrumentation to source code during a program's build process. Gprof relies on compiler-inserted instrumentation [66]. Model-dependent strategies often use instrumented libraries [34, 62, 101, 126, 148]. Intel's VTune [77], Cray's CrayPAT [48], and IBM's HPC Toolkit [74] statically instrument an application's binary. None of these strategies support performance analysis of unmodified applications. To work with unmodified application binaries, tools have taken two approaches. Some tools use dynamic binary instrumentation [29,49,99] or library preloading [44,60,107,127,130] (a special, less flexible, form of dynamic binary instrumentation). Other tools use asynchronous sampling [7, 13, 78, 85, 106, 127]. HPCToolkit's call path profiler uniquely combines preloading (to monitor unmodified dynamically linked binaries), asynchronous sampling (to control overhead), and binary analysis (to help handle unruly object code) for measurement. In addition, our call path measurement has novel aspects that make it more accurate and impose lower overhead than other call graph or call path profilers (see Chapter 3).

These different measurement approaches also fundamentally affect a tool's potential for accurate and precise measurements. Source code instrumentation cannot measure binary-only library code, may affect compiler transformations, and incurs large overheads. Binary instrumentation may have blind spots and incur large overheads. For example, Intel's widely used VTune [77] call path profiler employs binary instrumentation that fails to measure functions in stripped object code and imposes enough overhead that Intel explicitly discourages program-wide measurement. When measuring at a fine granularity, dynamic binary instrumentation suffers from overhead. HPCToolkit's call path profiler uses asynchronous sampling to obtain both accurate and precise measurements. Moreover, no other tool combines asynchronous sampling with post-mortem binary analysis to attribute those measurements back to source-level program structure, including loops and inlined procedures. Other tools [7, 13, 106] use post-mortem analyses to detect loops, but only at the binary level.

An alternative to asynchronous sampling is to (synchronously) sample instrumentation itself. The basic idea is to use extremely lightweight instrumentation to periodically employ heavyweight instrumentation. This technique can be used at either the source or binary level. When carefully applied, sampling instrumentation can be quite effective at reducing the overhead of gathering selective performance data [99] or (intraprocedural) path or edge profiles [17, 147, 156]. Applying this technique to call path profiles is also effective relative to heavyweight instrumentation. For instance, Zhuang et al. report 20% overhead as opposed to hundreds of percents [158]. Hirzel and Chilimbi collect both contextual (call path) and flow (path) information for 3-18%, though at the expense of extensive code duplication [71].

The basic difficulty with sampling instrumentation is that for small frequently executed routines, the lightweight instrumentation itself is executed frequently. In a few cases, the synchronous nature of instrumentation may be needed, such as when it is necessary to intercept every instance of a lock routine. We have successfully used lightweight instrumentation in such a specialized case [144]. However, in most cases, this is unnecessary. Even the DTrace [33] tool, which is based on extremely lightweight dynamic binary instrumentation, supports asynchronous sampling and stack unwinding for collecting profiles.

In some cases, it may be possible to overcome the problems of sampling instrumentation by sophisticated placement of instrumentation and individual control of instrumentation points [21]. However, we postulate that, when measurements do not naturally require synchroneity, asynchronous sampling is preferable. To effectively control overhead, sampling instrumentation relies on careful placement so that measurement-related code is not executed too frequently. This selective placement can result in blind spots that would not exist with asynchronous sampling. Our approach of using asynchronous statistical sampling to obtain call paths via stack unwinding, enriching those paths with static program structure and correlating the result to source code naturally avoids these problems while achieving a low level of overhead (usually 1–2%). Such accuracy and precision is difficult to replicate using instrumentation. Although we do not gather all details of intraprocedural flow, we highlight loops, which usually are critical to performance. Moreover, PMU-based sampling gives rich information about resource consumption and inefficiency — information that would at best be difficult to obtain for similarly low overhead using instrumentation-based measurement.

Tools for measuring parallel application performance are typically model dependent, such as libraries for monitoring MPI communication (e.g., [148, 149, 153]), interfaces for monitoring OpenMP programs (e.g., [34, 102]), or global address space languages (e.g., [137]). In contrast, HPCToolkit can pinpoint contextual performance problems independent of model — and even within stripped, vendor-supplied math and communication libraries [41].

To our knowledge, Appendix A presents the first formal analysis of statistical sampling as a means of obtaining a thread-based profile or trace. Although other profilers are based on statistical sampling, we are not aware of any formal attempts at analyzing their error and accuracy; cf. [7, 10, 13, 25, 48, 65, 78, 81, 85, 106, 127, 157]. Sastry et al. use systematic sampling to implement a lossy hardware compressor designed to support flat profiling [124]. Their analysis partially overlaps with ours, as they make an observation similar to our Equation A.9. However, the rest of their analysis depends on using a simulator to compare an ideal flat profile with flat profiles obtained from various compressor designs. Azimi et al. analyze the accuracy of using sampling to multiplex hardware performance counters [18]. Because these authors are only interested in program-wide totals, they compare probability distributions derived from absolute counts and from multiplexing. In contrast, we are interested in procedures, loops and statements in their full calling context. Maxwell et al. assess the accuracy of a given performance counter by comparing the results of analytical models, simulations and experiments for microbenchmarks [91]. While each of these analyses is useful in its context, none of them provides a formal analysis to address all the questions we do.

Although other work has grounded itself on a formal statistical analysis, its focus has been to use sampling as a mechanism for monitoring only a small subset of cluster nodes or application processes for a large-scale system. For instance, Mendes and Reed use simple random sampling of node characteristics to estimate system-wide attributes such as "the fraction of available nodes" on large-scale clusters [97]. Gamblin, Fowler and Reed use sampling of processes within a parallel program to drastically reduce data volume of tracing [64]. They describe an application-wide 'daemon' that uses adaptive stratified sampling to periodically select processes within each of the application's process groups. The daemon then instructs each selected and unselected process to enable and disable tracing, respectively. Although an unselected process may continue executing tracing instrumentation, it does not continue generating trace records that must be handled by a limited-bandwidth I/O subsystem. Thus, while both our work and this work use sampling to obtain measurements that grow sub-linearly with the population size, this work is different in two respects. First, whereas we use independent per-thread sample sources, this work uses one application-wide sample source. Second, by sampling processes, Gamblin et al. sample at a coarser level of granularity than our work, which samples contexts within a thread.

2.5 Discussion

We have described a unique methodology for analyzing the performance of an application's execution under the subheadings measurement, attribution, analysis and presentation. This methodology is unique in three important ways.

First, our techniques are based on accurate and precise measurement. If measurement includes systematic error, insightful presentation would be misleading and therefore useless. By pairing sampling-based profiling with binary analysis to aid both measurement and attribution, HPCToolkit achieves both highly accurate and precise measurements.

Second, our methodology is capable of obtaining unique and actionable insight into the performance of parallel programs. To obtain such insight, it is necessary to precisely identify where applications execute inefficiently. Moreover, poor presentation of excellent data obscures and hinders insight. HPCToolkit combines (1) accurate and precise thread-level measurements; (2) novel analyses for pinpointing and quantifying parallel inefficiency and scalability bottlenecks in parallel programs; and (3) data presentation using three complementary views to facilitate rapid top-down analysis.

Third, our methodology is comprehensive and capable of identifying performance issues in real large-scale parallel applications. `hpcrun` samples the whole calling context of an unmodified fully optimized parallel programs irrespective of whether the call chain passes through communication libraries or process launchers. `hpcstruct` recovers the source code structure for any portion of the calling context regardless of source code (as long as line map information is present). HPCToolkit's use of binary analysis to support both measurement (call stack unwinding of unmodified optimized code) and attribution to loops and inlined functions has enabled its use on today's grand challenge applications — multi-lingual programs that leverage third-party libraries for which source code and symbol information may not be available. `hpcprof` scalably attributes measurements to source code and summarizes thread-level performance metrics for large-scale executions. `hpcviewer` scalably presents the contextual measurements in three complementary views to enable top-down analysis. In sum, HPCToolkit can measure *what actually executes* and present it in an effective way that exposes details, but within the context of larger abstractions.

Our work has emphasized obtaining actionable insight. Such information is foundational for feedback-directed optimization, automated performance tuning, and for validating performance models. In the future, we are interested in transforming this *descriptive* information into targeted list of *prescriptive* recommendations for resolving performance bottlenecks.

Chapter 3

Measurement & Attribution: Fully Optimized Applications

3.1 Introduction

Modern programs frequently employ sophisticated modular designs that exploit object-oriented abstractions and generics. Composition of C++ algorithm and data structure templates typically yields loop nests spread across multiple levels of routines. To improve the performance of such codes, compilers inline routines and optimize loops. However, careful hand-tuning is often necessary to obtain top performance. To support tuning of such code, performance analysis tools must pinpoint context-sensitive inefficiencies in fully optimized applications.

Several contemporary performance tools measure and attribute execution costs to calling context in some form [7, 13, 48, 65, 77, 78, 85, 127, 129]. However, when applied to fully optimized applications, existing tools fall short for two reasons. First, current calling context measurement techniques are unacceptable because they either significantly perturb program optimization and execution with instrumentation (e.g., [7, 48, 65, 77, 129]), or rely on compiler-based information that is sometimes inaccurate or unavailable, which causes failures while gathering certain calling contexts (e.g., [7, 13, 78, 85, 127]). Second, by inlining procedures and transforming loops, optimizing compilers introduce a significant semantic gap between the binary and source code. Thus, prior strategies for attributing context-sensitive performance at the source level either compromise measurement accuracy or remain too close to the object code.

To clarify these issues, we consider the capabilities of some popular tools using three related categories: calling context representation, measurement technique and attribution technique.

Calling Context Representation

Performance tools typically attribute performance metrics to calling context using a call graph or call path profile. Two widely-used tools that collect call graph profiles are Gprof [65] and Intel's VTune [77]. A *call graph profile* consists of a node for each procedure and a set of directed edges between nodes. An edge exists from node p to node q if p calls q. To represent performance measurements, edges and nodes are weighted with metrics. Call graph profiles are often insufficient for modular applications because a procedure p that appears on multiple distinct paths is represented with one node, resulting in shared paths and cycles. Consequently, with a call graph profile it is in general not possible to assign costs to p's full calling context, or even to long portions of it. To remove this imprecision, a *call path profile* [67] represents the full calling context of p as the path of calls from the program's entry point to p. Call path profiling is necessary to fully understand the performance of modular codes.

Measurement Technique

There are two basic approaches for obtaining calling context profiles: instrumentation and asynchronous sampling. Instrumentation-based tools use one of four principal instrumentation techniques. Tools such as TAU [129] use *source code instrumentation* to insert special profiling code into the source program before compilation. The well-known Gprof [65] relies on *compiler-inserted instrumentation.* Intel's VTune [77] uses *static binary instrumentation* to augment application binaries with profiling code. The fourth technique is *dynamic binary instrumentation*, which is used by Pin [88] and DynInst [29].

While source-level instrumentors collect measurements that are easily mapped to source code, their instrumentation can interfere with compiler optimizations such as inlining and loop transformations. As a result, measurement approaches based on source-level instrumentation may not accurately reflect the performance of fully optimized code [139]. Compiler-inserted instrumentation may also compromise optimization. For example, in some compilers Gprof-instrumented code cannot be fully optimized.

An important problem with source, compiler-inserted and static binary instrumentation is that they require recompilation or binary rewriting of a program and all its libraries. This requirement poses a significant inconvenience for large, complex applications. More critically, the need to see the whole program before run time can lead to 'blind spots,' i.e., portions of the execution that are systematically excluded from measurement. For instance, source instrumentation fails to measure any portion of an application for which source code is unavailable; this frequently includes critical system, math and communication libraries. For Fortran programs, this approach can also fail to associate costs with intrinsic functions or compiler-inserted array copies. Static binary instrumentation is unable to cope with shared libraries that are dynamically loaded during execution.

The fourth approach, *dynamic binary instrumentation*, supports fully optimized binaries and avoids blind spots by inserting instrumentation in the executing application [29]. Intel's recently-released Performance Tuning Utility (PTU) [7], includes a call graph profiler that adopts this approach by using Pin [88]. However, dynamic instrumentation remains susceptible to systematic measurement error because of instrumentation overhead.

Indeed, all four instrumentation approaches suffer in two distinct ways from overhead. First, instrumentation dilates total execution time, sometimes enough to preclude analysis of large production runs or force users to *a priori* introduce blind spots via selective instrumentation. For example, because of an average slowdown factor of 8, VTune *requires* users to limit measurement to so-called 'modules of interest' [77]. Moreover, overhead is even more acute if loops are instrumented. A recent Pin-based 'loop profiler' incurred an average slowdown factor of 22 [106]. Second, instrumentation dilates the total measured cost of each procedure, disproportionately inflating costs attributed to small procedures and thereby introducing a systematic measurement error.

The alternative to instrumentation is *asynchronous sampling.* Since sampling periods can easily be adjusted (even dynamically), this approach naturally permits low, controllable overhead. Sampling-based call path profilers, such as the one with Intel's PTU [7], use call stack unwinding to gather calling contexts. Stack unwinding requires either the presence of frame pointers or correct and complete unwind information for *every* point in an executable because an asynchronous sample event may occur anywhere. However, fully optimized code often omits frame pointers. Moreover, unwind information is often incomplete (for epilogues), missing (for hand-coded assembly or partially stripped libraries) or simply erroneous (optimizers often fail to update unwind information as they transform the code). In particular, optimized math and communication libraries frequently apply every 'trick in the book' to critical procedures (e.g., hot-cold path splitting [43]) — just those procedures that are likely to be near the innermost frame of an unwind.

Attribution Technique

By inlining procedures and transforming loops, optimizing compilers introduce a semantic gap between the object and source code, making it difficult to reconcile binary-level measurements with source-level constructs. Compiler transformations such as inlining and tail call optimization cause call paths during execution to differ from source-level call paths. After compilers inline procedures and apply loop transformations, execution-level performance data does not correlate well with source code. Since application developers wish to understand performance at the source code level, it is necessary for tools to collect measurements on fully optimized binaries and then translate those measurements into source-level insight. Since loops are critical to performance, but are often dynamically nested across procedure calls, it is important to understand loops in their calling context.

Much prior work on loop attribution either compromises measurement accuracy by relying on instrumentation [106, 129] or is based on context-less measurement [93]. A few sampling-based call path profilers [7,13,106] identify loops, but at the binary level. Moseley et al. [106] describe a sampling-based profiler (relying on unwind information) that additionally constructs a dynamic loop/call graph by placing loops within a call graph. However, by not accounting for loop or procedure transformations, this tool attributes performance only to binary-level loops and procedures. Also, by using a dynamic loop/call *graph*, it is not possible to understand the performance of procedures and loops in their full calling context.

Our Approach

To understand the performance of modular programs, we built HPCToolkit's `hpcrun`, `hpcstruct` and `hpcprof`. `hpcrun` is a call path profiler that measures and attributes execution costs of unmodified, fully optimized executables to their full calling context — and with the help of hpcstruct and hpcprof — also attributes costs to loops and inlined code. Achieving this result required novel solutions to three problems:

- To measure dynamic calling contexts, we developed a context-free on-line binary analysis for locating procedure bounds and computing unwind information. We show its effectiveness on x86-64 applications in the SPEC CPU2006 suite compiled with Intel, Portland Group and PathScale compilers using peak optimization.[1]

- To attribute performance to source-level source code, we developed a novel post-mortem analysis of the optimized object code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code. The ability to expose inlined code and its relation to source-level loop nests without a special-purpose compiler and without any additional measurement overhead is unique.

- To compellingly present performance data, we combine (post-mortem) the recovered static program structure with dynamic call paths to expose inlined frames and loop nests. No other sampling-based tool attributes the performance of *transformed loops* in the *full calling context* of *transformed routines* for *fully optimized binaries* to source code.

In this chapter, we describe our solutions to these problems. The major benefit of our approach is that hpcrun is minimally invasive, yet accurately attributes performance to both static and dynamic context, providing unique insight into program performance.

[1]The Acknowledgments section recognizes the contributions of collaborators.

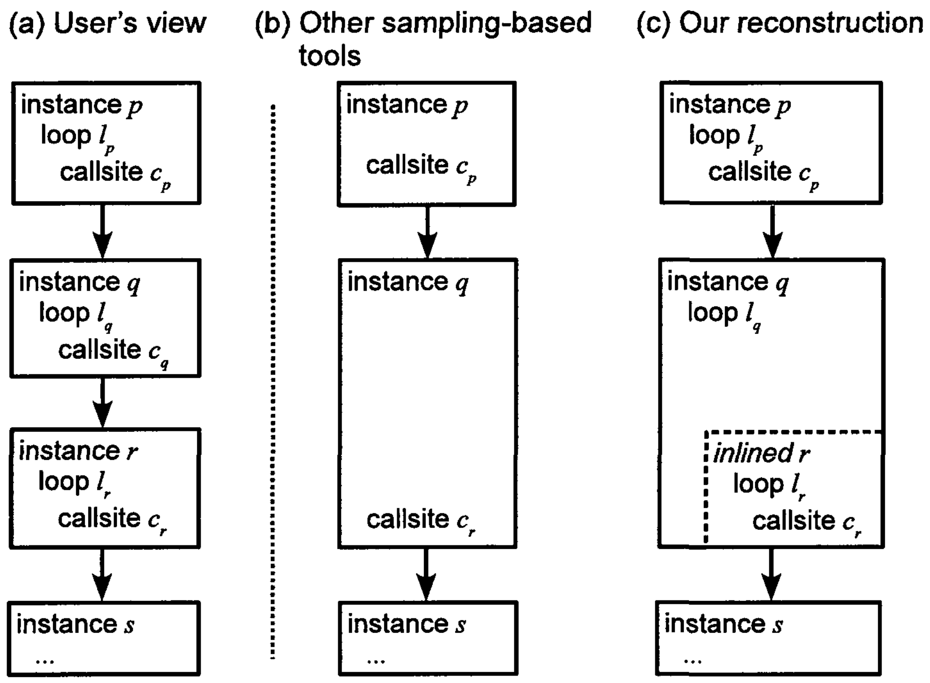


Figure 3.1: Attributing call path metrics to source code. If a compiler inlines call site c_q, current attribution techniques (b) produce confusing results. Our techniques (c) expose both loops and inlined frames by correlating call paths with program structure.

Our results are summarized by Figure 3.1. As shown in Figure 3.1a, let $p \rightarrow q \rightarrow r \rightarrow s$ be a source-level call chain of four procedures. Procedure p contains a call site c_p (that calls q) embedded in loop l_p; procedures q and r contain analogous call sites. Assume that a compiler inlines call site c_q so that code for procedure r appears within loop l_q. Consequently, at run time c_q is not executed and therefore a procedure frame for r is absent. Using call stack unwinding and line map information recorded by compilers yields the reconstruction of context shown in Figure 3.1b. By combining dynamic context obtained by call stack unwinding with static information about inlined code and loops gleaned using binary analysis, our tools obtain the reconstruction shown in Figure 3.1c. Specifically, our tools (1) identify that c_p and c_r are located within loops; (2) detect the inlining; and (3) nest c_r within both its original procedure context r and its new host procedure q. Most importantly, reconstructed procedures, loops and inlined frames can be treated as 'first-class' entities for the purpose of assigning performance metrics.

The rest of the chapter is as follows. Section 3.2 describes our use of binary analysis to support call path profiling of optimized code and evaluates its effectiveness. Section 3.3 describes our binary analysis to support accurate correlation of performance measurements to optimized code. Section 3.4 highlights the rich performance data we obtain by fusing dynamic call paths and static structure. Finally, Section 3.5 discusses related work; and Section 3.6 discusses the chapter's high-level themes.

3.2 Binary Analysis for Call Path Profiling

Call path profilers based on asynchronous sampling use call stack unwinding to gather calling contexts. For such profilers to be accurate, they must be able to unwind the call stack at *any* point in a program's execution. A stack unwind, which forms the calling context for a sample point, is represented by the program counter for the innermost procedure frame and a list of return addresses — one for each of the other active procedure frames. Successfully unwinding the call stack requires determining the return address for each frame and moving up the call chain to the frame's parent. Obtaining the return address for a procedure frame without a frame pointer is non-trivial since the procedure's frame can dynamically grow (as space is reserved for the caller's registers and local variables, or supplemented with calls to `alloca`) and shrink (as space for the aforementioned purposes is deallocated) as the procedure executes. If the return address is kept in the stack (as is typical for non-leaf procedures), the offset from the stack pointer at which the return address may be obtained often changes as a procedure executes.

Finding the return address for a procedure frame is simple with correct and complete compiler-generated unwind information [61]. Unfortunately, compilers routinely omit unwind information for procedure epilogues because it is not needed for exception handling. However, even if compilers generate complete unwind information, fully optimized applications often link with vendor libraries (e.g., math or OpenMP) that have incomplete unwind tables due to hand-coded assembly or partial stripping. Since codes may spend a significant fraction of time in procedures that lack proper unwind information,[2] dropping or mis-attributing samples that occur in such procedures could produce serious measurement error.

To enable accurate unwinding of *all* code, even code lacking compiler-based unwind information, we developed two binary analyzers — one to determine where a procedure begins and ends in partially stripped code, and a second to compute how to unwind to a caller's frame from any address within a procedure. At any instant, a frame's return address (which also serves as the program counter for the calling frame) may be located either (1) in a register, (2) in a location relative to the stack pointer, or (3) in a location relative to the frame pointer (which the frame must have initialized before using). The value of the frame pointer for a caller's frame may be found similarly. To recover the program counter, stack pointer and frame pointer values for a caller's frame, we compute a sequence of *unwind recipes* for a procedure. Each unwind recipe corresponds to an interval of code that ends in a *frame-relevant instruction*. A frame-relevant instruction is one that changes the machine state (e.g., by moving the stack pointer, saving the frame pointer value inherited from the caller, or initializing the frame pointer for the current frame) in such a way that a different unwind recipe is needed for instructions that follow.

[2]For example, the S3D turbulent combustion code described in Section 3.4.2 spends nearly 20% of its total execution time in the math library's exponentiation routine as it computes reaction rates.

Although procedure bounds and unwind recipes could be computed off-line, we perform both analyses on demand at run time. We perform binary analysis on each load module to recover the bounds of all of its procedures. This analysis is triggered at program launch for the executable and all shared libraries loaded at launch and whenever a new shared library is loaded with `dlopen`. The computed procedure-bounds information for a module is cached in a table that is queried using binary search. We perform binary analysis to compute unwind intervals for a procedure lazily — the first time that the procedure appears on the call stack when a sample event occurs. This approach elegantly handles dynamically loaded shared libraries and avoids wasting space and time computing unwind recipes for procedures that may never be used. To support fast queries, we memoize unwind recipes in a splay tree [132] indexed by intervals of code addresses. Algorithm 3.1 shows a high-level overview of the process of performing on-the-fly binary analysis to support call path profiling. Because dynamic analysis must be efficient, we prefer fast linear-time heuristics that may occasionally fail over slower fully general methods.[3] (An evaluation of our approach in Section 3.2.3 shows that our methods almost never fail in practice.) In the next two sections, we describe how we infer procedure bounds and compute unwind recipes.

3.2.1 Inferring Procedure Bounds

To compute unwind recipes for a procedure based on its instruction sequence, one must know the procedure's *bounds*, namely where the procedure begins and ends. In many cases, complete information about procedure bounds is not readily available. For instance, stripped shared libraries have only a dynamic symbol table that contains only information about global procedure symbols; all information about local symbols is missing. Often, libraries are partially stripped. For instance, the OpenMP run-time

[3]For example, Rosenblum et al. [122] developed an off-line analyzer to recover procedure bounds in fully stripped code. However, the focus of their work was on thorough analysis for security.

Algorithm 3.1: backtrace: Use on-the-fly binary analysis to unwind call stacks from fully optimized code.

```
Input: $\mathcal{B}$, procedure bounds for each load module
Input: $\mathcal{U}$, unwind recipes for procedure intervals (splay tree)
1  let $\mathcal{F} = \langle PC, FP, SP \rangle$ be the frame of the sample point (consisting of program
       counter, frame and stack pointer)
2  while $\mathcal{F}$ is not the outermost frame do
3      if $\mathcal{U}$ has no unwind recipe for $PC$ then
4          let $\mu$ be the load module containing $PC$
5          if $\mathcal{B}$ has no bounds for $\mu$ then
6              Compute bounds for all procedures in $\mu$
7          let $\pi$ be the procedure (from $\mathcal{B}$) with bounds $\beta$ containing $PC$
8          Scan the object code of $\pi$, (1) tracking the locations of its caller's program
              counter, frame and stack pointer; and (2) creating an unwind recipe for
              each distinct interval
9      let $\upsilon$ be the unwind recipe (from $\mathcal{U}$) for $PC$
10     let $\mathcal{F}' = \langle PC', FP', SP' \rangle$ be the caller's frame, computed using $\upsilon$
11     $\mathcal{F} \Leftarrow \mathcal{F}'$
```

library for version 3.1 of PathScale's x86-64 compiler only has symbol information for OpenMP API procedures; all information about other procedures is missing. For this reason, inferring procedure bounds for stripped or partially stripped code is an important precursor to computing unwind intervals.

Our approach for inferring procedure bounds is based on the following observations.

- *We expect each load module to provide information about at least some procedure entry points.*

  Performance analysis of a stripped executable is typically unproductive. Interpreting measurement results is difficult without procedure names. For this reason, entry points for user procedures will generally be available for an executable. Dynamically linked shared libraries have (at a minimum) procedure entry points for externally visible library procedures.

- *We must perform procedure discovery on all load modules.*

Partially stripped libraries are not uncommon. There is no *a priori* way to distinguish between a partially stripped load module and one that has full symbol information. We have also encountered (non-stripped) executables that lack information about some procedures. For instance, the SPEC benchmark 483.xalancbmk, when compiled with the PathScale C++ compiler (version 3.1, using `-O3`) contains small anonymous procedures.

- *Having the proper address for a procedure start is more important than having the proper address for a procedure end.*

  For a procedure with the interval $[s, e)$, incorrectly inferring the procedure end at address $e' > e$ will not change the unwind recipes that we compute for the interval $[s, e)$. This rule is especially relevant when data or alignment bytes separate two procedures.

- *We assume all procedures are contiguous.*

  In other words, we assume a single procedure is *not* divided into disjoint code segments. For the most part, this assumption holds. We have, however, encountered compilers that employ hot-cold optimization [43]. This optimization sometimes splits the procedure into disjoint segments. Furthermore, an unrelated procedure may be placed between the disparate parts of the hot-cold-optimized procedure. Our treatment of a divided procedure is to treat each part as a separate procedure. Our treatment simplifies procedure discovery, but requires additional consideration when determining the unwind recipe for the various segments of a divided procedure. See Section 3.2.2 for more information.

- *Not all false positives are equally problematic.*

We classify false procedures starts into two categories: *malignant* and *benign*. If we infer a false procedure start in a gap between two real procedures that contains data (e.g., a jump table for a switch statement), this will not affect the bounds of any real procedures for which we need to compute unwind intervals. For this reason, we call such a false procedure start benign. On the other hand, if we infer a false procedure start s' in the middle of a real procedure ranging from $[s, e)$, this may cause us to compute incorrect unwind information for the interval $[s', e)$. We call such a false procedure start malignant.

Approach

We take an aggressive approach to procedure discovery. Without evidence to the contrary, we assume that the instruction following an unconditional jump or a return is the start of a new procedure. In optimized code, we have also seen procedures that end with a call to a procedure that doesn't return (e.g., `exit` or `abort`). To handle this case, we infer a function start after a call if we immediately encounter code that is obviously a function prologue. We use the following collection of heuristics to avoid inferring a procedure start within a procedure (a malignant false positive).

- We call the interval between a conditional branch at an address a and its target at address t a *protected interval*. No procedure start will be inferred in a protected interval. If $a < t$, this yields a protected interval $[a, t')$, where t' is the end of the instruction at address t; otherwise, this yields a protected interval $[t, a')$, where a' is the end of the instruction at address a. (Conditional jumps are almost always within procedures. While we have found one or two conditional forward branches used as tail calls in `libc`, other heuristics prevent us from missing procedure starts in this rare case.)

- A backward unconditional jump at address a into a protected interval that extends from $[s, e)$ extends the protected interval to cover the range $[s, a')$, where a' is the end of the instruction at address a. (Such jumps often arise at the end of 'cold path' prefetching code that has been outlined from loops and deposited after what would have been the end of the procedure.)

- Moving the stack pointer upward at address a in a procedure prologue (to allocate stack space for local variables) must be followed by a compensating adjustment of the stack pointer in each of the procedure's n epilogues, at addresses $e_1, \ldots, e_n$. Let e_n be the epilogue with the largest address. We treat the interval $[a, e_n)$ as protected.

- Let the interval between initializing the frame pointer register with the value of the stack pointer and restoring the value of the frame pointer be a protected interval. Similarly, let the interval between a 'store' and 'load' of the frame pointer be a protected interval.

- A global symbol in the symbol table or the dynamic symbol table is always considered a procedure start, even if it lies within a protected interval. In contrast, a local symbol only considered a procedure start if it does not fall within a protected interval.

3.2.2 Computing Unwind Recipes

Because dynamic analysis must be efficient, we prefer fast linear-time heuristics that are typically accurate over slower fully general methods. Experiments described in Section 3.2.3 show that our approach is nearly perfect in practice. Although we initially developed our strategy for computing unwind recipes for x86-64 binaries, the general approach is architecture independent. We have adapted it to compute unwind recipes for MIPS and PowerPC binaries to support call path profiling on SiCortex[4] clusters and Blue Gene/P, respectively.

Our binary analyzer creates an unwind recipe for each distinct interval within a procedure. An interval is of the form $[s, e)$ and its unwind recipe describes where to find the caller's program counter, frame pointer (FP) register value, and stack pointer (SP). For example, the caller's program counter (the current frame's return address) can be in a register, at an offset relative to SP or at an offset relative to FP; the value of the caller's FP register, which may or may not be used by the caller as a frame pointer, is analogous.

The initial interval begins with (and includes) the first instruction. The recipe for this interval describes the frame's state immediately after a call. For example, on x86-64, a procedure frame begins with its return address on the top of stack, the caller's value of FP in register FP, and the caller's value of SP at SP $-$ 8, just below the return address (where 8 is the size of the return address). In contrast, on MIPS, the return address is in register RA and the caller's value of FP and SP are in registers FP and SP, respectively.

The analyzer then computes unwind recipes for each interval in the procedure by determining where each interval ends. (Intervals are contiguous and cannot overlap.) To do this, it performs a linear scan of each instruction in the procedure. For each instruction, the analyzer determines whether that instruction affects the frame. (For x86-64, where instruction decoding is challenging, we use Intel's XED2 tool [38].) If so, the analyzer ends the current interval and creates a new interval at the next instruction. The unwind recipe for the new interval is typically created by applying the instruction's effects to the previous interval's recipe. An interval ends when an instruction:

[4]This work was completed before SiCortex's unfortunate demise in May 2009.

1. modifies the stack pointer (pushing registers on the stack, subtracting a fixed offset from SP to reserve space for a procedure's local variables, subtracting a variable offset from SP to support `alloca`, restoring SP with a frame pointer from FP, popping a saved register);
2. assigns the value of SP to FP to set up a frame pointer;
3. jumps using a constant displacement to an address outside the bounds of the current procedure (performing a tail call);
4. jumps to an address in a register when SP points to the return address;
5. returns to the caller;
6. stores the caller's FP value to an address in the stack; or
7. restores the caller's FP value from a location in the stack.

There are several subtleties to the process sketched above: following a return or a tail call (items 4 and 5 above), a new interval begins. What recipe should the new interval have? We initialize the interval following a tail call or a return with the recipe for the interval that we identify as the *canonical frame*. We use the following heuristic to determine the canonical frame **C**. If a frame pointer relative (FP) interval was found in the procedure (FP was saved to the stack and later initialized to SP), let **C** be the first FP interval. Otherwise, we continue to advance **C** along the chain of intervals while the frame size (the offset to the return address from the SP) is non-decreasing, and the interval does not contain a branch, jump, or call. We use such an interval as a signal that the prologue is complete and the current frame is the canonical frame. In addition, whenever a return instruction is encountered during instruction stream processing, we check to make sure that the interval has the expected state: e.g., for x86-64, the return address should be on top of the stack, and the FP should have been restored. If the interval for the return instruction is not in the expected state, then the interval that was most recently initialized from the canonical frame is at fault. When a return instruction interval anomaly is detected, we adjust all of the intervals from the interval reaching the return back to the interval that was most recently initialized from the canonical frame.

To handle procedures that have been split via hot-cold optimization, we check the end of the current procedure p for a pattern that indicates that p is not an independent procedure, but rather part of another one. The pattern has two parts:

1. p ends with an unconditional branch to an address a that is in the interior of another procedure q.

2. The instruction preceding a is conditional branch to the beginning of p.

When the hot-cold pattern is detected, all intervals in p are adjusted according to the interval computed for a.

In the linear scan between the start and end address of a procedure, the analyzer may encounter embedded data such as jump tables. This may cause decoding to fail or lead to corrupt intervals that would leave us unable to unwind. Although such corrupt intervals could cause unwind failures (we note such failures in a log file), we have not found them to be a problem in practice. This is because x86/x86-64 disassembly tends to be self-synchronizing [122].

3.2.3 Evaluation

To evaluate the efficiency and effectiveness of our binary analyses for unwinding against contemporary tools, we compared `hpcrun` with two of the tools from Intel's Performance Tuning Utility (PTU) [7, 75] — PTU's sampling-based call path pro-

Integer programs

Benchmark	Overhead hpcrun	Overhead PTU-sample	Overhead PTU-Pin	Unwind Failures hpcrun[‡]	Unwind Failures PTU-sample Intel	Unwind Failures PTU-sample Others
400.perlbench	1.3%	0.9%	1043.3%	0.0	4.5%	87.5%
401.bzip2	2.9%	0.9%	197.1%	0.0	0.8%	52.2%
403.gcc	3.2%	1.3%	300.9%	15.1	4.5%	70.7%
429.mcf	1.3%	2.6%	8.5%	0.0	0.1%	60.4%
445.gobmk	1.7%	1.3%	481.3%	0.1	2.4%	71.6%
456.hmmer	0.4%	1.0%	36.4%	0.0	0.1%	74.4%
458.sjeng	0.3%	1.6%	694.4%	0.0	19.2%	100.0%
462.libquantum	-0.2%	-0.2%	16.3%	0.0	0.1%	99.9%
464.h264ref	0.1%	0.0%	784.2%	0.6	21.9%	69.7%
471.omnetpp	1.6%	1.7%	701.2%	0.0	1.4%	49.4%
473.astar	1.6%	1.7%	184.1%	0.0	0.5%	57.6%
483.xalancbmk	9.5%	10.8%	732.0%	0.0	1.0%	0.4%
Average*	2.0%	1.9%	431.6%	1.3	4.7%	66.1%
Std. Dev.	2.6%	2.8%	353.4%	4.3	7.6%	26.6%

Floating-point programs

Benchmark	Overhead hpcrun	Overhead PTU-sample	Overhead PTU-Pin	Unwind Failures hpcrun[‡]	Unwind Failures PTU-sample Intel	Unwind Failures PTU-sample Others
410.bwaves	1.7%	1.9%	9.9%	0.0	0.0%	66.6%
416.gamess	0.8%	0.1%	†	0.0	0.3%	99.7%
433.milc	0.6%	0.4%	61.0%	0.0	0.0%	99.9%
434.zeusmp	2.1%	2.0%	†	0.0	0.0%	99.7%
435.gromacs	0.6%	0.4%	57.3%	0.0	0.1%	100.0%
436.cactusADM	1.6%	1.5%	6.7%	0.0	0.0%	100.0%
437.leslie3d	2.0%	1.7%	2.5%	0.0	0.0%	93.5%
444.namd	0.2%	1.5%	5.1%	0.0	0.0%	42.0%
447.dealII	0.5%	0.7%	1746.4%	0.0	2.7%	83.8%
450.soplex	1.6%	1.8%	19.3%	0.0	2.0%	54.3%
453.povray	0.1%	0.3%	1732.8%	0.0	6.5%	49.8%
454.calculix	-0.5%	0.9%	62.5%	0.0	0.2%	99.5%
459.GemsFDTD	-0.8%	-1.2%	45.3%	0.0	0.1%	74.9%
465.tonto	0.3%	1.3%	287.4%	0.0	11.1%	98.0%
470.lbm	0.9%	1.2%	10.2%	0.0	0.0%	13.5%
481.wrf	3.0%	1.5%	59.5%	0.5	0.0%	98.2%
482.sphinx3	0.4%	2.4%	84.7%	0.0	1.9%	48.0%
Average*	0.9%	1.1%	279.4%	0.0	1.5%	77.7%
Std. Dev.	1.0%	0.9%	566.0%	0.1	3.0%	27.1%

* Neither the arithmetic nor geometric mean summarizes these values well.
† PTU-Pin failed to execute any version of these benchmarks.
‡ These values are *not* percents.

Figure 3.2: Comparing hpcrun's and Intel PTU's overhead and unwind failures on SPEC CPU2006.

filer (PTU-sample) and PTU's Pin-based call graph profiler (PTU-Pin) — using the SPEC CPU2006 benchmarks [134]. Since PTU is designed for Intel architectures, this evaluation focuses on analysis of x86-64 binaries. We compiled two versions of each benchmark, distinguished by 'base' or 'peak' optimization, using the Intel 10.1 (20080312), PathScale 3.1 and Portland Group (PGI) 7.1-6 compilers; this resulted in six versions of each benchmark. We used the following 'base' and 'peak' optimization flags: for Intel, `-O3` and `-fast` (but with static linking disabled); for PathScale, `-O3` and `-Ofast`; for PGI, `-fast -Mipa=fast,inline`. To permit high-throughput testing, we performed the experiments on a cluster where each node is a dual-socket Intel Xeon Harpertown (E5440) with 16 GB memory running Red Hat Enterprise Linux 5.2. Figure 3.2 summarizes our results.

Efficiency

The first multi-column of Figure 3.2 compares the average overhead of `hpcrun` with PTU-sample and PTU-Pin. We first observe that despite PTU-Pin's sophistication, dynamic binary instrumentation is not an acceptable measurement technique for two reasons. First, compared to a worst case sampling overhead of about 10% (average of 1–2%), instrumentation can introduce slowdown *factors* of 10-18. Second, the drastic variation in overheads strongly suggests that Pin's instrumentation dilates the execution of small procedures and introduces systematic distortion. Because of the extremely long run times and the clear advantage of sampling, we chose not to collect PTU-Pin results on executables generated by non-Intel compilers, assuming that an Intel tool used with an Intel-generated executable represents a best-case usage.

Both `hpcrun`'s and PTU-sample's results are averaged over all six versions of the benchmarks; each tool used a 5 ms sampling period, yielding approximately 200 samples/second. Because of `hpcrun`'s additional dynamic binary analysis, one might expect it to incur more overhead. However, our results show that a reasonable execution time and sampling rate quickly amortizes the binary analysis overhead over thousands of samples and makes it negligible.[5] In fact, the overhead differences between hpcrun and PTU are statistically insignificant. This is seen in two ways. First, the average overheads for each set of benchmarks are very similar; and given the high standard deviations, a statistical test would not meaningfully distinguish between the two. Second, average overheads for the individual benchmarks are within within 1–2% of each other, but no tool consistently performs better. Moreover, these small differences are well within the natural execution-time variability for a standard operating system (especially when using shared I/O) [109]; this fact accounts for the small *negative* overheads.

The one benchmark for which both hpcrun and PTU incur meaningful overhead is 483.xalancbmk, at around 10%. The reason is that 483.xalancbmk has many call paths that are 1000-2000 invocations long. An earlier version of hpcrun for the Alpha platform used a technique of inserting an 'active return' on a sample to memoize stack unwinds and collect return counts [60]. We plan to implement this technique and expect that it will significantly reduce hpcrun's overhead in such cases.

Effectiveness

Given that hpcrun and PTU-sample incur comparably low overheads, multi-column two of Figure 3.2 assesses the quality of their call path profiles in terms of unwind failures. An unwind failure is defined as the inability to collect a complete calling context. Note that for hpcrun, this metric directly assesses the quality of

[5]Although it is more difficult to amortize the overhead of our binary analyses for very short executions, this does not imply that for such executions tools like PTU-sample that use statically computed unwind information induce significantly less overhead. Because typical compiler-generated unwind information is stored sparsely, a tool like PTU-sample must invest some effort to read and interpret it.

unwind recipes and indirectly reflects the accuracy of procedure bounds. This is a reasonable metric because we have designed hpcrun's binary analyses to cooperate for the purpose of obtaining accurate unwinds.

There are two ways to directly measure unwind failures. The most comprehensive method uses binary analysis to attempt to verify each link in the recovered call chain. For each each step in the unwind, we have a segment $p \rightarrow q$ and a return address (RA) within p. The analysis can then certify the unwind from q to p as (almost certainly) *valid*, *likely*, or (provably) *invalid*:

- *valid*, if a statically linked call to q immediately precedes RA
- *valid*, if a dynamically linked call to q immediately precedes RA (via inspection of the procedure linkage table)
- *likely*, if a dynamically dispatched call immediately precedes RA
- *likely*, if a call to procedure r immediately precedes RA, and r is known to have tail calls
- *invalid*, if none of the above apply

Two details are worth noting. First, for architectures with variable-width instructions, it is reasonable to simply test offsets from RA that correspond to possible call or jump instructions rather than disassembling from the beginning of the procedure. Second, delay slots will offset the location of the call site.

The second way to measure unwind failures is based on the observation that, in practice, if an unwinder attempts to use an incorrect frame or stack pointer, errors very quickly accumulate and result in return addresses that are provably wrong in that they do not correspond to mapped code segments. Additionally, we make use of the fact that hpcrun's program monitoring technology intercepts a process's or thread's entry point (for both statically and dynamically linked binaries). Thus, this second method classifies an unwind as invalid if it finds a provably wrong return address or if the unwind is not rooted in the process's or thread's entry point.

`hpcrun` implements both methods. Because the first and stronger method incurs noticeable overhead, we do not activate it by default. Rather, we use the second method and make a note of all invalid unwinds. This gives us an efficient way to directly assess unwind failures.

In contrast, for PTU-sample, we measured unwind failures indirectly. PTU-sample does retain partial unwinds; and if it performs any sort of verification, that information is not exported. Therefore, we wrote a script to analyze the results of PTU-sample's 'hot path' listing. The script classifies a path as valid if it is rooted at some variant of 'main' or any ancestor frame. Observe that this requirement is more relaxed than `hpcrun`'s. It is also worth noting that this requirement does *not* not penalize PTU-sample for skipping a frame by incorrectly following its *parent*'s frame pointer rather than its own — an easy mistake for an x86-64 tool that is unwinding from an epilogue or frame-less procedure and that relies on compiler-generated unwind information.

Our results showed radically different failure rates for PTU-sample on Intel-generated code (5%) versus PathScale and PGI code (65-75%). Since PTU-sample is dependent upon frame pointers and unwind information, and since frame pointers are not reliably maintained in these binaries, the results strongly suggest that, compared to PathScale and PGI, the Intel compiler places a much higher priority on consistently recording correct unwind information. However, even on Intel-generated binaries, PTU-sample can have high enough failure rates — as high as 5-20% — that it risks introducing systematic distortion by failing to unwind through a commonly appearing procedure instance. On the non-Intel benchmark versions, PTU-sample's failure rate is so high that it essentially becomes a call path *fragment* profiler.

In contrast, the number of unwind failures for `hpcrun` is vanishingly small. `hpc-run`'s failures are reported as the average *number* (not percent) of failures over all six benchmark versions. Its worst performance was on the 403.gcc benchmark. The benchmark averages on the order of 100K samples. Across the six versions of the benchmark that we studied, `hpcrun` failed to gather a full call path for 15.1 of those samples on average. All of these failures stem from a calling-context-sensitive frame formed by a procedure calling `abort()` to handle an error. Specifically, the Intel compiler recognizes that the call to `abort()` never returns and uses this information to tear down the procedure's current frame *before* the call `abort()` occurs, something that usually only occurs before a tail call or a return instruction. Since the procedure must return in the non-exceptional case, it contains an additional epilogue to tear down the procedure's frame before the return statement. As a result, our heuristics detect an inconsistency in the computed unwind recipes and attempt to self-correct, but are unable to fully account for the context-sensitive complexity.

Summary

Despite the fact that `hpcrun`'s binary analysis for unwind recipes is (1) context insensitive, (2) operates without a control flow graph, (3) does not formally track register values, and (4) cannot treat embedded data as such, these results show that the cost of our analysis is very modest and its results are very effective. Given that `hpcrun` almost always collects a full call path and that PTU-sample much more frequently fails, we can say that on average `hpcrun` performs more useful work per sample than PTU-sample — at the same overhead.

```
(LM /mypath/hmc                                        load module
  (File /mypath/hmc.cc                                 source file
    (Proc doHMC 257-449 {[0xabe-0xfeed)}               procedure
      (Stmt 309-309 {[bab1-0xbabe)} )                  statement
      (Loop 311-435 {[0xdad-0xfad)}                    loop
        (Stmt 313-313 {[0xdaf-0xea1), [ee1-0xeef)} )
      ))))
```

Figure 3.3: Representing program structure with a mapping between object code and source-code structure. Static scopes include a load module, file, procedure, loop and statement. Procedures, loops and statements are annotated with their corresponding object address interval sets.

The clearest downside to our approach is the effort we have invested in developing these heuristics. The x86-64 unwinder was the most difficult to write, in large part because of its irregular architecture and variable-sized instructions. Nevertheless, once we arrived at the general approach we were able to relatively quickly develop MIPS and PowerPC unwinders. For example, we wrote the PowerPC unwinder — for use on Blue Gene/P — and resolved some OS-specific issues in about a week and a half. During our first major test, we collected performance data for an 8192-core execution of the FLASH astrophysics code [52] compiled with the IBM XL Fortran and C compilers for BG/P (versions 11.1 and 9.0, respectively) using options `-O4 -qinline -qnoipa`.[6] Out of approximately 1 billion total samples, `hpcrun` failed to unwind approximately 13,000 times — a failure rate of 0.0013%.

3.3 Binary Analysis for Source-Level Attribution

This section discusses the `hpcstruct` binary analysis tool for recovering static program structure from a binary. Although originally presented in our M.S. thesis [139], we include this summary for the sake of completeness.

[6]We were forced to disable inter-procedural analysis because of an incompatibility between IBM's compiler and our tool for inserting `hpcrun` in statically linked binaries.

To combine dynamic call path profiles with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure. Since the most important elements of the source code structure from the perspective of performance are procedures and loop nests, we focus our efforts on them. An example of what this mapping might look like is shown in Figure 3.3. The mapping is a tree of scopes representing static program structure. The scope hierarchy is straightforward: a load module (a binary) contains source files; files contain procedures; procedures contain loops; procedures and loops contain statements; and scopes such as procedures, loops and statements can be annotated with object code address interval sets.

There are two ways to obtain the desired mapping: use a summary of transformations recorded by the compiler or reconstruct it through analysis. Because debuggers must associate the execution of object code with source code, one would expect debugging information to provide the former. In 1992, Brooks et al. [27] developed debugging extensions for mapping object code to a scope tree of procedures, loops, blocks, statements and expressions. While they left to future work a solution for the inlining problem, neither compilers nor debugging formats followed their lead. Although DWARF [57], the *de facto* standard on Linux, can represent inlining, it cannot describe loops or loop transformations. Even worse, all x86 Linux compilers that we have used generate only limited DWARF, often failing to record inlining decisions. Intel's compiler (10.x) retains line-level information in the presence of inlining, but the information is incomplete (e.g., there is no association between inlined code and object code) and sometimes erroneous. Thus, however easy the problem of creating the object to source code mapping could have been, the fact remains that vendor compilers do not provide what we desire. Consequently, we wrote the `hpcstruct` tool to reconstruct the mapping through binary analysis, using only a 'lowest common

Address	File	Line	Procedure
`0x...15550`	hmc.cc	499	`main`
`0x...15570`	hmc.cc	14	`main`
`0x...17030`	qdp_multi.h	35	`main`
`0x...172c0`	stl_tree.h	1110	`main`

Figure 3.4: Example of typical line map information.

denominator' set of debugging information. We focus on programs written in C++, C, and Fortran.

An obvious starting point is to consult an executable's line map, which maps an object address to its corresponding source file, line number and procedure name for use by a debugger. However, the line map is insufficient for detecting inlined, or more generally, *alien* code, i.e., code that originates outside of a given procedure. To see this, consider the unexceptional line map excerpt from a quantum chromodynamics code shown in Figure 3.4. Given that the first entry maps to native (as opposed to alien) code, what is the first line of procedure `main`? Although one is tempted to answer 14, it turns out that the second line is actually alien; this is not detectable because the line map retains the *original* file and line information (from *before* inlining) but assumes the name of the host procedure (*after* inlining). Even worse, because optimizing compilers reorder the native and alien instructions (including prologues and epilogues), no particular entry is guaranteed to map to native code, much less the procedure's begin or end line. Consequently, to reconstruct the desired mapping we must supplement the line map with a 'lowest common denominator' set of DWARF-specific information.

3.3.1 Recovering the Procedure Hierarchy

Compilers perform several procedure transformations such as flattening nested procedures, inlining, and cloning for specialization. Recovering the procedure hierarchy involves re-nesting source code procedure representations, determining their source line bounds and identifying alien code.

It turns out that by combining standard DWARF information with certain procedure invariants, recovering the procedure hierarchy is less difficult than it first appears. A load module's DWARF contains procedure descriptors for each object procedure in the load module *and* the nesting relationship between the descriptors. Each descriptor includes (1) the procedure's name, (2) the defining source file and begin line, and (3) its object address ranges. The key missing piece of information is the procedure's end line. Observe however, that two source procedures do not have overlapping source lines unless they are the same procedure or one is nested inside the other. Intuitively, in block structured languages, source code does not 'overlap.' More formally:

Non-overlapping Principle. *Let scopes x_1 and x_2 have source line intervals σ_1 and σ_2 within the same file. Then, either x_1 and x_2 are the same, disjoint or nested, but not overlapping:*[7]

- $(x_1 = x_2) \Leftrightarrow (\sigma_1 = \sigma_2)$
- $(x_1 \neq x_2) \Leftrightarrow ((\sigma_1 \cap \sigma_2 = \emptyset) \vee (\sigma_1 \subset \sigma_2) \vee (\sigma_2 \subset \sigma_1))$

We can also say (where $x_2 \sqsubset\!\!\!< x_1$ means x_1 is nested in x_2):

- $(\sigma_1 \cap \sigma_2 = \emptyset) \Leftrightarrow ((x_1 \neq x_2) \wedge \neg(x_1 \sqsubset\!\!\!< x_2) \wedge \neg(x_2 \sqsubset\!\!\!< x_1))$
- $(\sigma_2 \subset \sigma_1) \Leftrightarrow (x_1 \sqsubset\!\!\!< x_2)$

The implication of this principle is that given DWARF nesting information, we can infer end line bounds for procedures, resulting in the following invariants:

[7]Unstructured programming constructs may give rise to irreducible loops or alternate procedure entries. While the former is not strictly an exception (no block of source code actually overlaps), the latter is. However, Fortran's alternate entry statement is deprecated and used very infrequently.

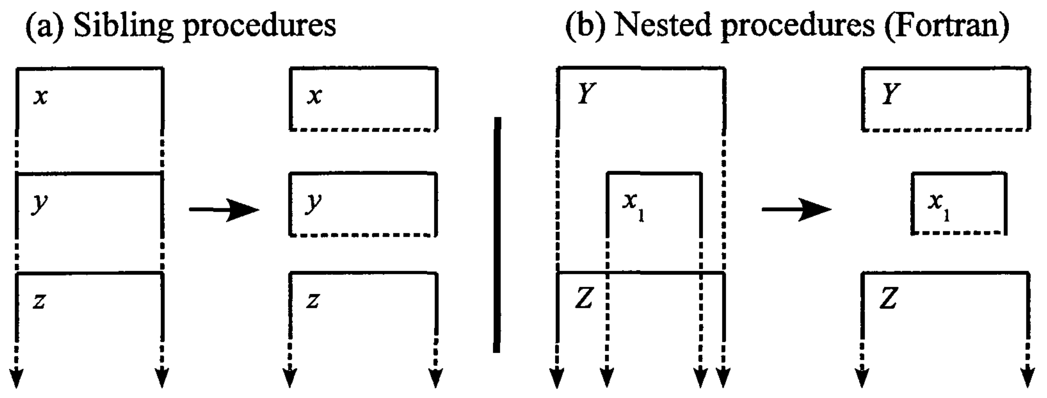


Figure 3.5: Bounding procedure end lines.

Procedure Invariant 1. *A procedure's bounds are constrained by any (parent) procedures that contain it.*

Procedure Invariant 2. *Let procedure y have sibling procedures x and z before and after it, respectively. Then, y's begin line is greater than x's end line and its end line is less than z's begin line.*[8] Figure 3.5a graphically depicts application of this invariant.

Neither C++ nor C permits procedure nesting. To handle Fortran, which places strict limits on where a procedure can be nested, we derive a special invariant (depicted graphically in Figure 3.5b):[9]

Procedure Invariant 3. *Let procedure Y have nested procedures $x_1 \ldots x_n$, in that order. Then Fortran nesting implies that the executable code of Y and $x_1 \ldots x_n$ forms $n + 1$ ordered, contiguous source code regions.*

These invariants enable hpcstruct to infer an upper bound on all procedure end lines except for the last top-level procedure of a source file, whose upper bound is ∞.

[8]We can ignore the case where two procedures are defined on the same source line; column information would make this precise.

[9]Because DWARF contains a language identifier, this nesting rule can be applied only when appropriate.

Moreover, accurate procedure bounds information is sufficient for detecting *all* alien code within a procedure (assuming two restrictions discussed below).

There are two complications with this strategy. First, it is often the case that a load module's DWARF does not contain a DWARF descriptor for every source-level procedure, creating 'gaps' in the procedure hierarchy. For example, no descriptor is generated for a C++ static procedure that is inlined at every call site. Although this knowledge can never be fully recovered, we have developed a simple and effective heuristic to close most of the important gaps [139].

Second, C++ permits classes to be declared within the scope of a procedure, thereby allowing class member functions to be transitively nested within that procedure. Consider a procedure-scoped C++ class with n member functions. The n^{th} member function may be inlined into the procedure but because the only end line bound we can establish on the n^{th} member function is the end line bound of the containing procedure itself, we will not be able to detect it. This means that in the presence of procedure-scoped classes, *even* with DWARF descriptors for every procedure we may not be able to detect all alien code. However, this issue is of little practical concern: procedure-scoped classes are rare; and we have developed a strategy for detecting the presence of most procedure-scoped classes [139].

A high-level sketch of `hpcstruct` is shown in Algorithm 3.2. It consists of two parts: recovering the procedure hierarchy (beginning at line 3) and recovering loop nests for each procedure (beginning at line 5). This section has covered the first part; the second part is covered below.

3.3.2 Recovering Alien Contexts

Before discussing loops, we note three important aspects of detecting alien code.

```
(File main.cpp
  (Proc zoo 10-100
A1  (Alien |zoo moo.cpp:10-13|
      ... )
L1  (Loop 20-50
A2    (Alien |zoo moo.cpp:10-15|
        ... )
```

```
(Proc ...
  (Alien1 ...
    (Loop1 ...
      ...
        (Alienm ...
          (Loopm ...
            (Alienm+1 ...
              (Stmt s...)
```

Figure 3.6: Recovering alien contexts: (a) Alien context ambiguity; (b) Maximum procedure context nesting for scope s.

Figure 3.6a shows an example of two alien scopes, A_1 and A_2, representing the presence of alien code within procedure zoo. Consider the task of identifying the alien code within zoo. In general, given an object code instruction, its corresponding source-level statement is classified as alien if its source file is different than the enclosing procedure's or if its source line is outside the line bounds of the enclosing procedure's. However, as an instruction is processed, adjacent instructions may belong to different alien contexts (i.e., different inlined procedures). Since inlining can be nested, it is natural to ask how to distinguish between nested and non-nested inlining. The short answer is that without DWARF inlining or source-level call graph information, we cannot. Therefore, we choose to flatten alien scopes with respect to their enclosing loop or procedure. This implies that for a loop nest of depth m, there can be at most $m + 2$ parent contexts (procedure or alien scopes), as illustrated in Figure 3.6b.

Return again to Figure 3.6a. Observe that A_1 and A_2 have overlapping bounds, where A_2 is embedded within loop L_1. Without call site information, it is not possible to distinguish between (1) one distinct call site within the loop, where some of the inlined code was was loop invariant; or (2) two distinct call sites where some of the code from the first call site (A_1) was entirely eliminated.

Finally, the number and bounds of alien scopes can be refined using the Non-overlapping Principle [139].

3.3.3 Recovering Loop Nests

Having an outline of the procedure hierarchy, hpcstruct recovers the loop nesting structure for each procedure. As shown in Algorithm 3.2, this task can be broadly divided into two components: (1) analyzing object code to find loops (line 6) and (2) inferring a source code representation from them (line 7). To find loop nests within the object code, hpcstruct first decodes the machine instructions in a procedure to compute the control flow graph (CFG) and then uses Havlak's algorithm [70] to recover the tree of loop nests [93]. Given this tree of object code loops, hpcstruct then recovers a source code representation for them. This is a challenging problem because with fundamentally line-based information hpcstruct must distinguish between (1) loops that contain inlined code, (2) loops that may themselves be inlined, and (3) loops that may be inlined *and* contain inlined code. Finally, hpcstruct must account for loop transformations such as software pipelining.

Because loops also obey the Non-overlapping Principle, there are analogous loop invariants for Procedure Invariants 1 and 2. However, without symbolic loop information, these invariants are of little value. Consequently, hpcstruct's strategy is to initially assume that the source loop nesting tree mirrors the object code loop tree, and then look for exceptions. Specifically, hpcstruct performs a preorder traversal of the object loop tree, recursively visiting outer loops before inner loops. The challenge we now discuss is reconstructing a source representation for every loop during this traversal.

As a starting point, we observe that loop invariant code motion implies that *a computation at loop level l will (usually) not be moved into a loop that is at a nesting* *level deeper than l*. Coupling this observation with accurate procedure bounds, we could scan through all the non-alien statements within a particular loop and compute a minimum and maximum line number, which we call the min-max heuristic.

One complication for the min-max heuristic is Fortran's use of *statement functions*, which are single-statement functions nested within a procedure. Statement functions have no associated DWARF descriptors. Code for statement functions is forward substituted wherever they are used. Applying the min-max heuristic to the first loop of a procedure that uses a statement function will result in a loop begin line that erroneously includes all executable statements prior to the loop. To prevent this problem, we would like some mechanism for estimating the begin line of a loop. When loops are compiled to object code, the loop header's continuation test is typically translated into a conditional backward branch that, based on the result of the continuation test, returns to the top of the loop or falls through to the next instruction. Moreover, most compilers associate the loop's backward branch with the source line of the continuation test, and therefore the loop header. We therefore modify the simple min-max heuristic to form the bbranch-max heuristic for computing loop begin and end lines: the loop begin line can be approximated using information from the backward branch; and the best loop end line is the maximum line after all alien lines have been removed.

Although the bbranch-max heuristic can be thwarted by unstructured control flow, it suffers from a more serious defect. The difficulty is that when estimating a loop's begin line from that loop's continuation test, the heuristic implicitly determines the loop's *procedure context*, i.e., the loop's enclosing alien or procedure scope. Specifically, bbranch-max assumes that the procedure context for that instruction is the same context as other instructions within the (object) loop body. This results in a severe problem if the loop's condition test derives from inlined code, something that

Algorithm 3.2: recover-program-structure: Recover static source code structure from an application binary.

```
Input: A load module $lm$ (with DWARF information)
Result: $\mathcal{S}$, $lm$'s object to source code structure map
1  let $\mathcal{D}$, dwarf map : $object\text{-}procedure \mapsto DWARF\text{-}descriptor$
2  let $\mathcal{L}$, line map : $address \mapsto \langle file\text{-}name, proc\text{-}name, line \rangle$
   // Recover procedure hierarchy (§3.3.1)
3  Create a source procedure $p_S$ for each DWARF descriptor in $\mathcal{D}$ with no object code
4  Create a source procedure $p_S$ for each object-procedure $p_O$ using $\mathcal{D}(p_O)$ or $\mathcal{L}(p_O)$.
   // Recover loop nests (§3.3.3)
5  foreach procedure $p_S$ in $\mathcal{S}$ with object-procedure $p_O$ do
6      Form $p_O$'s loop nests by creating the strongly connected regions tree $T$ induced
         by $p_O$'s control flow graph
7      foreach basic block $b$ in $T$ (preorder traversal) do
8          if $b$ is a loop header then
9              let $\sigma = \mathcal{L}(i)$ for backward-branch $i$
10             let $es_S$ = determine-context($\sigma$)
11             Create a source code loop $l_S$ located within $es_S$
12         foreach instruction $i$ in $b$ do
13             let $\sigma = \mathcal{L}(i)$
14             let $es_S$ = determine-context($\sigma$)
15             Create a statement scope $s_S$ for $\sigma$ within $es_S$
16 Normalize each procedure $p$ in $\mathcal{S}$ (§3.3.4)
```

Algorithm 3.3: determine-context: Determine the static context of a loop or statement.

```
Input: Let ($s_e$, $\sigma = \langle fnm, pnm, ln \rangle$) be the argument list. Let scope $s$ be a loop or
       statement whose context is unknown. Then $s_e$ is $s$'s expected enclosing
       scope (loop or procedure) and $\sigma$ its source code descriptor.
Result: The actual enclosing scope $c$ (loop or procedure context) for $s$.
```

is *very* common within object-oriented C++. Therefore, it is necessary to somehow distinguish between a loop deriving from an alien context (and which itself may have alien loops) and one that only contains alien contexts within its header or body. As previously suggested, our solution to this problem, is to *guess and correct.* In brief, `hpcstruct` processes instructions within a loop one-by-one (Algorithm 3.2, line 7); and for each instruction it determines that instruction's procedure context, its source

```
(File main.cpp                                   Steps
  (Proc init 145-199
A1  (Alien ... Array.cpp:82-83                   1. Find alien context
S1    (Stmt 82-82)
L2    (Loop 83-83                                2. Locate loop (incorrectly)
S2      (Stmt 83-83)
A3      (Alien ... main.cpp:158-158
S3        (Stmt 158-158)                         3. Self nesting!
```

Figure 3.7: Detecting incorrect loop placement via nesting cycles while recovering program structure.

line location within that context, and its enclosing loop (if any). Figure 3.7 shows a partially reconstructed procedure where alien scope A_1 has been identified (Step 1) by using the source line information for the instruction corresponding to S_1. When `hpcstruct` processes the loop header (S_2) for L_2 using bbranch-max (Step 2), it must determine whether the source line loop should be located in the current procedure context, a prior context (which would imply the current context is alien), or a new alien context. In this case, because of the presence of statement S_2, `hpcstruct` 'guesses' that the loop header should be located within the current alien procedure context A_1. `hpcstruct` next processes S_3 (Step 3), which it determines must be alien to the current procedure context A_1, resulting in the new alien context A_3. However, because A_3's bounds are within `init`'s bounds, this implies that `init` is inlined inside of itself, which is a contradiction. This shows that the guess at Step 2 was wrong.

This observation, which is another implication of the Non-overlapping Principle, can be formally stated as follows:

Procedure Invariant 4. *Let L be a loop nest rooted in an alien scope C_a. Furthermore, let L have loop levels $1 \ldots n$. Now, let s be a statement at level n that clearly belongs in a shallower procedure context C'. Since C' is a shallower procedure*

```
    Before                              After
    (File main.cpp                      (File main.cpp
      (Proc init 145-199                  (Proc init 145-199
A1      (Alien Array.cpp:82-83>             (Alien Array.cpp:82-83
          (Stmt 82-82)                        (Stmt 82-82)
                                            )
L1        (Loop 83-83)                      (Loop 158-158
                                              (Alien Array.cpp:82-83
S2          (Stmt 83-83)                        (Stmt 83-83)
                                              )
            (Alien main.cpp:158-158
S3            (Stmt 158-158)                  (Stmt 158-158)
```

Figure 3.8: Correcting nesting cycles while recovering program structure.

context, it must be a parent of C_a which implies that C' is nested within itself, which is impossible.

When an impossibility such as this is found, hpcstruct, knowing that L was mislocated, corrects the situation by relocating all levels of L from C_a to within C'. Figure 3.8 shows how we correct the loop nesting cycle shown in Figure 3.7. In this case, L_1 is un-nested one level, which places it within the correct procedure context and its bounds are updated to include S_3. S_2 remains nested in L_1, but A_1's context must be replicated to correctly represent it.

At first glance, the process of selecting the procedure context for a given instruction and possibly correcting an erroneous guess appears to be costly. However, because (1) a loop nest of depth m can have at most $m + 2$ parent contexts and (2) even after inlining, loop nests rarely exceed a depth of 10, scanning the current parent procedure contexts is, for practical purposes, a constant time operation.

Observe that to properly recover the corrected L_1, it is critical to appropriately expand its begin line so that statements that should belong in the loop are not ejected. To do this, we use a tolerance factor when testing for a statement's inclusion within the current loop. If the current begin line minus the tolerance factor would include the statement within the bounds, the statement is deemed to be within the loop and the bounds grow accordingly; the loop's end line can thought of having a tolerance of ∞ to assign the maximum line within the loop as the end line. The effects of fuzzy matching can be complex, because a loop may initially appear to be within an alien context (by backward branch information) but later emerge as a native loop. To account for this, `hpcstruct` uses different tolerances based on context [139].

3.3.4 Normalization

Because of loop transformations such as invariant code motion and software pipelining, the same line instance may be found both within and outside of a loop or there may be duplicate nests that appear to be siblings. To account for such transformations, we developed normalization passes based on the observation that *a particular source line (statement) appears uniquely with a source file* (an application of the Non-overlapping Principle) [93, 139].

For its most important normalization passes, `hpcstruct` repeatedly applies the following rules until a fixed point is reached:

- Whenever a statement instance (line) appears in two or more disjoint loop nests, fuse the nests *but only within the same procedure context.* (Correct for loop splitting.)

- Whenever a statement instance (line) appears at multiple distinct levels of the same loop nest (*i.e., not crossing procedure contexts*), elide all instances other than the most deeply nested one. (Correct for loop-invariant code motion.)

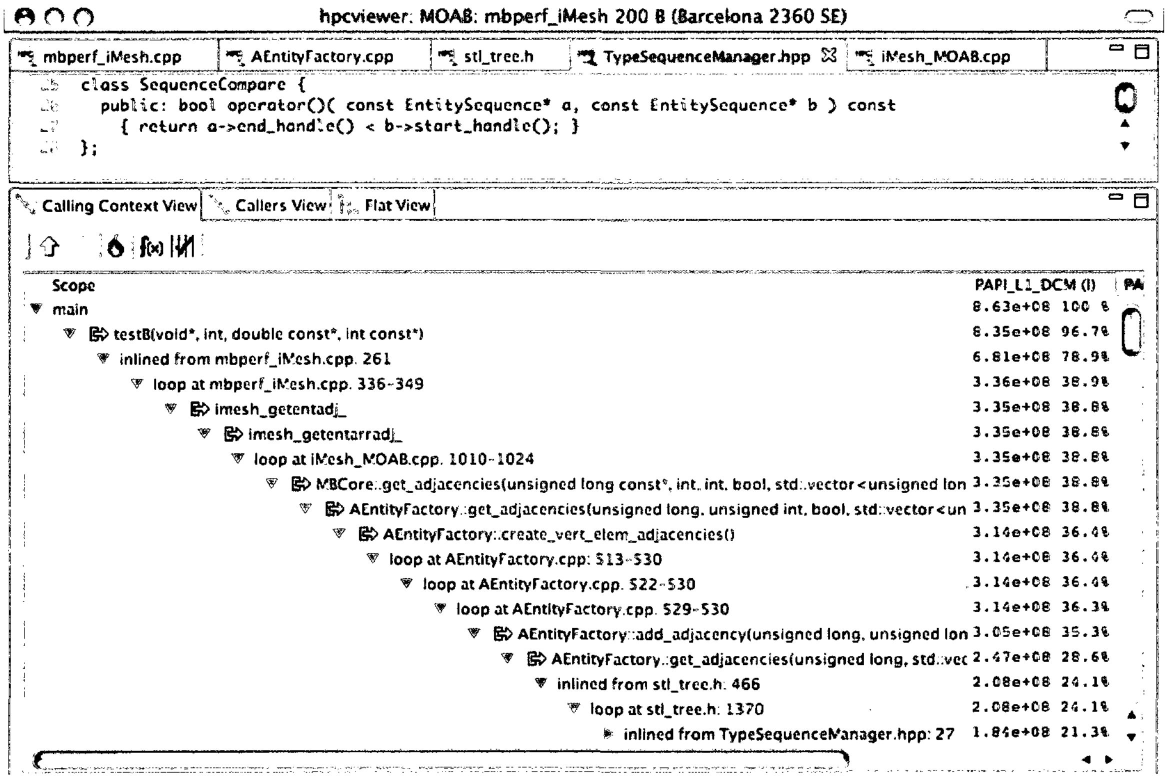
3.3.5 Summary

Thorough application of a small set of invariants enables `hpcstruct` to recover very accurate program structure even in the presence of complex inlining and loop transformations. Importantly, in the (rare) worst case, while the effects of an incorrect inference may be compounded, they are limited to at most *one* procedure. Further details, including discussions of macros, procedure groups and algorithms can be found in [139].

We have tested `hpcstruct` on the GCC, Intel, PathScale, Portland Group and IBM XL compilers (among others). When debugging information is accurate, `hpcstruct` produces very good results. However, we have observed that debugging information from certain compilers is sometimes erroneous — and even violates the DWARF standard. We have hardened `hpcstruct` to handle certain errors, but it cannot psychoanalyze. While compilers may opt to generate incomplete information, the information that they do generate should be *correct*.

3.4 Putting It All Together

By combining `hpcrun`'s minimally intrusive call path profiles and `hpcstruct`'s program structure, we relate execution costs for a fully optimized executable back to static and dynamic contexts overlaid on its source code. One particularly noteworthy result is that `hpcstruct`'s program structure naturally reveals inlining (or the absence of it) as well as loop fusion and the generation of scalarization loops to implement Fortran 90 array notation. To demonstrate our tools' capabilities for analyzing the performance of modular applications, we present screen shots of HPCToolkit's `hpcviewer` browser displaying performance data collected for two modern scientific codes.



```
class SequenceCompare {
  public: bool operator()( const EntitySequence* a, const EntitySequence* b ) const
    { return a->end_handle() < b->start_handle(); }
};
```

```
Scope                                                                                                   PAPI_L1_DCM (I)
main                                                                                                    8.63e+08 100 %
  testB(void*, int, double const*, int const*)                                                          8.35e+08 96.7%
    inlined from mbperf_iMesh.cpp. 261                                                                  6.81e+08 78.9%
      loop at mbperf_iMesh.cpp. 336-349                                                                 3.36e+08 38.9%
        imesh_getentadj_                                                                                3.35e+08 38.8%
          imesh_getentarradj_                                                                           3.35e+08 38.8%
            loop at iMesh_MOAB.cpp. 1010-1024                                                           3.35e+08 38.8%
              MBCore::get_adjacencies(unsigned long const*, int, int, bool, std::vector<unsigned lon   3.35e+08 38.8%
                AEntityFactory::get_adjacencies(unsigned long, unsigned int, bool, std::vector<un      3.35e+08 38.8%
                  AEntityFactory::create_vert_elem_adjacencies()                                        3.14e+08 36.4%
                    loop at AEntityFactory.cpp: 513-530                                                 3.14e+08 36.4%
                      loop at AEntityFactory.cpp. 522-530                                               3.14e+08 36.4%
                        loop at AEntityFactory.cpp. 529-530                                             3.14e+08 36.3%
                          AEntityFactory::add_adjacency(unsigned long, unsigned lon                     3.05e+08 35.3%
                            AEntityFactory::get_adjacencies(unsigned long, std::vec                     2.47e+08 28.6%
                              inlined from stl_tree.h: 466                                              2.08e+08 24.1%
                                loop at stl_tree.h: 1370                                                2.08e+08 24.1%
                                  inlined from TypeSequenceManager.hpp: 27                              1.84e+08 21.3%
```

Figure 3.9: `hpcviewer`'s Calling Context view showing call paths overlayed with static program structure for MOAB (C++). Context sensitive metrics are attributed to both inlined code and loops.

3.4.1 MOAB

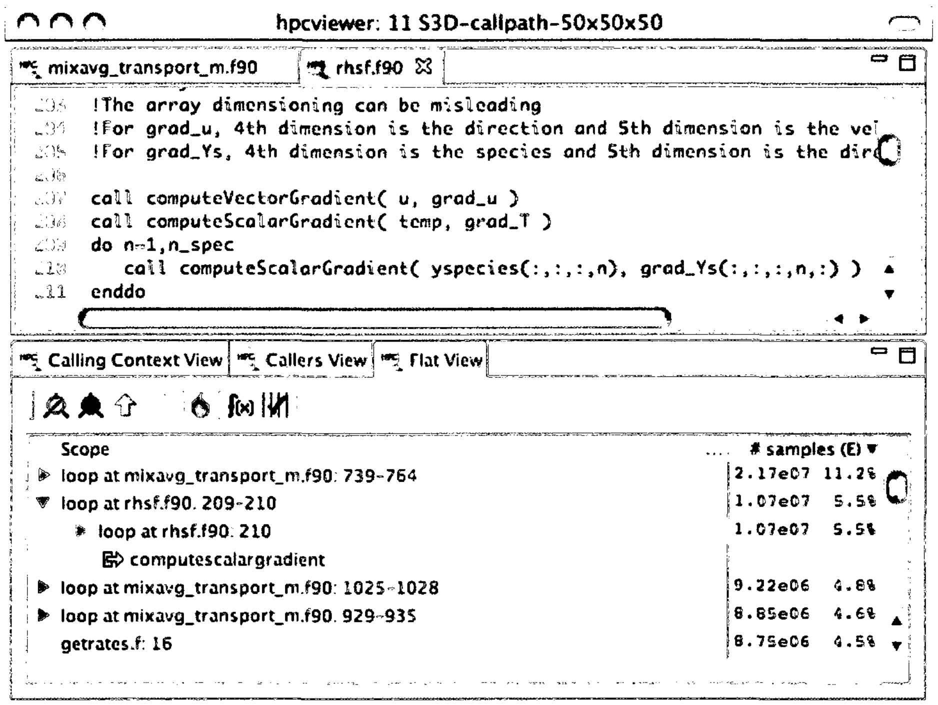
We first show the detailed attribution of performance data for MOAB, a C++ library for efficiently representing and evaluating mesh data [145]. MOAB implements the ITAPS iMesh interface [37], a uniform interface to scientific mesh data. We compiled MOAB on an AMD Opteron (Barcelona) based system using the Intel 10.1 compiler with `-O3`. (We could not use `-fast` because of a compiler error.) We profiled a serial execution the `mbperf` performance test using a 200 × 200 × 200 brick mesh and the array-based/bulk interface.

Figure 3.9 shows a calling context tree view of a call path profile of MOAB. The navigation pane (lower left sub-pane) shows a partial expansion of the calling context tree. The information presented in this pane is a fusion of hpcrun's dynamic and hpcstruct's static context information. The selected line in the navigation pane (at the bottom) corresponds to the highlight in the source pane (top sub-pane).

The navigation pane focuses on the hottest call path (automatically expanded by hpcviewer with respect to L1 data cache misses). A closer look reveals that the path contains *six* loops dynamically nested within inlined and non-inlined procedure activations. The root of the path begins prosaically with main $\rightarrow$ testB but then encounters an inlined procedure and loop from mbperf_iMesh.cpp. The inlined loop makes a (non-inlined) call to imesh_getentadj which descends through several layers of mesh iteration abstractions. Near the end of the hot call path, AEntityFactory::get_adjacencies contains an inlined code fragment from the C++ Standard Template Library (STL), which itself contains a loop over code inlined from the MOAB application (TypeSequenceManager.hpp). Closer inspection of the call path confirms that get_adjacencies calls an (inlined) procedure that calls the STL set::find function — which makes a call back to a user-supplied comparison functor in TypeSequenceManager.hpp. In this context, the comparison functor incurs 21.3% of all L1 data cache misses, suggesting that objects in the STL set should be allocated to exploit locality. Our tools are uniquely able to measure and attribute performance data at the source level with exquisite detail, even in the presence inlining.

3.4.2 S3D

The second application we discuss is S3D, a Fortran 90 code for high fidelity simulation of turbulent reacting flows [104]. We compiled S3D on a Cray XD1 (AMD Opteron 275) using Portland Group's 6.1.2 compiler with the -fast option.



```
203  !The array dimensioning can be misleading
204  !For grad_u, 4th dimension is the direction and 5th dimension is the ve
205  !For grad_Ys, 4th dimension is the species and 5th dimension is the dir
206
207  call computeVectorGradient( u, grad_u )
208  call computeScalarGradient( temp, grad_T )
209  do n=1,n_spec
210     call computeScalarGradient( yspecies(:,:,:,n), grad_Ys(:,:,:,n,:) )
211  enddo
```

```
Scope                                              # samples (E)
▶ loop at mixavg_transport_m.f90: 739-764          2.17e07  11.2%
▼ loop at rhsf.f90: 209-210                        1.07e07   5.5%
    ▶ loop at rhsf.f90: 210                        1.07e07   5.5%
        computescalargradient
▶ loop at mixavg_transport_m.f90: 1025-1028        9.22e06   4.8%
▶ loop at mixavg_transport_m.f90: 929-935          8.85e06   4.6%
  getrates.f: 16                                   8.75e06   4.5%
```

Figure 3.10: `hpcviewer`'s Flat view exposing loops for S3D (Fortran 90).

Figure 3.10 shows part of a loop-level Flat view for a call path profile of a single-core execution. The Flat 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 particular view was obtained by flattening away the procedures normally shown at the outermost level of the Flat view to show outer-level loops. This enables us to view the performance of all loop nests in the application as peers. We focus on the second loop on lines 209–210 of file `rhsf.90`. Notice that this loop contains a loop at line 210 that does not appear explicitly in the code. This loop consumes 5.5% of the total execution time. This is a compiler-generated loop for copying a non-contiguous 4-dimensional slice of array `grad_Ys` into a contiguous array temporary before passing it to `computeScalarGradient`. The ability to explicitly discover and attribute costs to such compiler-generated loops is a unique strength of our tools.

3.5 Related Work

There is a large body of prior work on call path profiling, but its focus has not been on using binary analysis to enable sampling-based measurement and attribution of performance metrics for fully optimized code. For this this reason we focus on comparing with contemporary tools with the most closely related capabilities for measurement and attribution.

To our knowledge, no other sampling-based profiler is capable of collecting full call path profiles for fully optimized code. Perhaps the closest conceptual work is a patent by Pierce that describes binary analysis for unwinding call stacks [116]. To unwind the call stack given an arbitrary sample point, Pierce proposes moving forward from that instruction to the first return point. During this process, each instruction is examined to determine how it affects the corresponding frame and return address location. One benefit of this approach is that because there is no necessity to know a function's begin point, it also applies to stripped binaries. In principle, this approach enables one to obtain call paths during execution of fully optimized code. However, it is difficult to provide a full comparison because the patent contains obviously expansive claims (e.g., Claims 14-16, 20-22) and lacks experimental results; additionally, we know of no publicly available implementation. One point of comparison is between approaches to binary analysis. Pierce's scan is not linear in the sense that it examines not only the region between the sample point and a return, but the callees within that region. In contrast, we perform a strictly linear scan through a function, computing function bounds as needed, and cache the results in a sparse data structure. We also demonstrate that we can use our approach to collect call path profiles for an average overhead of 1–2%.

Any tool based on libunwind [105], such as LoopSampler [106], requires frame pointers or unwind information. OProfile [85] and Sysprof [123], two well-known Linux system-wide call stack profilers, require frame pointers. AMD's CodeAnalyst [8] for Linux uses OProfile [85] to collect measurements and therefore inherits the latter's limitation. Since the x86-64 ABI does not require frame pointers, the restriction of these tools necessitates recompilation of any application and system library of interest. Apple's Shark [13], one of the nicer tools, also fails to correctly unwind optimized code. On a simple test, we observed it incorrectly unwinding calls from the `sinh` math library procedure.

Sampling-based call path profilers naturally fail to record a complete calling context tree. However, they also naturally highlight the most important paths, which comports well with performance analysis. Zhuang et al. develop 'bursty' call path profiling for Java [158] — a combination of sampling and adaptive, time-limited dynamic instrumentation — that more accurately approximates the complete CCT with an average overhead of 20%. For performance tuning, it is no bargain to pay such overhead to increase knowledge of infrequently executed paths.

The importance of correlating performance measurements with source code has been widely acknowledged. The task of correlation is easy with custom-generated compiler information [3, 150]. Unfortunately, this solution is impractical. Typically, open systems supply multiple compilers. Consequently, current sampling-based call path profilers trivially correlate dynamic data with source code using the binary's line map. In the presence of inlining and loop transformations, this approach results in confusing correlations that attribute costs of inlined code back to their source files rather than where they were incurred.

The major benefit of our approach is that `hpcrun` is minimally invasive, yet accurately attributes performance to both static and dynamic contexts, providing unique insight into program performance. No other sampling-based tool attributes the performance of *transformed loops* in the *full calling context* of *transformed routines* for *fully optimized binaries* to source code.

3.6 Discussion

We have designed methods of binary analysis for (1) minimally intrusive call path profiling of fully optimized code and (2) effective attribution and interpretation of performance measurements of fully optimized code. Our evaluation of `hpcrun` using the SPEC benchmarks on executables optimized by several different compilers shows that we can attribute costs incurred by fully optimized code to full calling context with low run-time overhead. The examples in Figure 3.10 highlight the unique contextual information we obtain by combining `hpcrun`'s dynamic call path information with `hpcstruct`'s static program structure. They show both how we attribute costs to inlined frames and loop nests and how this information yields insight into the performance of complex codes.

When compared with instrumentation-based techniques, our measurement and analysis methods have several advantages. *First*, (asynchronous) sampling-based call path profilers do not interfere with compiler optimization and introduce minimal distortion during profiling. On many operating systems, they can even be invoked on unmodified dynamically linked binaries. *Second*, using binary analysis to recover source code structure is uniquely complementary to sampling-based profiling. `hpcrun` samples the whole calling context in the presence of optimized libraries and even threads. `hpcstruct` recovers the source code structure, by using only minimal symbolic information, for any portion of the calling context — even without the source code itself. Using binary analysis to recover source code structure addresses the complexity of real systems in which source code for libraries is often missing. *Third*, binary analysis is an effective means of recovering the source code structure of fully optimized binaries. When source code is available, we have seen that `hpcstruct`'s object to source code structure mapping accurately correlates highly optimized binaries with procedures and loops. Among other things, it accounts for inlined routines, inlined loops, fused loops, and compiler generated loops. In effect, our binary analysis methods have enabled us to observe both what the compiler did and did *not* do to improve performance. We conclude that our binary analyses enable a unique combination of call path data and static source code structure; and this combination provides unique insight into the performance of modular applications that have been subjected to complex compiler transformations.

Both of our analyses have been motivated, in part, by a lack of compiler information. While we would welcome improved compiler support, it seems unlikely any will be forthcoming. Although compiler vendors have been sympathetic to our requests to fix or improve their symbolic information, they have been clear that their highest priority is highly efficient and correct code. Improving line maps or debugging information in binaries is at the bottom of their list of tasks. We have shown that accurate and rich contextual information can be obtained with only minimal compiler information and we believe that the utility of our results and the lack of a viable alternative justify our effort.

Chapter 4

Measurement & Attribution: Logical Call Path Profiling

4.1 Introduction

In recent years, the microprocessor industry has shifted its focus from increasing clock frequencies to delivering increasing numbers of processor cores. Following this general trend, cluster designs have shifted from single- or dual-processor nodes to multi-socket multicore processor nodes. For instance, nodes on the Department of Energy's 'leadership class' machines currently contain 4-12 cores and nodes on less-balanced large-scale systems will soon contain scores of threaded cores. Programming models for these machines have not shifted as decisively. Models that were designed for distributed-memory clusters are still being used on systems with shared-memory multicore processors, even though they may be less than optimal.

The shift to multicore processors plagues typical application developers as well. Without parallelism, no longer can a programmer expect an application to perform better on a next-generation processor. As a result, there is an urgent need for programming models and tools to support development of efficient multithreaded programs.

For a multicore programming model to become widely adopted, it must have four key properties. First, expressing parallelism should be simple. Second, parallel languages must be expressive enough to easily combine different parallel programming models. Although the (flat) data parallel model — in which the same computation is mapped across many data elements — has traditionally dominated high performance computing, many applications contain both data and task parallelism, and in irregular ways. Third, the programming model must make it possible to exploit parallel resources efficiently. Finally, the model must provide insurance against future architectural changes by transparently scaling to increasing core counts.

The Cilk language [58] was an early model that possessed these four properties. Cilk has proven very influential, spawning a commercial version of the language by Cilk Arts and serving as an exemplar for Intel's Threading Building Blocks [118], Microsoft's Concurrency Runtime, as well as ongoing research projects. In fact, a Cilk-like approach has even been applied to large-scale distributed-memory clusters [51]. These programming models raise the level of abstraction of parallel programming by partitioning the problem into two parts: the programmer is responsible for expressing the logical concurrency in a program and a run-time system is responsible for partitioning and mapping parallel work efficiently onto a pool of threads for execution.

Although programming models like Cilk substantially ease the difficulty of writing parallel programs, the developer is still responsible for identifying and resolving scaling bottlenecks in a poorly performing application. Consequently, there is an urgent need for performance tools that apply to the multithreaded programming models of choice. Unfortunately, the dynamic nature of Cilk-like run-time systems obscures application behavior and renders ineffective existing tools that measure and attribute performance directly to threads.

As described in Chapter 3, performance analysis of modern software requires associating costs with calling context. That chapter showed how to use asynchronous sampling to obtain very low overhead call path profiling of fully optimized applications. Of particular interest is providing this capability for high-level multithreaded programming models such as Cilk.

For Cilk-like programming models, using (asynchronous) sampling-based call path profiling to associate costs with the context in which they are incurred is not as simple as it sounds. At each sample event, a call path profiler must attribute the metric represented by the sample to the current execution context, which consists of the stack of procedure frames active when the event occurred. In contrast to native execution, Cilk's work-stealing scheduler dynamically partitions and maps work onto a thread pool, with the result that the stack of native procedure frames active within a thread represents only a suffix of the calling context. In effect, the work-stealing scheduler causes calling contexts to become separated in space and time as procedure frames migrate between threads as work is stolen. Since frames can be stolen, even the mapping between even an individual procedure frame and a thread may not be one to one. As a result, a standard call path profile of a Cilk program will show fragments of call paths mapped to each of the threads in the scheduler's thread pool, a result that is at best cumbersome and at worst incomprehensible. For effective performance analysis of multithreaded programming models with sophisticated run-time systems, it is important to bridge the gap between the abstractions of the user's program and their realization at run time.

To attribute metrics to the full source-level context of work-stealing computations, we develop a method for efficiently collecting *logical call path profiles.* Logical call path profiling is a generalization of call path profiling that enables one to measure and correlate execution behavior at different levels of abstraction. We show how to efficiently obtain a logical call path profile using a technique called *logical stack unwinding* and describe how to represent it using a *logical calling context tree.* Although we develop logical call path profiling to relate the execution of a multithreaded program by a work-stealing scheduler back to its source-level representation, it is applicable to any execution model for which native stack frames cannot serve as a proxy for a source-level call path.

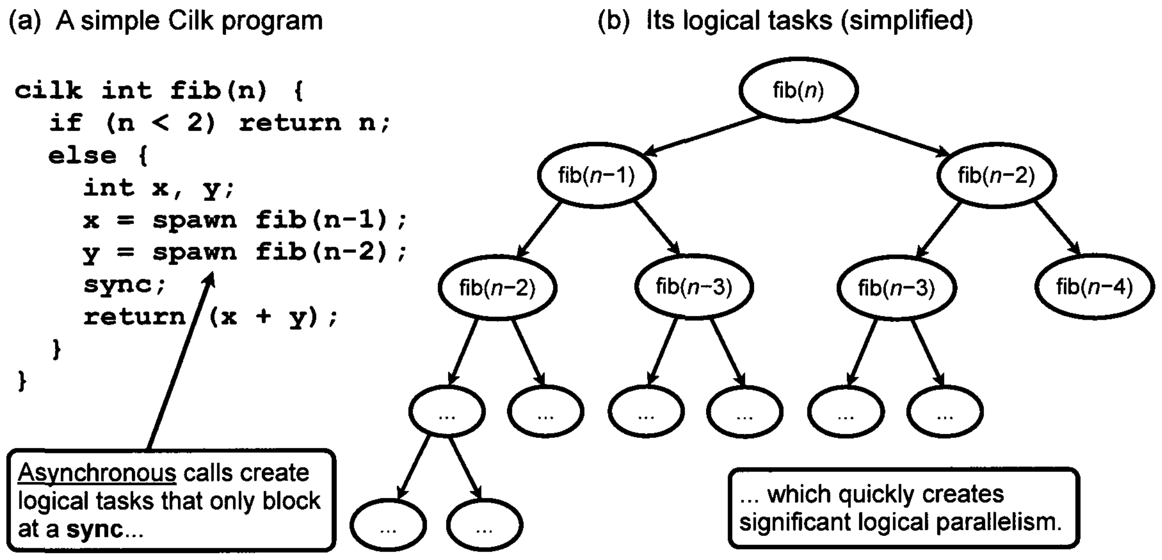
This chapter is organized as follows. Section 4.2 explains the specific challenge that work stealing raises for call path profiling. Section 4.3 defines a logical call path profile and Section 4.4 explains the process of obtaining one using logical stack unwinding; Appendix B presents some important implementation details. Then, Section 4.5 shows how to apply these ideas to Cilk in particular. Related work is discussed in Section 4.6. Finally, Section 4.7 discusses some high-level themes.

4.2 The Challenges of Work Stealing

Cilk is an extension of C and provides two keywords for expressing parallelism: `spawn` and `sync`. A `spawn` may be thought of as transforming a sequential (blocking) function call into an asynchronous (non-blocking) call. A `sync` blocks a function's execution until all of its spawned children have completed. Figure 4.1(a) shows an example of a Cilk program for computing the n^{th} Fibonacci number. The function computes `fib(n)` as the sum of `fib(n-1)` and `fib(n-2)`.[1] Since neither of the recursive calls to `fib` depends on the other, they may be executed in parallel, as indicated by the `spawn`. However, because the expression `(x + y)` depends upon the results of both of these calls, the `sync` ensures that both calls have completed before the addition commences.

Figure 4.1(b) graphically represents, in a simplified form, the logical parallelism in this computation. The `spawn`s and `sync`s form a tree of dependences where each interior (non-leaf) node directly depends on its two children. The tree is slightly

[1]This example is for illustration; there are much more efficient ways of computing `fib(n)`.


(a) A simple Cilk program
(b) Its logical tasks (simplified)
fib(n)
fib(n−1)
fib(n−2)
fib(n−2)
fib(n−3)
fib(n−3)
fib(n−4)
...
Asynchronous calls create logical tasks that only block at a sync...
... which quickly creates significant logical parallelism.

```
cilk int fib(n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
```

Figure 4.1: Example of Cilk's simplicity and expressiveness. The simple program of (a) uses asynchronous calls (spawn) to express (b) a complex pattern of parallelism.

unbalanced to reflect the fact that there is more work in each node's left child than on its right.

The challenge for the Cilk run time is to map logically independent calls onto compute cores in an efficient way. Each asynchronous call may be thought of as a lightweight thread, commonly called a *task*. Cilk's approach is to combine *lazy* task creation with a work-stealing scheduler. The Cilk run time creates a pool of OS-level worker threads, one per available core, to execute the program. The first worker thread begins execution of the program (the first task). If there are no other worker threads in the pool, execution of the program continues sequentially, without any additional task creation. Whenever the thread pool contains an idle worker, that worker attempts to steal a task from a working thread. Figure 4.2 shows the beginning of a possible parallel execution of the Fibonacci program of Figure 4.1. Execution begins by assigning the whole computation to worker thread 1 (red). This worker starts elaborating the call tree in a depth-first order and continues down the leftmost branch, as would a serial execution. Worker thread 2 (green), currently idle, steals the

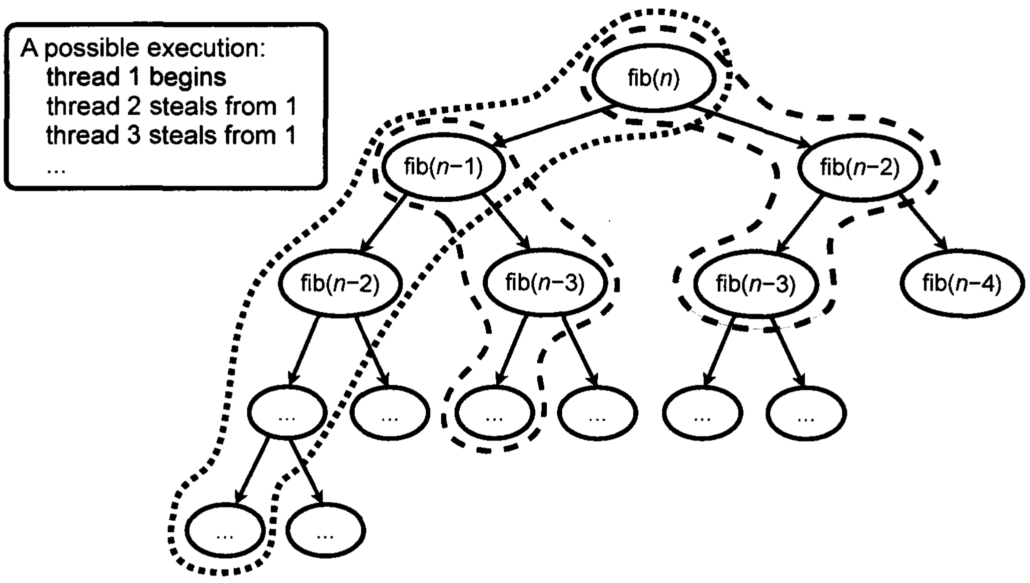


Figure 4.2: Scheduling work via work stealing. Cilk's scheduler separates source-level calling contexts in space and time.

continuation associated with fib(n), which promptly spawns a second asynchronous call to compute fib($n-2$). A third idling worker thread (blue) now has two threads from which to steal. Suppose that this thread randomly chooses to steal from thread 1 and then selects the next piece of available work, the continuation associated with fib($n-1$), which then spawns a call to fib($n-3$).

The Cilk model has many attractions. For example, although a **spawn** identifies an independent task, the overhead of assigning this work to a separate thread is only realized *when necessary*, i.e., when a worker thread is idle. Moreover, as long as worker threads execute enough **spawns**, it is easy to see that work stealing naturally achieves very good load balance. Both of these facts means that the same Cilk program can execute efficiently on one or several cores.

Unfortunately, Cilk's work-stealing scheduler renders useless even sophisticated techniques for gathering calling context. To appreciate the difficulty, consider how state-of-the-art call path profilers [67] — tools that attribute metrics to calling context

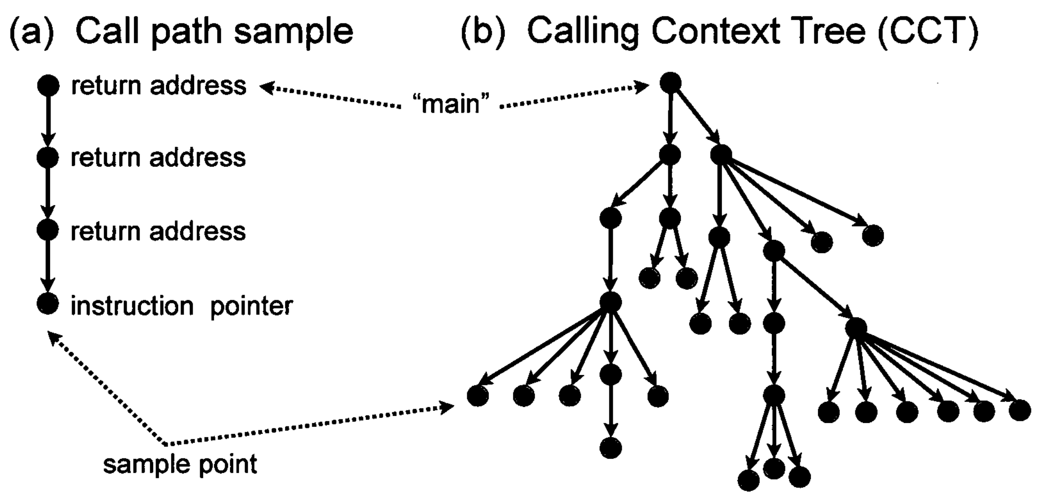


Figure 4.3: An asynchronous-sampling-based call path profiler (a) collects a call path for each sample point; and (b) several call paths form a calling context tree. (Duplicated from Figure 2.2.)

— perform their job. To achieve low overhead, (asynchronous) sampling-based call path profilers use asynchronous sampling (rather than instrumentation) to attribute costs of a program execution to the calling contexts in which they occur. To sample a program, a profiler initializes a timer or hardware counter that generates a signal when it expires or overflows. For each sampling signal, a call path profiler gathers the profiled application's calling context using stack unwinding. This results in a call path sample (Figure 4.3(a)), represented as a list of instruction pointers, with the leaf being the sample point. A collection of samples naturally forms a calling context tree (Figure 4.3(b)), where the program's entry point is the root of the tree. The key advantage of sampling over instrumentation is that the overhead of the former is proportional to the sampling frequency and not the call frequency. Moreover, sampling naturally elides unimportant data since (given a reasonable sampling rate) if an area of the application receives no samples, *then its cost is negligible.*

Cilk's work-stealing run time confuses call path profilers. Figure 4.4 shows what would happen if thread 3 (blue) from Figure 4.2 receives a sample. Because thread

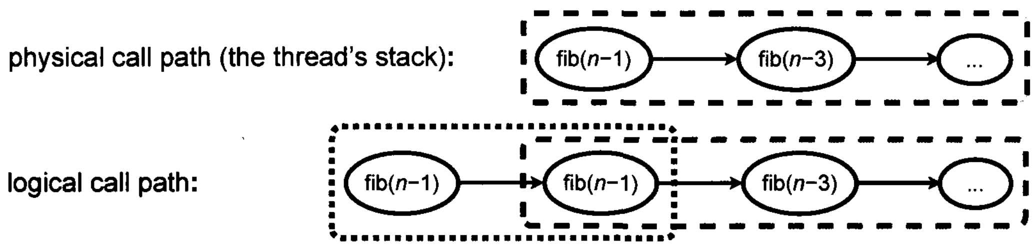


Figure 4.4: A case for logical call path profiling. Suppose that thread 3 (blue) from the example in Figure 4.2 receives a sample. Because that thread began its execution with a steal, the rest of its context (red) is separated in space and time. Logical call path profiling attributes metrics to their full logical context.

3 began its execution with a steal, the stack of native procedure frames within that thread represents only a suffix of the full calling context. In fact, the rest of thread 3's context is separated in both space and time: space, because thread 1 contains its parent context; time, because thread 1 continues executing rather than blocking and waiting for thread 3 to complete the asynchronous call. Over the course of an execution, call paths become even more fragmented as procedure frames migrate between threads during steals. As a result, a standard call path profile of a Cilk program yields a result that is at best cumbersome and at worst incomprehensible. For effective performance analysis, it is important to bridge the gap between source-level abstractions and their realization at run time by attributing costs to their full logical calling context. We call this logical call path profiling.

4.3 Logical Call Path Profiles

For languages based on work stealing, mapping measurements during execution back to a source program requires reassembling source-level contexts, which have been fragmented during execution. This and the next section (Section 4.4) extend the notion of call path profiling by defining *logical call paths* and describing how to generally and efficiently obtain logical call path profiles using a *logical calling context* *tree.* Logical call path profiling applies to both parallel and serial applications. In Section 4.5, we describe how this technique forms an essential building block for measurement and analysis of multithreaded Cilk program executions by a work-stealing scheduler.

4.3.1 Logical Call Paths

A sampling-based call path profiler obtains a call path by unwinding the call stack at a sample point to obtain a list of active procedure instances, or frames. Such a call path may not correspond directly to a source-level calling context. We introduce the notion of *logical* call paths to bridge this gap. We obtain *logical* call paths by *logically* unwinding the call stack. To support a precise discussion of this concept, we introduce and define the following terminology.

A *bichord* is a pair $\langle P_i, L_i \rangle$ consisting of a p-*chord* P_i and a l-*chord* L_i where each p-chord (or l-chord) is is a sequence of p-notes (l-notes), e.g.:

$$\langle P_i, L_i \rangle = \langle (p_{i,1}, \ldots, p_{i,m_1}), (l_{i,1}, \ldots, l_{i,m_2}) \rangle$$

A note represents a frame; a chord a grouping of frames; and a bichord the association of a group of physical stack frames (P_i) with a group of logical (L_i) stack frames. Logical frames correspond to a source-level calling context; physical frames correspond to an implementation-level realization of that view. The p-notes $P_i = (p_{1,1}, \ldots, p_{1,m_1})$ that form p-chord P_i represent the bichord's physical call path fragment, while the l-notes form the logical call path fragment. We say that the length $|P_i|$ of p-chord P_i, is the number of p-notes contained therein, i.e., m_1 in the above example; similarly, $|L_i| = m_2$.

A *logical call path* is a sequence of bichords

$$\langle\langle P_1, L_1\rangle, \langle P_2, L_2\rangle, \ldots, \langle P_n, L_n\rangle\rangle$$

where $\langle P_1, L_1\rangle$ is the program's entry point and where bichord $\langle P_n, L_n\rangle$ represents the innermost set of frames. It is natural to speak of the p-*chord projection* for the logical call path as

$$\langle P_1, \ldots, P_n\rangle$$

and the p-*note projection* as

$$\langle(p_{1,1}, \ldots, p_{1,m_1}), \ldots, (p_{n,1}, \ldots, p_{n,m_n})\rangle$$

where $p_{1,1}$ represents the physical program entry point and the projection represents the physical call path from the entry point to the sample point. *Logical projections* are analogous.

To provide intuition for a discussion of bichord forms, it is useful to consider a concrete representation. We represent a p-note projection as a list of instruction pointers, one for each procedure frame active at the time a sample event occurs. The first instruction pointer of the unwind (p_{n,m_n}) is the program counter location at which the sample event occurred. The rest of the list contains the return address for each of the active procedure frames. Similarly, each l-note in a logical call path contains an opaque logical instruction pointer that represents the logical context.

Defining a logical call path to consist of a sequence of bichords formed of notes enables us to preserve interesting relationships between the physical and logical call path. To formalize these relationships, we first observe that a logical call path's p-note projection should always have a non-zero length because the physical stack is never empty. Moreover, intuitively, every l-chord must be associated with at least one p-note. This implies that no bichord should have a zero length p-chord. Equivalently, we observe that a p-note projection should not have 'gaps,' i.e., a machine cannot return to a 'virtual' logical frame — an l-note without an associated p-note — and then return back to a physical frame. From this starting point, we consider the possible relationships, or *associations*, between the lengths of a bichords's p-chord and l-chord. Given bichord $B_i = \langle P_i, L_i \rangle$, there are several possible associations between $|P_i|$ and $|L_i|$ that we describe with a member from the set $\{0, 1, \mathbf{M}\} \times \{ 0, 1, \mathbf{M}\}$, where $\mathbf{M}$ (a mnemonic for multi or many) represents any natural number $m \geq 2$. We are interested in the following four categories accounting for five of the possible association types:

1. $1 \leftrightarrow 1$. One p-note directly corresponds to one l-note — the typical case for C or Fortran code where a physical procedure frame corresponds to a logical procedure frame.

2. $1 \leftrightarrow 0$ and $\mathbf{M} \leftrightarrow 0$. A p-chord corresponds to an *empty* l-chord. This situation typically arises when run-time support code is executed. For example, a sample event that interrupts the run-time system's scheduler may find several physical frames that correspond to no logical procedure frame.

3. $\mathbf{M} \leftrightarrow 1$. This association often describes the run-time system implementing a high-level user routine. For example, a Python interpreter may require a chain of procedure calls (several p-notes) to implement a user-level call to sort a list.

4. $1 \leftrightarrow \mathbf{M}$. At first sight, this association may seem esoteric. However, it has important applications. It directly corresponds to using Cilk's scheduling loop as a proxy for walking the cactus stack of parent procedures that are stored in the heap and have no physical presence on the stack. As another example, a

Java compiler could form one physical procedure from a 'hot' chain of source-level procedures.

Three observations are apropos. First, as previously discussed, associations $0 \leftrightarrow \{0, 1, \mathbf{M}\}$ are excluded meaning that the length of a p-chord is always non-zero. Second and in contrast, association (2) implies that it is possible to have a zero-length l-chord. The final omitted association, $\mathbf{M} \leftrightarrow \mathbf{M}$, can always be represented as some combination of categories (1–4) above.

We now concisely define a logical call path as a sequence of bichords

$$\langle\langle P_1, L_1\rangle, \langle P_2, L_2\rangle, \ldots, \langle P_n, L_n\rangle\rangle$$

where $n \geq 1$ and $\forall i[|P_i| \geq 1]$, but where it is possible that $|L_i| = 0$ for any i.

4.3.2 Representing Logical Call Path Profiles

At run time, we wish to efficiently obtain and represent a logical call path profile, i.e., a collection of logical call paths annotated with sample counts with the time dimension removed. Our approach is to form a logical calling context tree — an extension of a *calling context tree* (CCT) [9] — that associates metric counts with logical call paths.

Weighted logical calling context trees

We first define a very simple logical CCT. Given a logical unwind

$$\langle\langle P_n, L_n\rangle, \langle P_{n-1}, L_{n-1}\rangle, \ldots, \langle P_1, L_1\rangle\rangle$$

where $\langle P_n, L_n \rangle$ is a sample point, the straightforward extension of a CCT ensures that the path

$$\langle \langle P_1, L_1 \rangle, \langle P_2, L_2 \rangle, \ldots, \langle P_n, L_n \rangle \rangle$$

exists within the tree, where $\langle P_1, L_1 \rangle$ is the root of the tree and where $\langle P_n, L_n \rangle$ is a leaf node. Metrics such as sample counts are associated with each leaf node (sample point); in this example metrics at $\langle P_n, L_n \rangle$ are incremented.

We define the *physical projection* of a logical CCT to be the CCT formed by taking the p-chord projection of each call path in the logical CCT. The *logical projection* of a logical CCT is defined analogously.

Efficiently representing logical calling context trees

While this logical CCT representation is simple, treating bichords as atomic units can result in considerable space inefficiency. To reduce memory effects, we wish to share notes without losing any information represented in the logical CCT. Appendix B describes when sharing is possible and develops a more efficient and practical implementation.

4.4 Obtaining Logical Call Path Profiles

Given the definition of a logical call path and the representation of a call path profile using a logical calling context tree, we now turn our attention to obtaining a logical call path profile. To provide low controllable measurement overhead, we use asynchronous sampling and form the logical calling context tree by collecting and inserting logical call paths on demand for each sample. 'Physical' call path profilers use stack unwinding to collect the call path. Since the physical calling context alone is insufficient for obtaining the logical call path, we develop the more general notion of *logical stack unwinding* to collect the logical call path.

4.4.1 Logical Stack Unwinding

Consider a contrived example where a Python driver calls a Java routine that calls a Cilk solver. Though unusual, this example shows that each bichord in a logical call path could potentially derive from a different run-time system. Because run-time systems use the system stack in their implementation, this suggests that the actual process of logical unwinding should be controlled by the physical stack. This is natural because although the physical call stack may represent the composition of calls from many different languages, it conforms to a known ABI. In addition, using a physical unwind naturally corresponds to our requirement that a p-note projection not have 'gaps', i.e., there is at least one representative stack frame for each l-chord in the logical unwind. However, since a physical stack unwinder alone cannot determine either the association of the bichord or the length of the p-chord or the content of the l-chord, some sort of additional information must be available to construct the bichord. This information can be obtained using a language-specific plug-in or *agent* to assist a 'physical' stack unwinder. Each agent would understand its corresponding language implementation well enough to determine the particulars of reconstructing an l-chord given the *start* of a p-chord. It is important to emphasize a p-chord's *start* because assistance from the agent will in general be necessary to determine the p-chord's length, e.g., 1 versus $\mathbf{M}$.

There must be some way of selecting which agent to use at any point in the logical unwind. In the example above, one must know when to use the Cilk, Java and Python agents, respectively, to obtain the relevant bichords. Observe that at any point in the execution, the return address instruction pointer located in the stack frame should map to at most one run-time system and therefore one agent. Consequently, the frame's return address serves a proxy for the specific agent that should be consulted to assist formation of the bichord. During a program's execution, the mapping of code segments within the address space (the load map) can typically be determined by interrogating the operating system.

4.4.2 Thread Creation Contexts

Often it is useful to know the context in which a thread was created. The *creation context* of a thread is defined as the calling context at the time the thread was created. For example, consider a solver using fork-join parallelism where a pool of Pthreads [32] is created using several calls to `pthread_create`. It is desirable to capture the calling context of the `pthread_create` so that the Pthread can be rooted within the context of the solver. The thread creation context may be captured and maintained as an extension to the thread's physical stack.

4.4.3 An API for Logical Unwinding

We have designed and implemented a general API for obtaining logical unwinds given language specific agents. Technically, there are two sub-APIs, one for collecting logical unwinds (using agents) and one describing the interface to which language-specific agents must conform and the assumptions they may make.

The API for logical unwinding is designed to place as much burden as possible on the non-agent library routines so that agent implementation is as easy as possible. For example, an agent is not required to perform any look-ahead to determine the length of an l-chord. Although this information could be used by the logical unwinder (Algorithm 4.1) for allocating storage, we determined that it was more desirable to complicate the code for the unwinder than to complicate each agent's implementation.

Consequently, the logical unwinder ensures that enough buffer space is always available to store a bichord. As another example, the agent interface sub-API promises a small amount of functionality to ease agent implementation, such as a means to inspect the address space and a safe memory allocator (`malloc` may not be safe).

The logical unwinding API is divided into a two-level hierarchy corresponding to the division between bichords and notes. In particular, the top level addresses finding the bichords within a logical unwind while the other level targets finding the notes of a chord. An outline of of the backtrace routine is shown in Algorithm 4.1. Each level adopts semantics similar to libunwind [105]. This means that to find each bichord in the logical unwind $\langle\langle P_n, L_n\rangle, \langle P_{n-1}, L_{n-1}\rangle, \ldots, \langle P_1, L_1\rangle\rangle$,[2] n successive calls to step-bichord are required along with an additional call that returns a special value to indicate the unwind is completed. The advantage of these semantics is that they help ensure agents do not have to perform contextual look ahead. For example, to examine all l-notes within the l-chord $(l_{i,1}, \ldots, l_{i,m})$, $m+1$ calls are issued to step-lnote. This means that the agent need not know that $l_{i,1}$ is the last l-note in the l-chord unwind until the $(m+1)^{\text{th}}$ call to step-lnote. This fact is particularly useful for an agent to a multithreaded run-time system because thread-specific state need not be maintained within the agent. Rather, all state for the unwind can be maintained by a fixed-sized thread-specific cursor allocated by the logical unwinder.

As discussed previously, logical unwinding is driven by a stack unwind. On each call to step-bichord, the library determines if a valid physical stack frame exists. If so, it extracts the return address instruction pointer and determines if it maps to any agent. If it does, that particular agent is used to complete the discovery of the bichord. Otherwise, the 'identity' agent is used to create a $1 \leftrightarrow 1$ bichord representing native code.

[2]A *logical unwind* is simply the reverse of a logical call path.

Algorithm 4.1: logical-backtrace: Perform a logical unwind.

```
let $c$ be the unwind cursor, initialized with the machine context and
    language-specific logical unwind agents
while step-bichord(&$c$) $\neq$ EndUnwind do
    let $a$ be the bichord's association (from $c$)
    while step-pnote(&$c$) $\neq$ EndChord do
        Record $p$-note (instruction pointer from $c$)
    while step-lnote(&$c$) $\neq$ EndChord do
        Record $l$-note (logical instruction pointer from $c$)
    Form bichord from $a$ and the lists of $p$-notes and $l$-notes
```

Observe that the asymmetry between p-chords and l-chords plays a critical role in the unwind process. For a p-chord P_i of length m_i, the $(m_i + 1)^{\text{th}}$ call to step-pnote both completes enumeration of P_i's p-notes and discovers the next p-chord. For example, consider a section of the physical projection representing p-chords P_i and P_{i+1}:

$$(\ldots, p_{i,m_i})(p_{i+1,1}, \ldots)$$

While iterating over the p-notes in p-chord P_i, we first issue m_i calls to step-pnote. On the $(m_i+1)^{\text{th}}$ call, the agent discovers that there are no more p-notes in P_i, but only because it has found p-note $p_{i+1,1}$, the beginning of p-chord P_{i+1}. This means that the p-note portion of the cursor is pointing to the beginning of P_{i+1} *before* the cursor has stepped to P_{i+1}. This 'peeking' behavior is important because we must know the initial portion of P_{i+1} in order to know which agent to assign the responsibility of the next bichord. In contrast, step-lnote need not 'peek' ahead in to the next l-chord. Indeed, it should not because the next l-chord may be handled by a different agent and may have length 0.

4.5 Logical Call Path Profiles of Cilk Executions

To attribute metrics to logical calling contexts, we modified HPCToolkit to collect logical call path profiles for Cilk. We added capability to the `hpcrun` profiler to bridge the gap between Cilk's source-level calling contexts and their realization at run time within Cilk. In particular, we implemented the logical unwind API described in Section 4.4.3 and developed a Cilk-specific agent. To attribute source code static program structure and dynamic logical contexts, we extended the `hpcprof` tool to correctly interpret the measurements. Finally, in Section 5.3, we show how we present logical call path profiles in our interactive viewer.

The design of the Cilk agent illustrates several important points. Although discussing this agent necessarily involves details about the Cilk implementation, it is important to note that the API remains language independent.

To understand the Cilk agent, it is necessary to review some high-level details about the Cilk-5 implementation. For each source Cilk routine, the Cilk compiler generates two clones, a 'fast' and 'slow' version. The fast clone (which is similar to the corresponding 'C-elision' [58]) is executed in the common case. Importantly, whenever a procedure is spawned, the fast version is executed. The slow clone is executed only when parallel semantics are necessary such as when a procedure is stolen.

Each worker thread maintains a deque (stored in the heap) of ready procedure instances, which together form a Cactus stack, i.e., a tree where the root corresponds to the bottom (outermost frame) of the stack. Local work is pushed and popped from the tail of the deque (top or inner frames) while thieves steal from the head (bottom or outer frames). Execution proceeds on the thread's stack even though a 'shadow' continuation is maintained on the deque. Whenever a thief steals a procedure's continuation, it resumes it using the slow version of that procedure. Since frames may only be stolen from the deque's head (bottom of cactus stack), this implies that the descendants of a fast procedure may only be fast procedures themselves.

We may infer the following invariants about the frames on a worker's stack (in top-down order):

A. There may be i frames corresponding to Cilk run-time routines (e.g., creation of continuation information) or source-level C routines. Cilk run-time routines correspond to a bichord with association $1 \leftrightarrow 0$ (since they are not part of the logical call path), while source-level C routines correspond to an association of $1 \leftrightarrow 1$.

B. There may be j frames corresponding to Cilk fast frames. Since the fast clone of a Cilk routine directly corresponds to a physical frame and a logical frame, the pair corresponds to a bichord with association $1 \leftrightarrow 1$.

C. There is always at least one frame corresponding to the Cilk scheduler.

These segments may not be interchanged.

The exact interpretation of segment C depends upon whether there are additional ancestor frames in the Cactus stack. That is, when a worker steals any procedure other than 'main,' that procedure's logical context is represented as a chain of ancestor frames within the Cactus stack. In this case, the scheduler frame has association $1 \leftrightarrow \mathbf{M}$. Otherwise, if the innermost frame in segment B corresponds to 'main,' which has no logical calling context, the scheduler frame has association $1 \leftrightarrow 0$.

Figure 4.5 shows an example of the case where the scheduler frame has association $1 \leftrightarrow \mathbf{M}$. The logical call path in the figure has five pairs, where the outermost frame is at the left. For each pair, source-level frames are on the bottom (the green nodes) and native frames (red and blue nodes) are on the top. Thus, the top frames represent

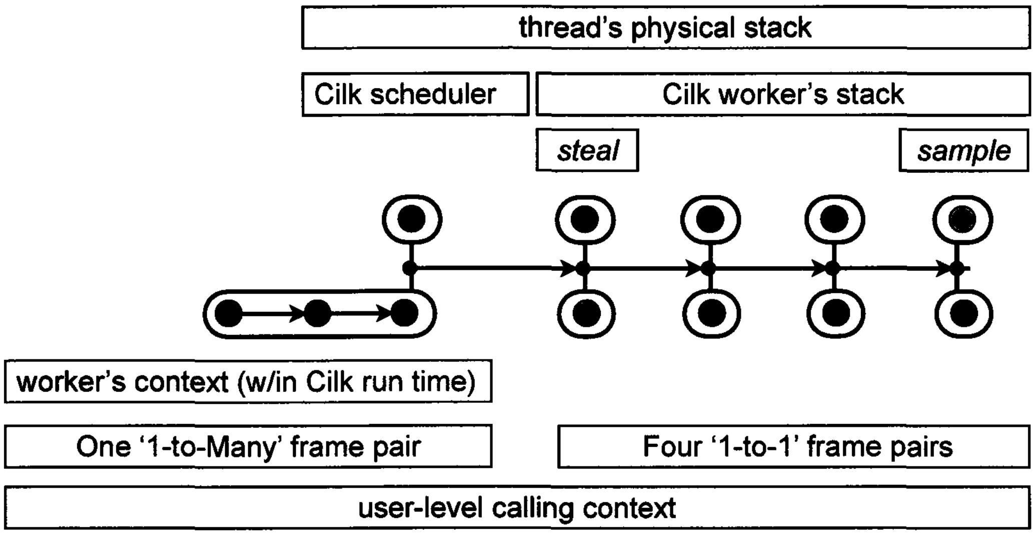

thread's physical stack
Cilk scheduler
Cilk worker's stack
steal
sample
worker's context (w/in Cilk run time)
One '1-to-Many' frame pair
Four '1-to-1' frame pairs
user-level calling context

Figure 4.5: The logical call path for a typical Cilk worker thread

the native frames of a worker thread's stack. The outermost native frame represents Cilk's scheduler loop and the next native frame is a steal point. Because of the steal point, the outermost native frame corresponds to several source-level frames that represent the context of the steal. In contrast, each native frame after the steal point corresponds to only one source-level frame.

4.6 Related Work

Several tools for obtaining call path profiles have been developed, they collect only physical call path profile *projections* [44, 60, 107, 127, 130] or logical call path profile *projections*, such as for Java [23, 156, 158]. Furthermore, we know of no prior work for collecting even logical call path profile projections for a multithreaded programming language based on lightweight tasks.

In parallel but independent work, Itzkowitz et al. describe an OpenMP API that enables a statistical call path profiler to correlate source-level call paths with run-time metrics about whether a thread is working or waiting [79]. Our work is more general in the sense that we define logical call path profiles, explain how they can be efficiently represented, and describe a general API for obtaining them.

Cantrill et al. [33] point to interesting stack unwinding possibilities using the DTrace systems tool. DTrace dynamically instruments a large number of system events, including function entry or exit points. With a DTrace-enabled kernel, it is possible to obtain stack unwinds that bridge the the user/kernel boundary. Cantrill et al. also cite future work that includes obtaining "a user-level stack trace that contains both Java and C/C++ stack frames."

4.7 Discussion

Because of the growing influence of languages with dynamically managed parallelism, effective tools for quantifying and for pinpointing performance bottlenecks in multithreaded applications are absolutely essential. No tool can be effective without attributing performance metrics to source-level contexts. Consequently, there is a clear need to use logical call path profiling as a foundation for gathering low-overhead contextual measurements that highlight inefficient computation. In Chapter 5 we will use logical call path profiling to attribute work, parallel idleness and parallel overhead to the *logical* calling contexts of a Cilk application. The results enable one to quickly obtain unique insight into the application's performance.

Logical profiling is a powerful tool for understanding performance. An especially useful technique is to combine logical call path profiling with *differential profiling*, where corresponding sections of different execution profiles are mathematically combined [92]. Differencing two profiles that are expected to be similar is especially powerful. For example, a logical call path profile could be an effective way to compare two different implementations of Cilk executing the same program.

Besides high-level parallel languages, logical unwinding applies to serial codes developed in languages that rely on managed run-time systems such as Java and Python. The concept could be applied to multi-lingual applications such as those built using common component architectures [16] and inter-language binding systems such as Babel [80].

Chapter 5

Analysis of Multithreaded Executions: Work Stealing

5.1 Introduction

Over the last several years, power dissipation has become a substantial problem for microprocessor architectures as clock frequencies have increased [103]. As a result, the microprocessor industry has shifted its focus from increasing clock frequencies to delivering increasing numbers of processor cores. For software to benefit from increases in core counts as new generations of microprocessors emerge, it must exploit threaded parallelism. As a result, there is an urgent need for programming models and tools to support development of efficient multithreaded programs.

As Chapter 4 discusses, Cilk [58] was developed to simplify the development of multithreaded programs. In particular, Cilk pioneered a sophisticated and influential work-stealing scheduler that is provably efficient assuming the availability of sufficient concurrency. Nevertheless, while Cilk eases the burden of writing parallel programs, it does not necessarily make it easier to write programs that scale well with the number of available cores.

To help developers to rapidly understand why their programs do not perform as intended, it is necessary to have effective performance tools. Performance tools typically report how resources, such as time, are *consumed* rather than *wasted.* For parallel programs, it is typically most important to know where time is wasted as a result of an ineffective parallelization. To enable an average developer to quickly assess the quality of the parallelization in a multithreaded application, tools should pinpoint program regions where the parallelization is inefficient and quantify their impact on performance. Two aspects of a parallelization in particular are important for efficiency: whether there is adequate parallelism in the program to keep all of the processor cores busy, and whether the parallelism is sufficiently coarse-grain so that the cost of managing the parallelism does not become significant with respect to the cost of the parallel work.

In this chapter, we develop two novel techniques for assessing both of these aspects of parallel efficiency.

- A technique for measuring and attributing *parallel idleness* — when threads are idling or blocked and unable to perform useful work. This technique primarily applies to work-stealing-based languages such as Cilk [58] and Threading Building Blocks [118]. It relies on minor modifications to the run-time systems of multithreaded programming models.

- A technique for measuring and attributing *parallel overhead* — when a thread is performing miscellaneous work other than executing the user's computation. This technique can be applied to both library-based programming models such as Pthreads [32] and Threading Building Blocks, as well as compiler-based programming models such as Cilk and OpenMP. By employing a combination of compiler support and post-mortem analysis, we incur no measurement cost beyond normal profiling to glean this information.

We pair these techniques with logical call path profiling (Chapter 4) to effectively measure, attribute, and analyze the performance of multithreaded programs. Logical call path profiles are the key for mapping measurements of work, idleness and overhead back to the source-level abstractions in high-level multithreaded parallel programming models. Our *idleness* and *overhead* metrics enable one to pinpoint areas of an application where concurrency should be increased (to reduce idleness), decreased (to reduce overhead), or where the present parallelization is hopeless (where idleness and overhead are both high). To show the utility of these techniques, we describe their implementations within Cilk. We then use the HPCToolkit suite of performance tools to attribute work, idleness, and overhead to Cilk source code lines in their full source-level calling contexts.

This chapter is organized as follows. First, Section 5.2 describes parallel idleness and overhead. Section 5.3 describes the application of these ideas to Cilk. Finally, Section 5.4 discusses related work and Section 5.5 discusses the chapter's high-level themes.

5.2 Pinpointing Parallel Bottlenecks

We describe two novel measurement and analysis techniques that enable an average developer to quickly determine whether a multithreaded application is effectively parallelized. If the application is not effectively parallelized, our techniques direct one's attention to areas of the program that need improvement.

5.2.1 Quantifying Insufficient Parallelism

To quantify insufficient parallelism in work-stealing-based applications, we have developed a method to efficiently and directly measure *parallel idleness*, i.e., when threads are idle and unable to perform useful work. Our goal is to compute the metrics 'work' and 'idleness' where:

$$\text{effort} = \text{work} + \text{idleness}$$

Assume we are using a (asynchronous) sampling-based logical call path profiler to profile a Cilk application. Further assume that our asynchronous sample source is a time-based counter such as the wall clock or a hardware cycle counter. Recall that Cilk's work-stealing scheduler creates one worker thread per core. When a sample event occurs during profiling, each thread receives an asynchronous signal. Worker threads are either working or idle. If a worker thread is idle, then it is spinning within a scheduler loop waiting for another thread to create a stealable task. A logical call path profiler attributes samples based on a 'first party' basis, i.e., based on what a thread itself is doing. This means that working threads accumulate samples where they work, but idle threads accumulate samples in the scheduler loop.

While this method quantifies parallel idleness — samples received within the scheduler clearly reflect idleness — the results are not *actionable* because they do not pinpoint the *cause* of idleness. To pinpoint the cause of idleness, there must be a way to correlate a thread's idleness with those threads that are responsible for its idleness. To establish this correlation, threads must have some 'third party' knowledge about other threads, such as which threads are responsible for another thread's idleness. Recall that within the Cilk model, a thread is idle precisely because other threads have no extra tasks available to steal. Therefore, when a thread is idle, the current working threads are in an important sense culpable for not being sufficiently parallelized. Consequently, we want to change how samples are attributed for idle threads to form a metric that blames *working* threads for not spawning enough tasks to keep all workers busy.

We can accomplish our goal by doing two things. First we make a slight adjustment to the Cilk run-time to always maintain W and I, the number of working and idle threads, respectively. This can be done by maintaining a node-wide counter representing W. When a thread begins a task, it atomically increments W. When that thread completes its current task it atomically decrements W to indicate that it is no longer actively working. Thus, $I = n - W$, where n is the number of worker threads.

Second, we slightly modify our sampling strategy. If a sample event occurs in a thread that is not working, we ignore it. When a sample event occurs in a thread that is actively working, the thread attributes one sample to the work metric for its sample context. It then obtains W and I and attributes a fractional sample I/W to the idleness metric for the sample context. Even though the thread itself is not idle, it is critical to understand what work it is performing when other threads are idle. Our strategy charges each working thread its proportional responsibility for not keeping the idle processors busy at that moment at that point in the program.

As an example, consider taking a sample of a Cilk execution where five threads are working and three threads are idle. According to our scheme, each working thread records one sample of work in its work metric, and 3/5 sample of idleness in its idleness metric. The three idle threads ignore their samples. The total amount of work and idleness charged for sampling each thread is 5 and 3, respectively.

After measurement is completed, idleness can be computed for each program context. Since samples are accumulated during measurement, the idleness value for a given thread and context is $\sum I_i/W_i$ over all samples i for that context. It is often useful to express this idleness metric as a percentage of the total idleness for the program. Total idleness may be computed post-mortem by summing idleness metric over all threads and contexts in the program. The idleness value may be converted to a time unit by multiplying by the sample period. One can also divide the idleness for each context by the application's *total effort* — the sum of work and idleness everywhere across all threads — to understand the fraction of total effort that was wasted in each context.

The measurement overhead of our strategy is expected to be low for two reasons. First, logical call path profiling has very low overhead when using sampling frequencies of hundreds to thousands of samples/second; in addition, the sampling rate is controllable by adjusting the sampling frequency. Second, at least on small-scale nodes, a work-stealing scheduler is unlikely to cause contention for atomically modifying the global counter W. To see this, observe that the global counter is modified only when a thread steals. Thus, contention can only occur when multiple threads enter/exit the scheduler loop simultaneously. We have empirically verified that contention is very low on (at least) up to 16 cores. For large-scale shared-memory machines or for applications where stealing is very frequent, it may be necessary to adapt the distributed blame shifting strategy we present in Chapter 6.

5.2.2 Quantifying Parallelization Overhead

Now that we have quantified parallel idleness, we wish to refine the work metric in the equation

$$\mathsf{effort} = \mathsf{work} + \mathsf{idleness}$$

to distinguish useful work from parallel overhead:

$$\mathsf{work} = \mathsf{useful\text{-}work} + \mathsf{overhead}$$

We define parallel overhead to be time spent executing something other than the user's computation. Sources of parallel overhead include task synchronization and bookkeeping operations to prepare tasks for the possibility of being stolen.

Our goal is to pinpoint parallel overhead with logical call path profiling. For library-based programming models such as Pthreads, identifying parallel overhead is easy: any time spent in a routine in the Pthreads library can be labeled as parallel overhead. For language-based parallel programming models, the problem is harder because within a working thread, both overhead and useful work are indistinguishable without prior arrangement. We could use instrumentation, but that is too costly.

Our main insight is that if we could distinguish instructions that contribute to overhead from the application's work, then we could quantify parallel overhead. In the case of Cilk, we modified the compiler to tag statements in its generated code to identify instructions that are associated with parallelization overhead. The tags therefore partition the application code into instructions corresponding to either useful work or overhead. These tags could take several forms, but one particularly convenient form is to associate overhead instructions with special file or procedure names within the binary's debugging information. For example, synchronization code could be tagged with the special procedure or file name `parallel-overhead:sync`. In a post-mortem analysis, we recover the compiler-recorded tags, identify instructions associated with overhead, and attribute any samples of work associated with them to parallelization overhead. In Section 5.3.2, we describe how we mark sources of parallel overhead for Cilk.

The key benefit of this scheme is that tags are only meta-information: they can be created and used without affecting run-time performance in *any* way. (Although tags consume space, they need not be loaded into memory at run time.) In addition, the tags may be refined to partition sources of overhead into multiple types. For example, it may be useful to distinguish between task-packaging overhead and all other overhead. Such a refinement would provide more detailed information to users or analysis tools.

parallel		
idleness	overhead	interpretation
low	low	effectively parallel; focus on serial performance
low	high	coarsen concurrency granularity
high	low	refine concurrency granularity
high	high	switch parallelization strategies

Figure 5.1: Using parallel idleness and overhead to determine if the given application and input are effectively parallel on n cores.

5.2.3 Analyzing Efficiency

In a parallel program, one must consider two kinds of efficiency: parallel efficiency across multiple processor cores and efficiency on individual processor cores. With information about parallel idleness and overhead attributed hierarchically over loops,[1] procedures, and the calling contexts of a program, we can directly assess parallel efficiency and provide guidance for how to improve it. Figure 5.1 provides a high-level guide for interpreting the results. If a region of the program (e.g., a parallel loop) is attributed with high idleness and low overhead, the granularity of parallelism could profitably be reduced to enhance parallel efficiency. If the overhead is high and the idleness low, the granularity of the parallelism should be increased to reduce overhead. If the overhead is high and there is still insufficient parallelism, the parallelism is inefficient and no granularity adjustment will help; keeping the idle processors busy requires a different parallelization. For instance, one might use a combination of data and functional parallelism rather than one alone.

One can assess the efficiency of *work* and identify rate limiting factors on individual processor cores by using metrics derived from hardware performance counter measurements. Many different factors can limit an application's performance such

[1]Because we collect performance metrics using asynchronous sampling of hardware performance counters, which associates counts directly with instructions, and use binary analysis to associate instructions with higher-level program structures such as loops, we can directly compute and attribute metrics at the level of individual loops.

as instruction mix, memory bandwidth, memory latency, and pipeline stalls. For each of these factors, information from hardware performance counters can be used to compute derived metrics that quantify the extent to which the factor is a rate limiter. Consider how to assess whether memory bandwidth is a rate limiter. During an execution, one can sample hardware counter events for *total cycles* and *memory bus transactions.* By multiplying the sampling period by the sample count for each instruction, one can obtain an estimate of how many bus transactions are associated with each instruction. By multiplying the number of bus transactions by the transaction granularity (e.g., the line size for the lowest level cache), one can compute the amount of data transferred by each instruction. By dividing the amount of data transferred by instructions within a scope (e.g., loop) by the total number of cycles spent in that scope, one can compute the memory bandwidth consumed in that scope. By comparing that with a model of peak bandwidth achievable on the architecture, one can determine whether a loop is bandwidth bound or not.

5.3 Measurement and Analysis of Cilk Executions

To demonstrate the power of using our parallel idleness and overhead metrics in combination with logical call path profiling, we added capabilities to HPCToolkit to profile programs written in Cilk-5 [58] (currently at version 5.4.6).

To attribute work, idleness, and parallel overhead metrics to source-level calling contexts, HPCToolkit's `hpcrun` tool collects logical call path profiles (Chapter 4). After a profile is collected, HPCToolkit's `hpcprof` tool correlates the work, idleness, and parallel overhead metrics with the static and dynamic structure of the Cilk source program. Finally, HPCToolkit's `hpcviewer` interactively presents the resulting performance data. In the following sections, we describe our approach, along with minor related modifications to the Cilk scheduler.

5.3.1 Parallel Work and Idleness

To support measurement of our idleness metric, we modified the Cilk scheduler to classify threads as working or non-working and to maintain the number of working and idle threads (W and I, respectively). These modifications were straightforward. Each worker thread executes a scheduling loop that acquires work (through a steal, if necessary) and then performs that work. Since the work is executed via a method call, the scheduling loop is 'exited' to perform the work and then re-entered as the worker thread waits to acquire more work. To identify a thread as actively working or idle, we set a thread-specific state variable just before the thread exits or enters the scheduling loop, respectively. At the same time, a global counter representing the number of working threads is atomically incremented or decremented as each thread exits and enters the scheduling loop, respectively. When a sample event interrupts a worker thread, one of two things happen. If the worker is idle, the sample event is ignored. Otherwise, if the worker is active, `hpcrun` collects the logical calling context for the sample point and then attributes one sample to the context's work metric and a fractional sample I/W to the context's idleness metric.

5.3.2 Parallel Overhead

To attribute parallel overhead to logical calling contexts we use several mechanisms to identify all overhead inserted by the Cilk compiler into a Cilk application binary. At run time, `hpcrun` attributes all work-related samples to the logical call path profile's work metric, regardless of whether these samples represent useful work or overhead.

Then, after program completion, hpcprof uses a post-mortem analysis to reattribute work-related samples to either a useful-work or overhead metric.

Our strategy for identifying the parallel overhead within a Cilk application binary relies on HPCToolkit's hpcstruct binary analysis tool for recovering program structure from a binary. hpcstruct analyzes an application binary to recover a mapping between object code and program structure. In particular, hpcstruct recovers the structure of procedures, including a procedure's loop nests, and identifies code that has been inlined therein. Thus, hpcstruct naturally identifies overhead-related code in a procedure if that code appears to have been inlined. To simulate inlining, we use #line compiler directives.

Given this overall strategy, we used two different methods to ease the implementation effort. The Cilk compiler compiles Cilk source code to C and then uses a vendor C compiler to generate an executable. It turns out that nearly all parallel overhead inserted into the intermediate C code by the Cilk compiler is encapsulated either by a call to a method or macro.[2] Consequently, it is possible to identify essentially all overhead by (1) tagging about 45 Cilk run-time library routines with #line directives, and (2) inserting appropriate #line directives surrounding the appropriate macro references before the generated C code is fed to the vendor compiler.[3] Given this fact, and given our unfamiliarity with the Cilk compiler's source code, we determined that instead of modifying the compiler it would be easier to (1) appropriately tag the Cilk run-time routines and (2) write a Cilk post-processor that inserted the appropriate tags in the intermediate C file. To preserve the ability to recover sensible structure for a routine and use a debugger with the resulting executable, our post-processor

[2]Parallel overhead that derives neither from a method nor macro call is either continuation control flow, a declaration, or trivial.

[3]When a macro is expanded by the C preprocessor, no indication of its originating source file is typically recorded. In contrast, if a function call is inlined, a C compiler will effectively generate the appropriate #line directives.

```
1 cilk int fib(int n)
2 {
3   if (n < 2)
4     return (n);
5   else {
6     int x, y;
7     x = spawn fib(n - 1);
8     ...
```

```
1 int fib(WorkerState* ws, int n) { struct frame* fr;
2 #line 28 "hpctoolkit:parallel-overhead"
3   CILK2C_INIT_FRAME(fr, ...);
4   CILK2C_START_THREAD_FAST();
5 #line 28 "fib.cilk"
6
7   if (n < 2) { int t = n;
8 #line 31 "hpctoolkit:parallel-overhead"
9     CILK2C_BEFORE_RETURN_FAST();
10 #line 31 "fib.cilk"
11     return t;}
12   else {
13     int x; int y;
14     { fr->header.entry=1; fr->scope0.n = n;
15 #line 34 "hpctoolkit:parallel-overhead"
16       CILK2C_BEFORE_SPAWN_FAST();
17       CILK2C_PUSH_FRAME(fr);
18 #line 34 "fib.cilk"
19       x = fib(ws, n - 1);
20 #line 34 "hpctoolkit:parallel-overhead"
21       CILK2C_XPOP_FRAME_RESULT(fr, 0, x);
22       CILK2C_AFTER_SPAWN_FAST();
23 #line 34 "fib.cilk"
24     }
25     ...
```

Figure 5.2: (a) Fragment of a Cilk program for computing Fibonacci numbers; and (b) compiled C code for that fragment. Regions of parallel overhead are demarcated with `#line` directives that contain special file names.

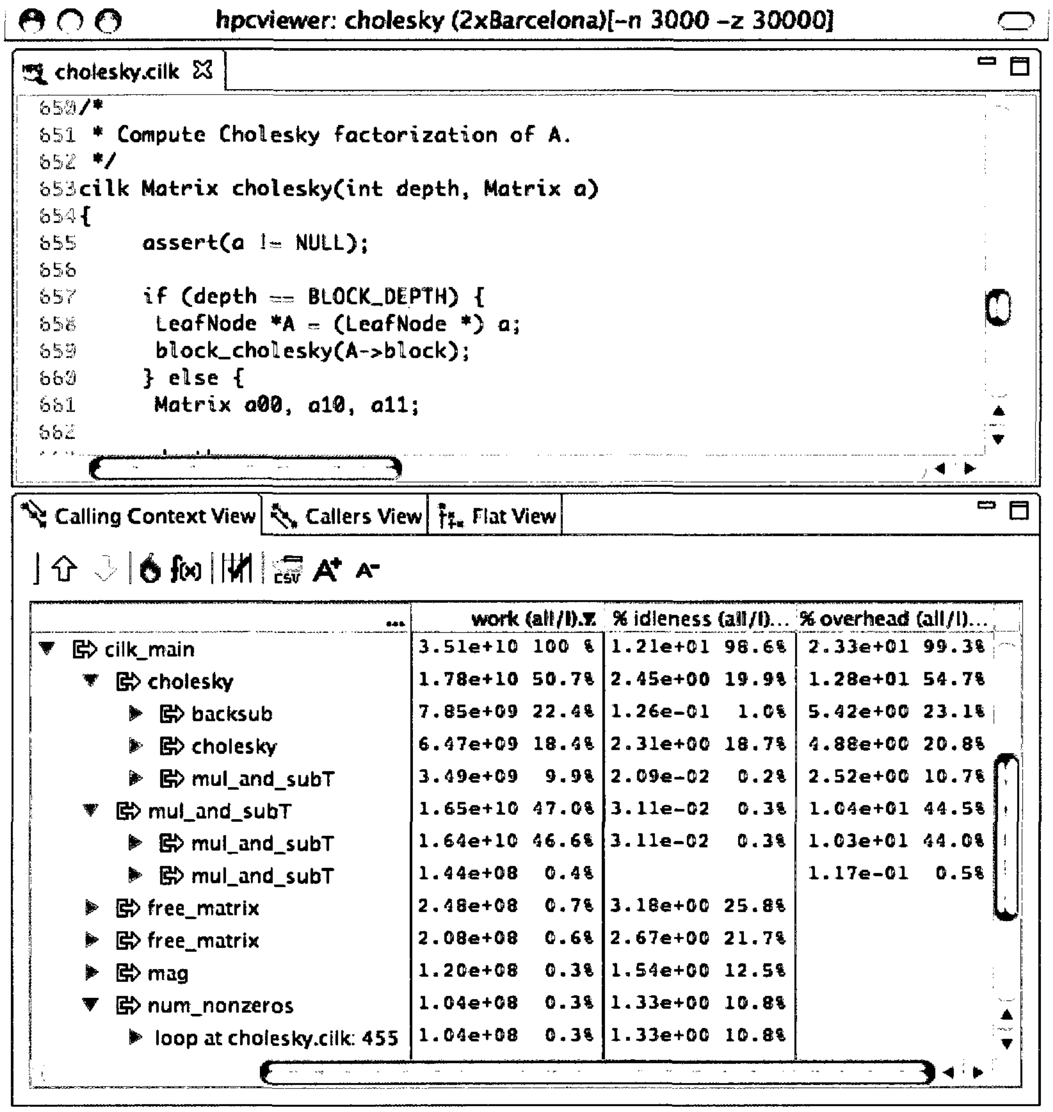
preserves the line number of the original source file. A sanitized example of an original Cilk routine and its corresponding post-processed C code is shown in Figure 5.2. (Note that the 'unusual' formatting in the post-processed C, such as Cilk's frame `struct` declaration on line 1 of Figure 5.2(b), is critical for aligning the line numbers of the generated code with the source.)

5.3.3 Case Study

To demonstrate the effectiveness of attributing work, parallel idleness and parallel overhead to logical call path profiles, we applied our method to analyze the performance of a Cilk program for Cholesky decomposition. We used the example Cholesky program included in the Cilk 5.4.6 source distribution. We profiled a problem size of 3000 $\times$ 3000 (30,000 non-zeros) on an SMP with dual quad-core AMD Opterons (2360 SE, 2.5 GHz) and 4 GB main memory.

Figure 5.3 presents one view of the aggregated results displayed by our presentation tool `hpcviewer`. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, partially expanded. One can see several source-level procedure instances along the call paths. (Physical procedure instances are not shown.) The selected line in the navigation pane and the source pane (top sub-pane) shows the procedure `cholesky`. Each entry in the navigation pane is associated with metric values in the metric pane to the right. Sibling entries are sorted with respect to the selected metric column (in this case 'work (all/I)'). Observe at the bottom of the navigation pane a loop, located within the context of `cilk_main`; the navigation pane actually contains a fusion of the dynamic logical calling contexts and static loop contexts.

The metric columns in Figure 5.3 show values for work (useful-work, in cycles), parallel idleness and parallel overhead. These values are summed over all of the eight worker threads, yielding the 'all' qualifier in their names. Both idleness and overhead are shown as percentages of total effort, where effort is the sum of work, idleness and overhead. In the idleness and overhead columns, the values in scientific notation represent the aforementioned percentages; the values shown as percentages to their right give an entry's proportion of the total idleness or overhead, respectively. The metrics are *inclusive* (hence the 'I' qualifier) in the sense that they represent values



hpcviewer: cholesky (2xBarcelona)[-n 3000 -z 30000]

cholesky.cilk

```
/*
 * Compute Cholesky factorization of A.
 */
cilk Matrix cholesky(int depth, Matrix a)
{
     assert(a != NULL);

     if (depth == BLOCK_DEPTH) {
      LeafNode *A = (LeafNode *) a;
      block_cholesky(A->block);
     } else {
      Matrix a00, a10, a11;

```

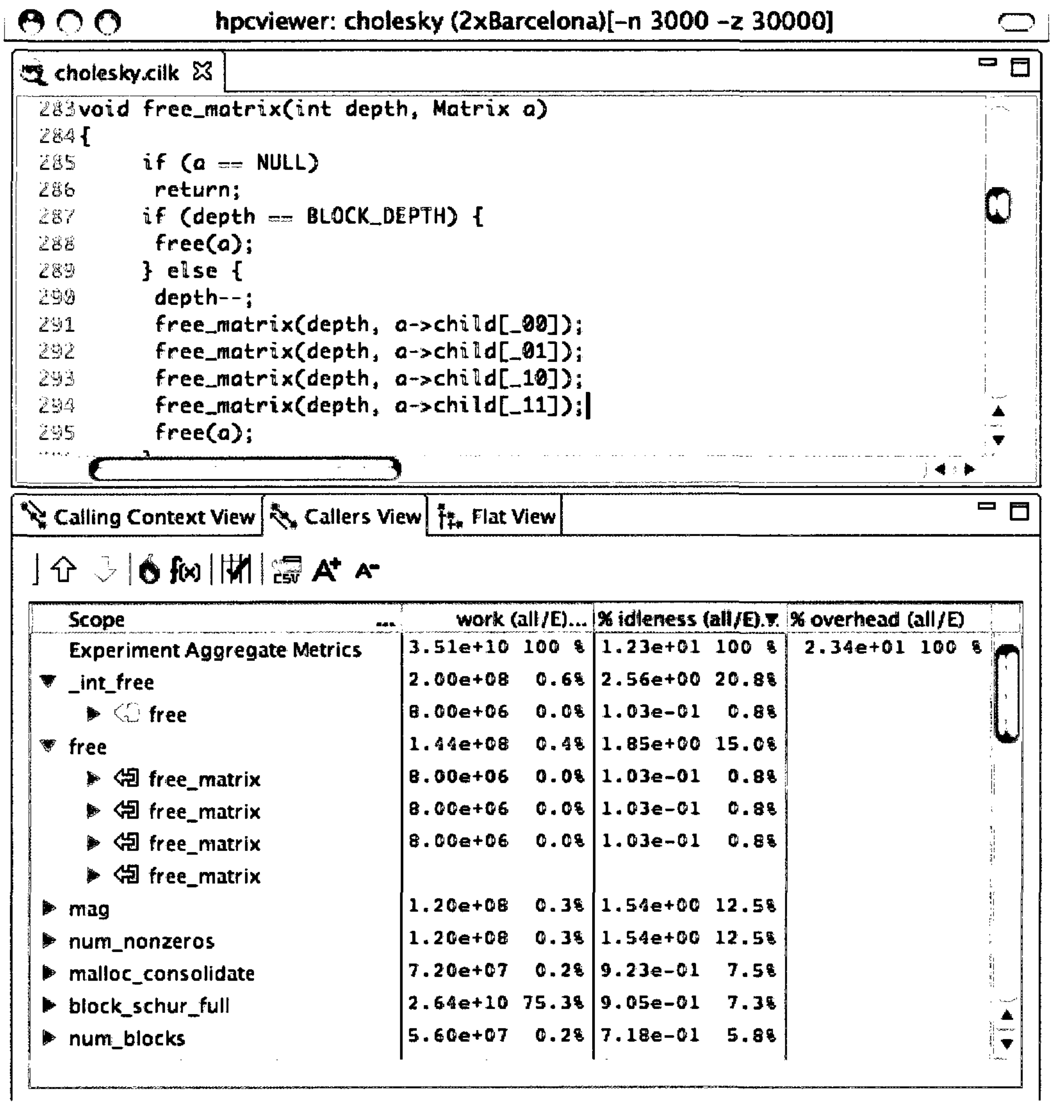
```
                                  work (all/I)   % idleness (all/I)...  % overhead (all/I)...
▼ cilk_main                       3.51e+10 100 %  1.21e+01 98.6%   2.33e+01 99.3%
   ▼ cholesky                     1.78e+10 50.7%  2.45e+00 19.9%   1.28e+01 54.7%
      ▶ backsub                   7.85e+09 22.4%  1.26e-01  1.0%   5.42e+00 23.1%
      ▶ cholesky                  6.47e+09 18.4%  2.31e+00 18.7%   4.88e+00 20.8%
      ▶ mul_and_subT              3.49e+09  9.9%  2.09e-02  0.2%   2.52e+00 10.7%
   ▼ mul_and_subT                 1.65e+10 47.0%  3.11e-02  0.3%   1.04e+01 44.5%
      ▶ mul_and_subT              1.64e+10 46.6%  3.11e-02  0.3%   1.03e+01 44.0%
      ▶ mul_and_subT              1.44e+08  0.4%                   1.17e-01  0.5%
   ▶ free_matrix                  2.48e+08  0.7%  3.18e+00 25.8%
   ▶ free_matrix                  2.08e+08  0.6%  2.67e+00 21.7%
   ▶ mag                          1.20e+08  0.3%  1.54e+00 12.5%
   ▼ num_nonzeros                 1.04e+08  0.3%  1.33e+00 10.8%
      ▶ loop at cholesky.cilk: 455 1.04e+08  0.3%  1.33e+00 10.8%
```

Figure 5.3: `hpcviewer`'s Calling Context view of Cholesky.

for the associated procedure instance in addition to all of its callees. Thus, the metric name 'work (all/I)' means inclusive work summed over all threads.

Because Cilk emphasizes algorithms based on recursive decomposition — parallelism is exposed through asynchronous procedure calls — call chains can become quite long. Nevertheless, expanding the calling context tree to the first call of `cholesky` and noting the metrics on the right is very informative. Figure 5.3 shows that 50.7% of of the total work of the program is spent in the top level call to cholesky; the top level call to mul_and_subT (which verifies the factorization) is a close second at about 47.0%. We can also quickly see that 19.9% and 54.7% of the total parallel idleness and overhead, respectively, occur in cholesky. However, because this idleness and overhead are relatively small with respect to effort (about 2.45% and 12.8%, respectively), we can say that the parallelization of cholesky is effective for this execution. In contrast, the parallelization of the entire program (for which we can use cilk_main as a proxy) is less effective, with both overhead and idleness increasing to 23.3% and 12.1% of total effort, respectively.

To pinpoint exactly where inefficiency occurs using the idleness and overhead metrics, we turn to the Callers or bottom-up view in Figure 5.4. If the top-down view looks down the call chain, the bottom-up view looks up to a procedure's callers. Thus at the first level, the bottom-up view lists all the procedures in the program, rank-ordered according to the selected metric — in this case, relative idleness. Note that in contrast to Figure 5.3, these metric values are *exclusive* (signified with an 'E') in the sense that they do not include values for a procedure's callees. The top two routines in the rank-ordered list are versions of the C library routine free and together account for about 35.8% (20.8% + 15.0%) of the program's idleness. When the callers for these routines are expanded, it is evident that they are both called by free_matrix, a *non*-Cilk, i.e., *serial*, helper routine that deallocates the matrix for the Cholesky driver. Continuing down the list reveals that every routine shown in the screen shot is a serial helper. Since each of these serial routines except block_schur_full is related to initialization or finalization, it is immediately evident that to reduce parallel idleness either the size of the matrix must be increased or the initialization and finalization routines must be parallelized. The significance of this conclusion is that *without having any prior knowledge of the source code*, our techniques have enabled us to quickly make strong and precise statements about the



```
void free_matrix(int depth, Matrix a)
{
    if (a == NULL)
     return;
    if (depth == BLOCK_DEPTH) {
     free(a);
    } else {
     depth--;
     free_matrix(depth, a->child[_00]);
     free_matrix(depth, a->child[_01]);
     free_matrix(depth, a->child[_10]);
     free_matrix(depth, a->child[_11]);
     free(a);
```

Scope	work (all/E)...	% idleness (all/E)	% overhead (all/E)
Experiment Aggregate Metrics	3.51e+10 100 %	1.23e+01 100 %	2.34e+01 100 %
▼ _int_free	2.00e+08 0.6%	2.56e+00 20.8%	
▶ free	8.00e+06 0.0%	1.03e-01 0.8%	
▼ free	1.44e+08 0.4%	1.85e+00 15.0%	
▶ free_matrix	8.00e+06 0.0%	1.03e-01 0.8%	
▶ free_matrix	8.00e+06 0.0%	1.03e-01 0.8%	
▶ free_matrix	8.00e+06 0.0%	1.03e-01 0.8%	
▶ free_matrix			
▶ mag	1.20e+08 0.3%	1.54e+00 12.5%	
▶ num_nonzeros	1.20e+08 0.3%	1.54e+00 12.5%	
▶ malloc_consolidate	7.20e+07 0.2%	9.23e-01 7.5%	
▶ block_schur_full	2.64e+10 75.3%	9.05e-01 7.3%	
▶ num_blocks	5.60e+07 0.2%	7.18e-01 5.8%	

Figure 5.4: hpcviewer's Callers view of Cholesky.

parallel efficiency of this program. Although it is not surprising that serial code is responsible for idleness, the fact that we can immediately quantify and pinpoint its impact on parallel efficiency shows the effectiveness of our methods.

5.4 Related Work

Our parallel idleness metric is similar to Quartz's [12] notion of *normalized time* to highlight code with poor concurrency. Normalized time is computed by attributing $1/W$ to the relevant section of code on each sample of a working thread, inflating compute times in areas of poor parallelization. While our idleness metric is similar in that it also highlights code sections with poor concurrency, it is different in that it is a direct measure of parallel idleness: I/W. This quantitative/qualitative distinction is important because Quartz's qualitative metric can be ambiguous. Consider a program that executes with n threads (on n cores) with two phases named ϕ_{x} and ϕ_{y}, where each phase executes for an equal amount of time, t. During phase ϕ_{x}, procedure x executes serially; during phase ϕ_{y}, n instances of procedure y execute without any loss to overhead. Unintuitively, the normalized times $||\tau_{\mathsf{x}}||$ and $||\tau_{\mathsf{y}}||$ for procedures x and y are identical ($t/1$ and nt/n, respectively) even though $n-1$ threads are idle for the whole duration of phase ϕ_{x}. In contrast, our idleness metric would yield values of $\mathcal{I}_{\mathsf{x}} = (n-1)t$ and $\mathcal{I}_{\mathsf{y}} = 0$. Although Quartz eliminates this ambiguity by using n counters for each procedure, assigning t to counter x_1 and 0 to counters $\mathsf{x}_2 \ldots \mathsf{x}_n$, this solution requires a comparison between n counters to convey the same thing as $\mathcal{I}_{\mathsf{x}}$. Additionally, we attribute idleness to full logical calling contexts, even in the presence of a work-stealing run time.

The idea of computing parallel overhead is not new. For example, *cycle accounting* is a powerful methodology for partitioning stall cycles during the execution of serial code [55, 84]. To predict parallel performance, Crovella and LeBlanc describe a *lost cycles analysis* [45] that separates parallel overhead from pure computation. They further divide parallel overhead into sub-categories useful for differentiating between different performance problems. However, they lament that "[m]easuring lost cycles directly for the entire environment space is still impractical." Our method directly measures parallel overhead without any run-time cost above and beyond that of normal profiling.

It is interesting to compare our performance analysis of Cilk to Cilk's own performance metrics. Cilk computes two metrics that attempt to directly correspond to the theoretical model that underlies Cilk's provably-efficient scheduler. The first is total work or the time for a serial execution of the program with a given input. The second is critical path, or a prediction of the execution time on an infinite number of processors. The significant advantages of Cilk's metrics are that they approximate a platform independent model and provide a theoretical upper bound on the scalability of a program with a given input. However, they share two important disadvantages. First, Cilk's metrics are computed using extremely costly instrumentation — which itself disturbs the application's performance characteristics. Second, these metrics do not aid the programmer in pinpointing *where* in the source code inefficiency arises. In contrast, our method *immediately pinpoints parallel inefficiency in source-level code*. Moreover, paired with hardware performance counter information, our method can help distinguish between different types of architectural bottlenecks in different regions of code.

Critical path is a classic metric for understanding parallel programs. While Cilk computes the critical path's lower bound for a program and given input, it is also possible to determine the actual critical path for an execution. Intel's VTune [77] computes the actual critical path for an execution, though at the native thread level. The classic problem with critical path information is that after expending much effort to reduce its cost, a completely different critical path may emerge, slightly less costly than the original. Therefore, it is much more useful to know how much 'slackness' exists in the critical path. Intel's Thread Profiler [26, 76] not only computes critical path but classifies its segments by concurrency level and thread interaction. Given a segment where n_T threads execute on n cores ($n > 1$), the tool classifies that segment's concurrency level as either serial ($n_T = 1$), under-subscribed ($1 < n_T < n$), fully parallel ($n_T = n$), or oversubscribed ($n_T > n$). These categories are then qualified by three interaction effects, which are called cruise time, impact time and blocking time. *Cruise time* is time that a thread does not delay the next thread on the critical path while *impact time* is the opposite. If a thread on the critical path waits for some external event, it accumulates *blocking time.* Thus, performance tuners should focus on areas of serial or under-subscribed impact time rather than fully parallel cruise time. The disadvantages of Thread Profiler are that it uses costly instrumentation, reports information at the native (Win32) thread level, and does not provide contextual information.

An interesting observation about our idleness and overhead metrics is that, in the context of Cilk, they approximate a quantitative measure of critical path slackness, tied to full calling context. To see this, note that a Cilk worker thread is idle only if it is waiting for another worker thread to (1) make asynchronous calls or (2) release a lock. Therefore, if a thread's idleness is high in a certain context, then that context was on one of the 'interesting' critical paths. One deficiency of our profile data is that it does not distinguish between idleness (or overhead) that is the result of a few calls to a long-running function as opposed to many calls to a fast one. However, given the properties of the Cilk scheduler, we can compute metrics similar to Thread Profiler's but for a fraction of the overhead.

5.5 Discussion

Because of the growing need to develop applications for multicore architectures, effective tools for quantifying and for pinpointing performance bottlenecks in multithreaded applications are absolutely essential. This will be increasingly true as less skilled application developers are forced to write parallel programs to benefit from increasing core counts in emerging processors.

We have shown that attributing work, parallel idleness and parallel overhead to logical calling contexts enables one to quickly obtain unique insight into the run-time performance of Cilk programs. In particular, we demonstrated the power of our method by using it to pinpoint and quantify serialization in a Cilk execution. A strength of our approach is that our performance metrics are completely intuitive and can be mapped back to the user's programming abstractions, even though the run-time realization of these abstractions is significantly different. While we described a prototype tool for measurement and analysis of multithreaded programs written in Cilk, our underlying techniques for computing parallel idleness, parallel overhead, and obtaining logical call path profiles are more general and can be applied directly to other multithreaded programming models such as OpenMP and Threading Building Blocks.

Our work shows that it is possible to construct effective and efficient performance tools for multithreaded programs. The run-time cost of our profiling can be dialed down arbitrarily low by reducing the sampling frequency. We have also shown that it is possible to collect implementation-level measurements and project detailed metrics to a much higher level of abstraction without compromising their accuracy or utility.

Chapter 6

Analysis of Multithreaded Executions: Lock Contention

6.1 Introduction

Many programs exploit shared-memory parallelism using multithreading based on thread libraries such as POSIX Threads (Pthreads) [32]. Despite a recent surge of interest in transactional memory [82], locks remain the principal mechanism used to guard the integrity of shared data structures in multithreaded programs. Indeed, fine-grain locking remains the gold standard for performance. Moreover, some of the fastest software implementations of transactional memory use locks under the hood [50].

Contention for locks has long been recognized as a key impediment to performance for shared-memory parallel programs. Early simulation studies of large-scale shared-memory parallel systems showed that hot spots, such as those caused by spin-waiting for locks on machines without coherent caches, could dramatically degrade performance by clogging multistage interconnection networks [115]. Later work explored alternative implementations for locks that reduce interconnection network traffic associated with spin-waiting, e.g., [11, 94]. Today, the potential for performance losses in parallel systems due to synchronization traffic resulting from spin-waiting is well understood and in most cases it can be largely avoided by using appropriate algorithms.

However, there remains a fundamental performance problem caused by using locks in parallel programs and run-time systems: contention for locks causes serialization. As a result, idling while waiting for a lock reduces parallelism and parallel efficiency. For this reason, pinpointing and ameliorating sources of lock contention in parallel applications is of significant interest. As the number of cores per processor increases, the scale of multithreading will grow. Diagnosing performance bottlenecks in multithreaded applications will be of increasing interest as multithreaded applications become ubiquitous. A tool that helps pinpoint sources of lock contention and quantifies their performance impact can provide invaluable guidance for tuning multithreaded applications.

This chapter proposes and evaluates three strategies that a performance tool can use to gain insight into performance losses due to lock contention. The approaches we consider move from blaming lock contention on victims, then to suspects, and finally to perpetrators. This shift in perspective can be subtle — the first two strategies are actually modest extensions to state-of-the-art measurement techniques — but it is critical. Section 6.2 explores the utility of attributing the idleness of spin-waiting for locks directly to the calling contexts in which spin-waiting occurs (victims). Section 6.3 considers spreading the blame for idleness due to lock spin-waiting among threads holding locks (suspects). Section 6.4 describes a new strategy for directly blaming a lock holder for the idleness of threads spinning on a lock that it holds (perpetrators).[1]

We evaluate our new strategy of directly attributing blame for lock contention in Section 6.5. We use three codes: MADNESS [69] — a quantum chemistry appli-

[1]The Acknowledgments section recognizes the contributions of collaborators.

cation that makes extensive use of locking; UTS [112] — an unbalanced tree search benchmark; and SSCA #2 [20] — a graph analysis benchmark that is a member of the Synthetic Scalable Compact Application Benchmark suite [46]. For complex applications like these, locks may be acquired frequently — an execution of MADNESS uses 65M distinct locks, a maximum of 340K live locks, and an average of 30K lock acquisitions per second per thread — and the sources of lock contention can be context sensitive. Moreover, a performance tool must not itself significantly affect an execution. This is difficult to ensure. Adding overhead to critical sections can make the tool itself a new source of contention, while adding overhead outside of critical sections can *reduce* contention. Consequently, any tool for understanding lock contention must operate with very low overhead, obtain calling context, and produce insightful metrics. The significance of our result is that we achieve *all* these goals.

Finally, Section 6.6 relates our strategies to prior work; and Section 6.7 discusses the chapter's general themes.

6.2 Attributing Idleness to its Calling Context

6.2.1 A Straightforward Strategy

The first strategy we consider for understanding the impact of lock contention in multithreaded programs is straightforward and is based on two key ideas.

The first idea is to quantify lock contention by measuring lock idleness, i.e., the idle time a thread spends waiting for a lock. Thus, we distinguish between the useful work that a thread performs and its idleness. If a thread repeatedly idles waiting for a lock, then its idleness metric will consume a significant percentage of the thread's total effort (effort = work + idleness).

The second idea is to use *call path profiling* [67] to attribute these metrics to the calling context in which they are incurred. Call path profiling is especially useful for modular programs, where it is important to attribute costs incurred by procedures to the different contexts in which the procedures are called. We use HPCToolkit's hpcrun [141], a sampling-based call path profiler that attributes metrics to the full static and dynamic contexts in which they are incurred. Asynchronous-sampling-based call path profilers use a recurring event trigger to raise signals within the program being profiled. When an event trigger occurs, it raises a signal, and a signal handler obtains a call path by unwinding the call stack. HPCToolkit's profiler incurs minimal overhead for reasonable sampling frequencies (typically 2–3% for hundreds to thousands of samples/second) and is capable of measuring and attributing performance metrics to fully optimized code.

To combine these two ideas, when attributing a sample to its calling context, it is necessary to know whether the sample represents work or idleness. Consider the case of *right-sized* parallelism, where each thread is associated with a unique hardware context. In this case, threads would typically use spin locks, i.e., locks that busy-wait rather than yield to the operating system (OS). Since each thread has a sample source, samples are delivered to a thread both while it is working and while it is spinning for a lock. To determine whether to charge a sample to a work or idleness metric, we intercept a monitored application's calls to lock routines to set a thread-local flag immediately before and after the thread begins waiting for a lock. In contrast to samples, which arrive asynchronously and whose frequency can be controlled independently of the application, this flag is set *synchronously* on *every* lock attempt. Keeping instrumentation overhead low is important; the cost of having locking routines maintain a flag is not a problem.

6.2.2 Blocking (Sleep-waiting)

In contrast to spin locks, Pthreads mutex locks and condition variables sleep-wait. When a thread is sleeping, no user-level resources are used, effectively muting any sampling triggers based on those resources.[2] An obvious solution to the problem at hand is to directly measure lock (or condition variable) wait time. However, this requires gathering time stamps both before and after a wait and, if the idleness is non-zero, attributing it to the calling context. Thus, it is potentially necessary to perform an unwind for *every* lock release, which would cause significant overhead for programs that have a high volume of lock acquisitions and releases. Applying this strategy to measure locking in a non-trivial execution of MADNESS [69] (see Section 6.5.1), which performed 30K lock acquisitions per second per thread, yielded a monitoring overhead of 260%. To reduce this overhead, we can *sample* the lock acquisitions themselves. That is, on every p^{th} lock acquisition, we measure the thread's idleness I and attribute $p \times I$ units of idleness to the calling context. In effect, this scheme amortizes the cost of heavyweight instrumentation across p lock acquisitions.

6.2.3 Evaluation

For the Pthreads library, we implemented this strategy by *overriding* routines that could potentially cause a thread to idle: `pthread_{spin,mutex}_lock` and `pthread_cond_wait`. To override a routine in a dynamically linked application, we use library preloading.[3] That is, at program launch time, HPCTOOLKIT injects a dynamically linked profiling library into an unmodified program's address space. For statically linked programs, compilation remains unchanged, but we require users to adjust their

[2]It is possible to use a sampling trigger based upon real time rather than user time, but on standard OS's, this does not work well with threads. For example, on Linux, `ITIMER_REAL` does not provide a thread-specific sample source and therefore delivers signals to a random thread within a process.

[3]On Linux, see the loader's special environment variable `LD_PRELOAD`.

link step to invoke a script that adds HPCToolkit's profiling library to a statically linked executable.[4] When a monitored application calls one of the overridden routines, control is transferred to the monitored version of the routine, or the *override*. The override then sets a thread-local idleness flag — pessimistically assuming the thread will idle — and immediately calls the actual Pthreads routine. When the thread enters the lock or condition variable critical section, the Pthreads routine returns to the override, which immediately clears the idleness flag and returns to the monitored application.

This strategy computes a thread's idleness with accuracy and with low overhead. On average, a thread receives samples while its idleness flag is set in proportion to the time it is actually idle. If a thread attempts to acquire a lock many times but without contention, that thread will spend relatively little time with its idle flag set and its idleness metric will be proportionally small. In contrast, if a thread spends a large percentage of time idle, whether due to few or many lock acquisitions, its idleness metric will proportionally reflect this fact. Consequently, our conservative assumption yields a simple implementation without sacrificing accuracy. Another important benefit of this scheme is that all data is thread-local which means that it naturally scales to a large number of threads.

One limitation of our implementation is that it does not handle over-subscription — i.e., when there are more threads than available hardware contexts — if a thread sleep-waits.

The more serious limitation of this approach is that it fails to yield the insight into lock contention that we desire. While this idleness metric reflects contention in the sense that higher contention results in higher idleness, it pinpoints the symptom rather than the cause; the victim rather than the perpetrator. In other words, this

[4]On Linux, see the linker's special `--wrap` option.

idleness metric takes a 'first party' view of lock contention and records its effect rather than its provenance by blaming a waiting thread for its own waiting. To pinpoint the cause of idleness, idle threads must have some 'third party' knowledge about which threads are responsible for their idleness. We next describe an idleness metric that attempts to account for this problem.

6.3 Blaming Idleness on Lock-holders

6.3.1 Extending a Prior Strategy

In Chapter 5, we recognized the problem of attributing idleness as a symptom rather than as a problem source. There, we described an idleness metric that blamed idleness in work-stealing programs to regions of code with too little parallelism. In Cilk [58], such parallelism is expressed with asynchronous calls. We implemented our ideas by modifying the Cilk run time to (1) track when an individual thread was working or idle; and (2) maintain a node-wide counter representing the total number of working (W) and idle (I) threads. Like the strategy of Section 6.2, if a sample event occurs in a thread that is actively working, the thread attributes that sample to a work metric associated with the sample context. However, there are two key differences. First, the working thread also attributes a fractional sample I/W to an idleness metric associated with the sample context to blame itself for the current idleness in the execution. Second, if a sample occurs in an idle thread, it is simply ignored. This strategy equally spreads the blame for not keeping threads busy at that moment to the active contexts of working threads.

This strategy can be adapted to Pthreads. As in Section 6.2, we override Pthreads routines that potentially cause a thread to idle (`pthread_{spin,mutex}_lock` and `pthread_cond_wait`). We add a node-wide counter to maintain the number of working threads, W. During an override, immediately before calling an actual Pthreads library primitive that might wait, we atomically decrement W; we then increment W when the primitive returns. At any point in time, I can be computed implicitly as $T - W$, where T is the number of threads. Then we process samples as described above.

One natural benefit of this strategy is that there is no need to distinguish between spin-waiting and sleep-waiting. In the first strategy it was necessary to handle sleep-waiting specially (using timers) because sleeping threads do not receive samples. However, in this scheme, any samples received by an idle thread are already ignored.

Although our prior work suggested that this strategy could be effectively applied to Pthreads, we found that it did not yield actionable insight into lock contention within complex applications like MADNESS. There is a simple explanation for why evenly apportioning blame for waiting due to lock contention is not very useful for threaded applications. For a work-stealing scheduler such as Cilk, any working thread may rightly be blamed for idleness: if that thread is not shedding parallel work, it is part of the cause of idleness. However, the same is not true for lock-waiting in explicitly threaded programs. For example, if one thread is working but not holding a lock, then it is misleading for that thread to accept blame for threads contending for a lock. Consequently, evenly apportioning blame is not a sound strategy.

To rectify the problem of misappropriated blame, we redesigned our strategy to assign blame more precisely. We wish to apportion idleness deriving from lock contention only to threads that hold locks. We also wish to minimize the number of atomic increments that are required during critical sections.

We first observe that working threads W may be in one of three mutually exclusive states:

W_l: working directly in a lock critical section

W_c: working directly in a condition variable critical section

W_o: working neither directly nor indirectly within any critical section (other)

Note that because critical sections can be nested, a thread in state W_{co} may additionally acquire another lock, moving to state W_l until this additional lock is released. (Again, we ignore the case of over-subscription.)

Similarly, idle threads I may be classified according to one of three mutually exclusive states:

I_l: idling at a (non-condition variable) lock

$I_{c,l}$: idling at a condition variable lock (i.e., the thread has been signalled but is waiting to obtain the associated condition variable lock)

$I_{c,v}$: idling at condition variable (i.e., waiting for a signal)

Given these observations, the most natural form of blaming is:

- Blame idleness I_l on workers in state W_l.
- Blame idleness $I_{c,l}$ on workers in state W_c.
- Blame idleness $I_{c,v}$ on workers in state W_o since any of the workers in state W_o could signal the threads in state $I_{c,v}$.

In the first two cases, idleness is blamed on the worker directly responsible for it. In the third case, it is impossible to attribute idleness directly since, relative to the Pthreads API, no particular thread is necessarily responsible for signalling.

6.3.2 Making It Practical

To implement this revised scheme for Pthreads it is necessary to make a minor adjustment to what we have just presented. At the user-level it is impossible to distinguish between idleness categories $I_{c,l}$ and $I_{c,v}$. While it is possible to distinguish between threads waiting for only the condition variable lock and both a signal and the lock, this distinction can only be made within the Pthreads library. As discussed more fully in Section 6.4, our interest is in building tools by using techniques that are as general and portable as possible. Since the Linux Pthreads library is part of the low-level `glibc` system library, revising Pthreads would require that we recompile a system-level library (and possibly relink the monitored application) before using our tools. Therefore, we merge categories $I_{c,l}$ with $I_{c,v}$ and W_c with W_o to obtain the new rule:

- Blame idleness in category $I_c = I_{c,l} + I_{c,v}$ on workers in state $W_{co} = W_c + W_o$.

Clearly, it is possible to use four global counters to compute the number of idle and worker threads in states I_l, I_c, W_l, and W_{co}. Unfortunately, these counters require frequent adjustment within critical sections. Because a key implementation concern is minimizing the overhead of the Pthreads overrides, it is important to refrain from lengthening critical sections. For example, it is less of a problem for the override to perform bookkeeping *before* calling the actual `pthread_spin_lock` routine as opposed to after this routine has returned and the lock acquired. Therefore, it is important to minimize the number of atomic increments during critical sections.

It is possible to reduce the number of frequently maintained counters. Given that $T = W + I$, we have

$$W = W_l + W_{co}$$

$$I = T - W = I_l + I_c$$

Consequently, to compute all necessary values it is possible to use T (which only changes on thread creation/destruction) along with only three frequently adjusted counters, e.g., W, W_l and I_c. All other state can be thread-local. By directly main-

Algorithm 6.1: blame-suspects: On sampling a working thread, compute that thread's blame for the execution's idleness based on associated suspects.

```
Assume: $T$, $W$, $W_l$ and $I_c$ are directly maintained.
Input: $T$, $W$, $W_l$ and $I_c$
$W_l \Leftarrow \mathsf{max}(1, W_l)$                        // $W_l \geq 1$
$I_c \Leftarrow \mathsf{max}(0, I_c)$                        // $I_c \geq 0$
if is working within lock then
    let $I = (T - W)$                                        // $I \geq 0$
    let $I_l = \mathsf{max}(0, I - I_c)$                     // $I_l \geq 0$
    return $I_l / W_l$
else
    let $W_{co} = \mathsf{max}(1, W - W_l)$                  // $W_{co} \geq 1$
    return $I_c / W_{co}$
```

taining the suggested subset of counters, only two counters need to be atomically adjusted within lock and condition variable critical sections.

Algorithm 6.1 shows how this scheme apportions idleness when a sample is fielded by a working thread. If the worker is in category W_l, it attributes one unit of work to its work metric and I_l/W_l units of idleness to its idleness metric. Otherwise the worker is in category W_{co} and it attributes I_c/W_{co} units of idleness to its idleness metric. The algorithm uses max to account for possible timing windows between the (multiple) atomic increments that occur during the overrides.

It is worth noting that there are complications with correctly maintaining the global counters. For example, because critical sections can be nested, a thread can move from state W_{co} to W_l and back, which means that correctly maintaining counters requires some care.

6.3.3 Evaluation

Unfortunately, we found that even our extension to more precisely attribute blame was ineffective for complex programs. There are two key problems.

The first problem is that, as was *not* the case with work stealing, contention to atomically increment or decrement the global counters can be a significant issue. By using tuned primitives and by preventing false sharing with cache-block alignment, we managed to bring overhead to an acceptable 5% on a 16-core machine. Nevertheless, even though we managed to achieve respectable overhead, the prospect of 48- and 64-core systems — or massively multithreaded systems such as the Cray XMT — suggests that global counters are likely to be an important weakness. A monitoring scheme should not itself cause significant amounts of new contention.

The second problem is even more fundamental. Even assuming low-overhead monitoring, we found that the lock-contention blame of this approach was still spread too diffusely for complex applications. While the approach of Section 6.2 attributes blame to *victims*, this approach targets *suspects.* While it is an improvement to attribute the idleness of lock-waiting threads to lock-working threads, the results can be inaccurate if most of the idling threads are waiting on one critical lock. For similar reasons, it can be misleading to attribute the idleness of 'cond'-waiting threads to all other working threads, even though any one could in theory potentially signal the condition variable. Consequently, for complex programs, we found blame to be too diluted because it is accumulated by actively working threads that have no relation to a source of contention.

6.4 Communicating Blame Directly to Lock-holders

6.4.1 Blame Shifting: A Distributed and Precise Strategy

To pinpoint the cause of lock contention in its context, while avoiding the problems we have encountered thus far, we developed a fully distributed scheme that we call *blame shifting* to communicate blame for contention directly to lock-holders.

Because it uses a fully distributed strategy and only lightweight instrumentation of synchronization primitives, it incurs very low overhead.

The key idea is to use a lock as a communication channel for directing blame. Consider the case of spin locks where threads busy-wait while contending for a lock. While profiling an application using sampling, threads contending for locks will receive samples while idling. When a thread takes a sample while waiting for a lock, we use an atomic add to accumulate that idleness in a counter associated with the lock. Then, when a thread that possesses a lock releases it, that thread blames itself for all of the idleness that accumulated while it held the lock. To accept blame, when a thread releases a lock, it atomically swaps zero into the lock's associated idleness counter. If the result of the swap is a non-zero value, then other threads must have contended for that lock while the lock-holder was working. So, the thread holding the lock attributes that idleness to the context of its lock release operation.

Although one might desire to attribute idleness to the lock acquisition point, using the release point provides a key benefit. Typically, there are several points in an execution where certain lock acquisitions are uncontested. Consequently, there are likely to be many lock release points where it is not necessary to incur the cost of unwinding the call stack to attribute zero blame. In contrast, attributing idleness to a lock acquisition point would require eager unwinds since that context may never again exist. Moreover, if a lock is contested only a short time, then it is unlikely to have a sample of idleness attributed to it. To see this, note that whereas a thread may acquire hundreds of thousands of locks per second, it is sufficient to use sample frequencies of hundreds to thousands of samples/second for most programs.

6.4.2 Blame Shifting in Action

To implement blame shifting, it is necessary (1) to have thread-local data to indicate when a thread is not working and (2) to create a shared piece of monitoring state for each lock. As the former has been discussed in prior schemes, we focus on the latter.

In-band versus out-of-band state

The first question is how to create the shared monitoring state. There are two possibilities: within the existing lock structure (in-band) or outside of it (out-of-band).

An in-band approach requires storing additional information within the existing lock. In particular, blame shifting requires a shared idleness counter for each lock. In general, reinterpreting bits within a data structure to add an extra field is difficult and at the very least requires overriding every routine that might access that data. Pthread's spin locks are simply 32-bit integers, even on 64-bit platforms. An in-band approach requires unevenly dividing this space into two fields to have enough room for the idleness counter. It also requires that the idleness field *never* overflow. It is also worth observing that both fields will be accessed by different threads and will be the target of atomic operations, even though neither is the natural architectural word size.

A second option is to create a special library and include file to implement an extended representation for a lock that includes a counter for blame shifting. This approach suffers from the disadvantage that one would need to recompile the application to use the larger lock structure. Because one of our underlying goals is to develop techniques that can be used to monitor unmodified programs, we consider such an option an approach of last resort. Of course, one could modify a system's standard threading library to use the extended representation for a lock; however, such an approach would not be portable.

A third approach is to allocate additional state associated with a lock in out-of-band data. A benefit of this approach over the in-band approach is that it is a more flexible solution; for example, additional monitoring state can easily be added. We implemented this approach.

Allocating out-of-band state

We now consider when to allocate this additional out-of-band state. At first glance, it might appear straightforward to allocate the out-of-band state when a lock is initialized with `pthread_{mutex,spin}_init`. This would be attractive since one could assume a race-free context. However, this approach is fraught with difficulty. First, while it is possible to override every instance of a Pthreads call, some of these overrides may occur in contexts in which a profiler cannot manage the out-of-band state. For example, Pthreads locks are often used very early during execution within `glibc` and during initialization of shared libraries and static constructors.

Second, supporting out-of-band lock state requires managing dynamic allocation and deallocation of state instances. In many programs, components of dynamic data structures are decorated with locks (e.g., nodes in a tree). In such cases, a lock is destroyed when a node is freed; thus, managing the destruction of lock state is an essential part of an overall strategy for dynamic allocation. This shows that allocating out-of-band state for monitoring locks at the time of lock initialization requires the ability to dynamically allocate lock state and manage a per-thread free list[5] to which lock states could be appended when they are no longer needed. (Similarly, locks may be used after the application exits and monitoring tool shuts down but before the

[5]Using a per-thread free list avoids contention for the free list.

process has completely retired.) Providing both of these capabilities very early in an execution before the profiler is initialized is problematic.

Therefore, the shared lock state must in general be created on demand, i.e., when the performance tool first sees an attempt at locking (which may be different than the first attempt at locking). This implies the state is created in a context where other lock operations might be executed concurrently.

Accessing out-of-band state

On each call to a Pthreads locking routine, it is necessary to obtain the associated out-of-band state. There are two possibilities for accessing this data. The first option is to replace the contents of the lock itself with a pointer to a *monitored lock*. The second option is to write a function to quickly map between a pointer to a lock (which is unique) and its associated monitoring state.

The primary advantage of the first scheme is that finding a monitored lock can be an extremely fast constant-time operation. The primary disadvantage is that, because a performance tool might not see a lock's initialization, a native lock must be converted to a monitored lock within a race-sensitive context. For example, one thread may attempt to convert a lock into a monitored lock while that lock is currently held by a second thread and while a third thread is attempting to acquire that same lock. This implies that there must a concurrency protocol between the locking routines and the conversion routine.

The second option requires a data structure that supports both fast look ups and high concurrency. Because complex applications have a high rate of lock acquisitions, it is necessary to eschew coarse-grain locking. One potentially easy way to support high concurrency at the expense of extra memory is to make per-thread look ups faster by using an additional per-thread mapping data structure such as a splay tree. In other words, many look ups benefit from thread-local caches.

We initially tried the second approach because of its easier implementation. However, even using a local-global lookup to reduce contention on a centralized data structure — a balanced tree which itself used a sophisticated reader-writer lock — we were not satisfied with the resulting profiler overhead for programs that performed a high rate of lock acquisitions. Consequently, we developed protocols to support installing and managing monitored locks in a concurrent environment.

6.4.3 Dual-representation Locks

To support fast accesses to shared lock state and to sidestep a difficult refactoring of profiler initialization to enable out-of-band monitored lock states to be used very early during execution, we opted to use a dual representation for locks. In prior work, Bacon et al. used a dual representation for object locks in Java [19], though for different reasons. We discuss this in more detail in Section 6.6. Note that the algorithms presented below for managing dual-representation locks use the atomic primitives swap and CAS (compare-and-swap), which are defined in Appendix C.

Before profiler initialization, a lock is simply represented by a (32-bit) `pthread_spinlock_t`. Lock operations that occur before profiler initialization use this native lock representation. Once the profiler state is initialized, any lock, trylock, or unlock operation converts the native lock, in demand-driven fashion, to point to a monitored lock. The monitored lock includes the extra state needed to attribute contention. Once a lock has been converted into a monitored lock, it will remain a monitored lock until it is destroyed.[6] On each subsequent lock operation, the representation

[6]Bacon et al. use an analogous approach for Java locks. Once they inflate a Java lock to a "fat" out-of-band representation, the lock remains inflated for its remaining life.

Algorithm 6.2: demand-mon-lock: The protocol for converting a native lock into an out-of-band lock in demand-driven fashion.

```
typedef struct mon_lock { // a monitored lock
  pthread_spinlock_t lock; // typedef'd as "volatile int"
  long idleness;
} mon_lock_t;
mon_lock_t* demand_mon_lock(pthread_spinlock_t* lock) {
  if (!is_mon_lock(*lock)) {
    mon_lock_t* mlock = alloc_mon_lock();
    int newVal = make_mon_lock_ptr(mlock);
    bool didSwap = false;
    while (true) {
      int curVal = *lock;
      if (is_mon_lock(curVal)) break;
      mlock->lock = curVal;
      didSwap = (CAS(lock, curVal, newVal) == curVal);
      if (didSwap) break;
    }
    if (!didSwap) free_mon_lock(mlock);
  }
  return get_mon_lock(*lock);
}
```

is examined, the monitored lock is obtained, and the operation proceeds using the monitored representation.

After profiler initialization, all lock, trylock, or unlock operations request a native lock's monitored lock by calling demand_mon_lock, shown in Algorithm 6.2. If the lock already represents a monitored lock, the routine simply accesses the associated monitored lock by reinterpreting the bits of the native lock. If a native lock is not yet a monitored lock, then the routine initiates a protocol for converting the native lock (of type `pthread_spinlock_t`) into a monitored lock. The protocol first allocates a new monitored lock and computes a 'pointer' to install in the native lock.[7] Then, it

[7]A `pthread_spinlock_t` is 32 bits, even for 64-bit programs. In a program running in 64-bit mode, this is not long enough to contain a full pointer. To address this problem, we allocate a segment for locks. We represent a lock pointer in a `pthread_spinlock_t` as an offset from a base

Algorithm 6.3: lock-mon-lock: Lock a dual-representation lock.

```
 1 const int UNLOCKED = 1, LOCKED = 0;
 2 int pthread_spin_lock(pthread_spinlock_t* lock) {
 3   if (is_profiler_initialized) demand_mon_lock(lock);
 4   while (true) {
 5     if (is_mon_lock(*lock)) {
 6       // acquire a monitored lock
 7       mon_lock_t* mlock = get_mon_lock(*lock);
 8       lock = &mlock->lock;
 9       while (true) {
10         while (*lock == LOCKED);
11         if (swap(lock, LOCKED) == UNLOCKED)
12           return 0; // success
13       }
14     }
15     // acquire a native unmonitored lock
16     while (*lock == LOCKED);
17     if (CAS(lock, UNLOCKED, LOCKED) == UNLOCKED)
18       return 0; // success
19   }
20   return 1; // failure
21 }
```

enters the compare-and-swap (CAS) loop beginning on line 10. The loop obtains the current value of `lock` and ensures that since the test on line 6, `lock` is *still* a native lock. In that case, the protocol initializes a monitored lock with `lock`'s current value and attempts to atomically install a pointer to the monitored lock with the CAS on line 14. The loop exits when the CAS succeeds or some other thread converts the lock. If the latter occurs, the newly allocated monitored lock is reclaimed by placing it on a thread-local free list.

Algorithms 6.3–6.5 show the lock, trylock, and unlock protocols we use on these dual-representation locks. The algorithms are optimized for the typical case: a `pthread_spinlock_t` contains a pointer to a monitored lock.

address for the segment of monitored locks. For simplicity, in the rest of the chapter we omit the quotation marks around 'pointer.'

The lock operation shown in Algorithm 6.3 works as follows. First it tests to see if the native lock has been overlaid with a pointer to a monitored lock state (line 5). If so, it extracts the pointer and then attempts to acquire the lock with a simple test-and-test-and-set protocol. While the lock word of the monitored lock is found to be in the LOCKED state, it continues to spin (line 10). When this condition is no longer true, some other thread must have set the lock word to its UNLOCKED state. A swap operation is used to atomically set the value of the lock word to LOCKED and recover its prior value. If the lock was UNLOCKED when the swap occurred, the lock acquisition is complete and the protocol returns. Otherwise, another thread acquired the lock. In that case, the protocol returns to the spin-wait loop where it again delays until the lock word is no longer LOCKED.

If a lock operation initially finds that `lock` does not point to a monitored lock, it enters a protocol to acquire the lock using the native representation. As with acquisition of a monitored lock, the protocol enters a loop that spin-waits for the lock representation to no longer be in the LOCKED state (line 16). When attempting to acquire an unmonitored lock, there are two conditions that might cause one to exit this spin-wait: another thread may have set the lock word to unlocked, or the profiler may have been initialized and another thread may have exchanged the lock word representation to point to a monitored lock. If the lock is available and in the UNLOCKED state, the subsequent compare-and-swap (CAS) operation will find it in the UNLOCKED state, set it to LOCKED, and return that it was in the UNLOCKED state. At this point the protocol will terminate after successfully acquiring the lock using the native representation. It is noteworthy that at this point in the protocol, it is necessary to use a CAS rather than a swap as used in the protocol for monitored locks. The reason is simple: the representation may have changed since we last inspected the lock word. If the lock word has been promoted to a pointer, one cannot obliviously

Algorithm 6.4: trylock-mon-lock: Trylock on a dual-representation lock.

```
 1 int pthread_spin_trylock(pthread_spinlock_t* lock) {
 2   if (is_profiler_initialized) demand_mon_lock(lock);
 3   while (true) {
 4     if (is_mon_lock(*lock)) {
 5       // trylock a monitored lock
 6       mon_lock_t* mlock = get_mon_lock(*lock);
 7       lock = &mlock->lock;
 8       int prev = swap(lock, LOCKED);
 9       return ((prev == UNLOCKED) ? 0 /* success */ : 1 /* failure */);
10     }
11     // trylock a native unmonitored lock
12     int prev = CAS(lock, UNLOCKED, LOCKED);
13     if (prev == UNLOCKED)
14       return 0; // success
15     else if (prev == LOCKED)
16       return 1; // failure
17   }
18 }
```

overwrite it with LOCKED using a swap; instead, we conditionally overwrite it only if it is a native lock word in the UNLOCKED state. If the CAS fails, we return to the top of the outermost loop, check if the representation has changed, and execute the appropriate branch of the protocol to repeat the attempt to acquire the lock. An important feature of the protocol is that both the spin-wait and the CAS for the unmonitored lock representation can tolerate the representation being asynchronously switched to its monitored form. That would not be the case if line 16 read `while (*lock != UNLOCKED)` or line 17 used swap rather than CAS.

The trylock operation shown in Algorithm 6.4 similarly is designed to cope with our dual representation. If the lock word points to a monitored lock, it extracts the pointer and then attempts to acquire the lock with simple swap (line 8). Depending upon whether swap returns UNLOCKED, trylock succeeds or fails. Since a lock will never revert from a monitored lock pointer to a native representation until the lock is destroyed, if a lock is found to be using a monitored representation, it is safe to acquire it using a swap. If initially the lock word is not a pointer to out-of-band state, trylock attempts to acquire the lock in native form. In this case, the protocol uses a CAS operation (line 12) since the lock word may asynchronously change to a monitored lock pointer. If the lock word is still using the native representation (i.e., with value LOCKED or UNLOCKED), the trylock returns immediately with the appropriate result. If the representation was asynchronously converted to a monitored lock pointer, execution will continue at the top the `while` loop on line 3, enter the protocol to try to acquire a monitored lock, and complete in a few operations. Note that that although this protocol contains a `while` loop, the loop will execute at most two iterations, resulting in a fixed number of instructions and leaving the trylock protocol non-blocking.

While the use of CAS in these dual-representation protocols is potentially more costly than simply using a swap to try to acquire a native lock, or using a simple write to unlock, this will have little impact on the run-time cost of the locking protocol. These CAS operations execute only before profiler initialization. Since profiler initialization happens relatively early, in the typical case, the expected additional cost of the dual-representation in these protocols is limited to testing the lock word for a monitored lock pointer and converting that pointer into an actual pointer to the monitored lock.

The unlock operation shown in Algorithm 6.5 is quite similar to trylock in its handling for the dual representation. If the lock is found to point to a monitored lock, it simply sets the monitored lock's lock word to UNLOCKED. Otherwise, it attempts to unlock the lock by using a CAS (line 15) to update the lock word from LOCKED to UNLOCKED. If this fails, the lock must have been asynchronously converted to a monitored lock pointer. A second pass around the `while` loop (line 3) will release the

Algorithm 6.5: unlock-mon-lock: Unlock a dual-representation lock.

```
int pthread_spin_unlock(pthread_spinlock_t* lock) {
  if (is_profiler_initialized) demand_mon_lock(lock);
  while (true) {
    int lockval = *lock;
    if (is_mon_lock(lockval)) {
      // release a monitored lock
      mon_lock_t* mlock = get_mon_lock(lockval);
      if (mlock->lock == UNLOCKED) return 1; // failure
      else {
        mlock->lock = UNLOCKED;
        return 0; // success
      }
    }
    // release a native unmonitored lock
    if (CAS(lock, LOCKED, UNLOCKED) == LOCKED) return 0; // success
    if (*lock == UNLOCKED) return 1; // failure (prevent indefinite spinning)
  }
}
```

monitored lock. Although this protocol contains a `while` loop, the loop will execute at most two iterations, resulting in a fixed number of instructions and leaving the unlock protocol non-blocking.

6.4.4 Blocking (Sleep-waiting)

Recall that when Pthreads mutex locks sleep-wait, they receive no samples because samples are only delivered while threads are running. To implement blame shifting for sleep-waiting, we used a sampling strategy similar to that in Section 6.2.2. That is, on every n^{th} blocking call we time the thread's idleness and store it in the associated monitored lock's idleness counter. If the idleness count is non-zero when a thread releases the lock, it gathers the calling context. In principle this strategy should also work for condition variable waiting, but we have not implemented it.

6.4.5 Hints for Developers

Many subtle implementation issues arise when overriding various Pthreads library functions for profiling. For our profiling tools to be broadly applicable, each issue needs to be solved generically in a way that induces low run-time overhead. In some cases, the nature of interactions between target programs, run-time systems, and our profiler forced more complicated solutions than originally desired.

For instance, overriding `pthread_mutex_lock` and performing any non-trivial operation involves subtle complexities. Many operations in thread-safe run-time libraries, such as `malloc` or `dlsym`, directly or indirectly call `pthread_mutex_lock` in at least some circumstances. The former would commonly be used to allocate out-of-band memory for monitoring locks; the latter for preparing the override for `pthread_mutex_lock`. To allocate dynamic memory, we use `mmap`-ed regions. To prepare the `pthread_mutex_lock` override, we use the special symbol `__pthread_mutex_lock` exported in the Linux implementation of Pthreads.

Although only a subset of Pthreads functions need to be wrapped, care must be taken to prevent inconsistent versions. Problems of this sort come in two flavors. First, one might wrap a Pthreads function that sets values visible to other functions that are not wrapped. One must choose the set of functions to wrap carefully to ensure that all functions sharing data have a consistent notion of appropriate states. Second, intra-library calls have to see a consistent world. In particular, calls that use hidden interfaces within libraries that cannot be overridden must be handled.

Finally, most unwinders — including HPCToolkit's — are not designed to be recursive. Since our strategy uses both asynchronous-sampling-based call path profiling and synchronous unwinds of the call stack at a lock release point, it is important to specify what happens if an asynchronous sampling trigger occurs during a synchronous unwind. The simplest way to prevent interference is to prevent asynchronous samples during any unwind.

6.5 Case Studies

To show the effectiveness of blame shifting, we describe our experience applying it to three multithreaded applications with interesting locking and scheduling patterns. Our goal is to provide evidence that our method yields insight into non-trivial codes. In doing this, we distinguish between *obtaining* and *applying* insight. This is an important distinction because given an understanding of lock contention that includes a quantitative measure of the problem (insight), one might either resolve the problem or determine that a resolution is too costly (different applications). Because of the effort that would be involved in resolving the problems we identify, these studies focus on obtaining and not applying insight.

All experiments were performed on a Dell M905 blade running CentOS 5.2 and with four quad-core AMD 2.2 GHz Opterons (8354) and 48 GB main memory.

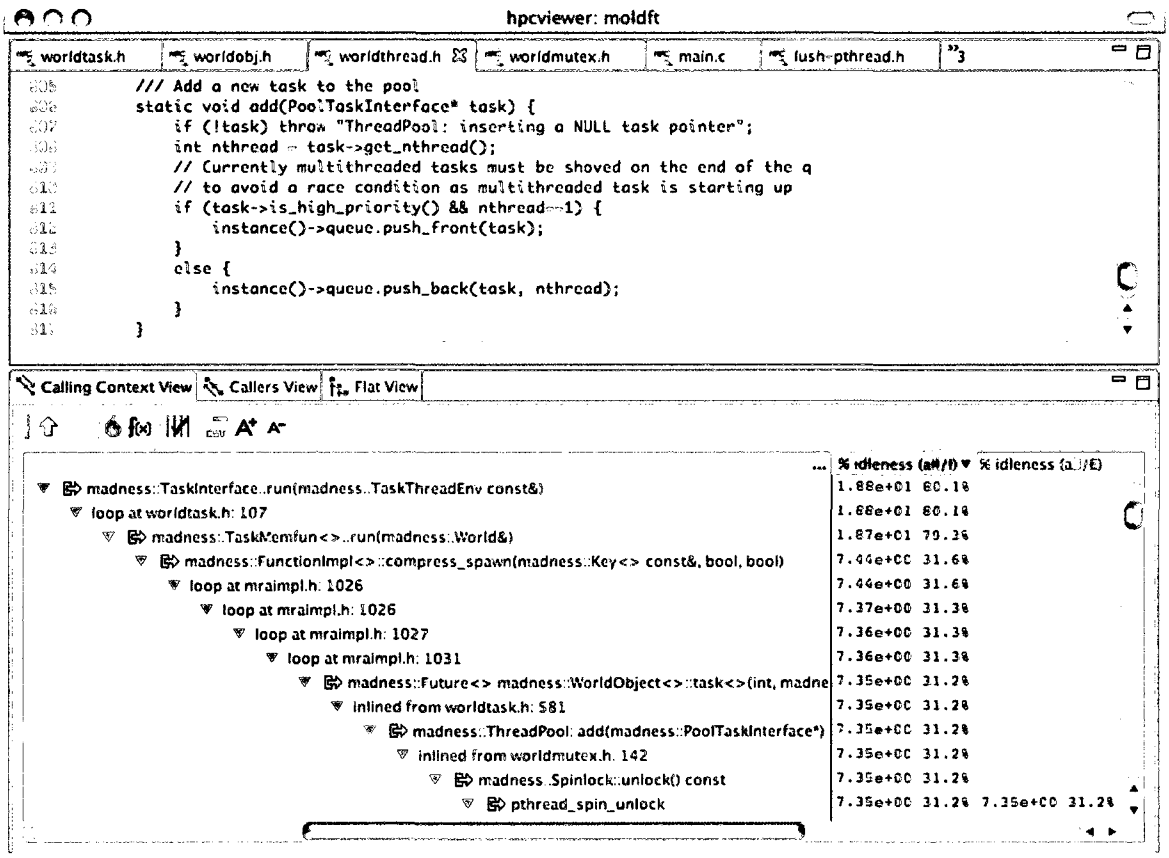
6.5.1 MADNESS

The first application we consider is MADNESS [69], a quantum chemistry application that makes extensive use of locking. MADNESS is designed to scale well both in SMP environments and on petascale clusters with multicore nodes. We focus on SMP executions here, but note that node-based performance is also critical for efficient performance on petascale clusters. MADNESS uses its own dynamic work scheduler based on a centralized queue. Worker threads create tasks (futures), which are pushed the queue. As necessary, workers pop tasks from the queue to obtain work. Among other things, MADNESS uses locks to manage access to the queue.

To obtain a sense of MADNESS's scaling losses, we gathered elapsed time for 4 and 16-core executions using the same input (strong scaling, averaged over five runs). While a 4-core run completed in 1150 seconds, a 16-core run took 516 seconds, an improvement of only a factor of 2.2. MADNESS' authors were aware of scaling losses but were unsure of the precise cause. Ignoring architectural concerns such as memory bandwidth, an obvious suspect is lock contention from managing a centralized task queue. However, it is not at all easy to show this for two reasons. First, understanding the different sources of lock contention in MADNESS is difficult because of its complex structure. Futures are implemented with templates. Typically, locks are implicitly acquired automatically through object creation and destruction. Furthermore, most critical sections are not straight-line code but a chain of templated method calls, heavily optimized by the compiler. Second, any monitoring tool must manage locks very efficiently to have low overhead for MADNESS. During a single 16-core execution of a non-trivial input, MADNESS used 65M distinct locks, had a maximum of 340K live locks, and performed an average of 30K lock acquisitions per second per thread. Finally, it is worth noting that MADNESS's authors had already spent considerable time experimenting with different implementation parameters.

We used our blame shifting strategy to measure lock contention on a version of MADNESS using spin locks. We used a sampling period of 5 ms to yield an average sampling rate of 200 samples/second. Curiously, during profiling, the execution time actually slightly *decreased* from 516s to 508s (averaged over 5 runs with no significant variability). We are not sure of the precise reason but note that this is an anomaly. Typically, our profiling overhead is positive, but less than 5%.

Figure 6.1 presents one view of the aggregated results displayed by our presentation tool. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, zoomed to focus on



```
/// Add a new task to the pool
static void add(PoolTaskInterface* task) {
    if (!task) throw "ThreadPool: inserting a NULL task pointer";
    int nthread = task->get_nthread();
    // Currently multithreaded tasks must be shoved on the end of the q
    // to avoid a race condition as multithreaded task is starting up
    if (task->is_high_priority() && nthread==1) {
        instance()->queue.push_front(task);
    }
    else {
        instance()->queue.push_back(task, nthread);
    }
}
```

```
                                                                              % idleness (all/I)  % idleness (all/E)
madness::TaskInterface::run(madness::TaskThreadEnv const&)                     1.88e+01 80.1%
  loop at worldtask.h: 107                                                     1.88e+01 80.1%
    madness::TaskMemfun<>::run(madness::World&)                                1.87e+01 79.3%
      madness::FunctionImpl<>::compress_spawn(madness::Key<> const&, bool, bool) 7.44e+00 31.6%
        loop at mraimpl.h: 1026                                                7.44e+00 31.6%
          loop at mraimpl.h: 1026                                              7.37e+00 31.3%
            loop at mraimpl.h: 1027                                            7.36e+00 31.3%
              loop at mraimpl.h: 1031                                          7.36e+00 31.3%
                madness::Future<> madness::WorldObject<>::task<>(int, madne    7.35e+00 31.2%
                  inlined from worldtask.h: 581                                7.35e+00 31.2%
                    madness::ThreadPool::add(madness::PoolTaskInterface*)      7.35e+00 31.2%
                      inlined from worldmutex.h: 142                           7.35e+00 31.2%
                        madness::Spinlock::unlock() const                      7.35e+00 31.2%
                          pthread_spin_unlock                                  7.35e+00 31.2%  7.35e+00 31.2%
```

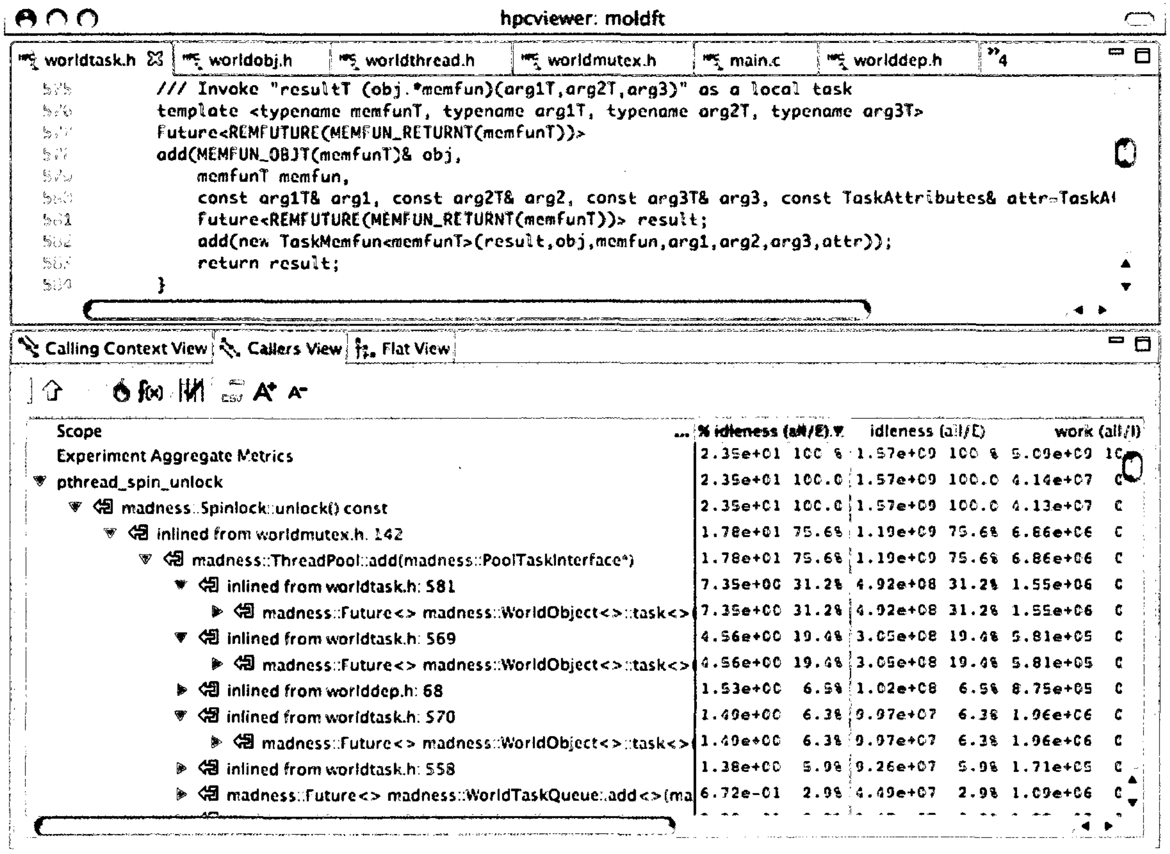
Figure 6.1: `hpcviewer`'s Calling Context view of MADNESS's `moldft`.

a portion of one call path. The call path is actually a fusion of dynamic calling contexts and the static context information such as loops and inlined frames. The selected line in the navigation pane highlights an instance of `ThreadPool::add` whose corresponding source code is shown in the source pane (top sub-pane). Each entry in the navigation pane is associated with metric values in the metric pane to the right. Two metrics are visible: '% idleness (all/I)' and '% idleness (all/E).' Both metrics represent idleness as a percentage of total effort (giving the '%' qualifier) and summed over all threads (yielding the 'all' qualifier). (Recall that effort is the sum of work and idleness.) The former metric shows *inclusive* ('I') values, or values that are inclusive of an entry's children. The latter shows *exclusive* ('E') values that exclude its children. In the metric columns, metric values are shown in scientific notation. Note that because these particular metrics are percentages, the values in scientific notation are actually percents. The values formatted as percentages on the right side of a column give an entry's proportion of the total idleness (as opposed to total effort).

The call path in the navigation pane is the hot call path with respect to the former metric and was expanded automatically. It is actually a fusion of dynamic calling contexts and static contextual information such as loops and inlined frames. The highlighted line in the navigation pane of Figure 6.1 indicates that 7.35% (scientific notation) of the total effort of the execution was spent in idleness at this context. Three lines below, we see the call to `pthread_spin_unlock`, exactly where blame shifting attributed the idleness due to lock contention. Within this call, both the inclusive and exclusive idleness metrics are identical, indicating that the call to `pthread_spin_unlock` accounts for all the idleness in this context.

This call path shows that there is lock contention associated with adding tasks to the centralized thread queue via `ThreadPool::add`. However, the remaining 68.8% of the idleness arises in other calling contexts. To avoid the need to search for other contexts in which there may be lock contention caused by `ThreadPool::add`, we turn to a bottom-up Callers view in Figure 6.2. If the top-down view looks down the call chain, the bottom-up view looks up to a procedure's callers. At the first level, the bottom-up view lists all the procedures in the program, rank-ordered according to the selected metric. Bottom-up metrics are computed by apportioning the costs of a procedure on behalf of its various calling contexts.

The first thing we observe is the very top line which gives aggregate values for the various metrics. (This line was not visible in Figure 6.1 because of scrolling.) We immediately see from the column labeled '% idleness (all/E)' that 23.5% of the execution's total effort consisted of lock contention. The column labeled 'idleness (all/E)' gives the absolute value of idleness (in microseconds): $1.57 \times 10^9 \mu s$. We



```
/// Invoke "resultT (obj.*memfun)(arg1T,arg2T,arg3)" as a local task
template <typename memfunT, typename arg1T, typename arg2T, typename arg3T>
Future<REMFUTURE(MEMFUN_RETURNT(memfunT))>
add(MEMFUN_OBJT(memfunT)& obj,
    memfunT memfun,
    const arg1T& arg1, const arg2T& arg2, const arg3T& arg3, const TaskAttributes& attr=TaskA
    Future<REMFUTURE(MEMFUN_RETURNT(memfunT))> result;
    add(new TaskMemfun<memfunT>(result,obj,memfun,arg1,arg2,arg3,attr));
    return result;
}
```

```
Scope                                                        % idleness (all/E)  idleness (all/E)    work (all/I)
Experiment Aggregate Metrics                                 2.35e+01 100 %      1.57e+09 100 %      5.09e+09 1C
pthread_spin_unlock                                          2.35e+01 100.0      1.57e+09 100.0      4.14e+07 C
  madness::Spinlock::unlock() const                          2.35e+01 100.0      1.57e+09 100.0      4.13e+07 C
    inlined from worldmutex.h: 142                           1.78e+01 75.6%      1.19e+09 75.6%      6.86e+06 C
      madness::ThreadPool::add(madness::PoolTaskInterface*)  1.78e+01 75.6%      1.19e+09 75.6%      6.86e+06 C
        inlined from worldtask.h: 581                        7.35e+00 31.2%      4.92e+08 31.2%      1.55e+06 C
          madness::Future<> madness::WorldObject<>::task<>   7.35e+00 31.2%      4.92e+08 31.2%      1.55e+06 C
        inlined from worldtask.h: 569                        4.56e+00 19.4%      3.05e+08 19.4%      5.81e+05 C
          madness::Future<> madness::WorldObject<>::task<>   4.56e+00 19.4%      3.05e+08 19.4%      5.81e+05 C
        inlined from worlddep.h: 68                          1.53e+00  6.5%      1.02e+08  6.5%      8.75e+05 C
        inlined from worldtask.h: 570                        1.49e+00  6.3%      9.97e+07  6.3%      1.96e+06 C
          madness::Future<> madness::WorldObject<>::task<>   1.49e+00  6.3%      9.97e+07  6.3%      1.96e+06 C
        inlined from worldtask.h: 558                        1.38e+00  5.9%      9.26e+07  5.9%      1.71e+05 C
        madness::Future<> madness::WorldTaskQueue::add<>(ma  6.72e-01  2.9%      4.49e+07  2.9%      1.09e+06 C
```

Figure 6.2: `hpcviewer`'s Callers view of MADNESS's `moldft`.

should note that this value does not reflect all the idleness in the program. Because Pthreads does not provide a spin-based condition variable, MADNESS implements its own. In principle, we could instrument MADNESS itself. Since this is not the point of our work, our MADNESS results only measure regular lock contention and ignore any waiting at a condition variable critical section. However, we obtain an accurate measure of Pthreads spin lock contention.

When we automatically expand the hot path relative to the metric '% idleness (all/E)', we see something similar to the screen shot in Figure 6.2. This view shows how all the idleness attributed to `pthread_spin_unlock` is apportioned to its callers (in their context). Just above the selected line in the navigation pane is `ThreadPool::add`. Its associated idleness metrics show that it is responsible for 75.6% of the locking contention, accounting for 17.7% of the execution's total effort. This line not only confirms that adding tasks to a centralized queue is problematic, but quantifies its effect on idleness.

To see the effects of lock contention by context, we look up the call chain to the callers of `ThreadPool::add`. The selected line and its siblings (some of which are not shown) lists those callers (for this particular *callee* context). Since sibling entries in the navigation pane are sorted relative to their exclusive idleness (the selected metric), we can easily examine the handful of important ones. Doing this shows that most of the locking contention (67.5% of the total idleness) derives from creating `Futures`. The idleness costs are spread across distinct templates — *not* distinct instantiations — that manage `Futures` with different numbers of arguments. The selected line shows the templated add function for a `Future` with three arguments. An approach using distributed work queues and work stealing would likely significantly reduce lock contention.

Our original scaling experiment shows that we have not accounted for all scaling losses. There are at least two sources. First, the fact that memory bandwidth does not scale linearly with the number of cores is likely to be a factor. Second, besides missing idleness due to condition variable waiting, we cannot effectively monitor the non-idle overhead of creating and managing tasks. In Chapter 5, we precisely computed overhead values for Cilk by modifying the Cilk compiler to distinguish between useful work and parallel overhead. While we have adopted this approach to identify the non-idle overhead of Pthreads routines, that overhead is negligible. The approach does not directly translate to MADNESS where there is no formal separation between the task management and the user code.

In hindsight, it is not surprising that a centralized queue protected by locks could introduce lock contention. However, it would be an error to conclude that these results are trivial. To see this, consider the question of how severe lock contention is on 8 cores. It turns out that the total lock contention on 8 cores is 1–2% because MADNESS' developers had optimized for this case. However, MADNESS' developers had no clear answers to questions like: How severe is lock contention for a particular execution? Do these executions fail to scale because of lock contention or some other reason? Is lock contention occurring primarily at the centralized queue or is it more evenly spread among other lock acquisitions? Our results help answer these questions.

6.5.2 UTS

The second case study is a Pthreads implementation of the Unbalanced Tree Search (UTS) benchmark [112]. UTS was designed to evaluate the performance and ease of programming parallel applications that require dynamic load balancing. UTS builds and searches trees where each vertex unpredictably either has no children or millions of descendants. The number of active vertices varies between a few and tens-of-thousands during the execution (depending on the starting parameters and current depth).

UTS uses a work-stealing scheduler where each worker thread maintains a queue with two pieces, a local section that can be accessed without locks and a shared portion from which work can be stolen and which is protected by locks. A lock is acquired when work is moved from the local to the shared portion of a queue.

We profiled UTS and examined the resulting work and idleness metrics (microseconds) aggregated across all 16 threads. It was immediately apparent that although all cores were busy throughout the execution, they were only doing useful work about 40% of the time. With the idleness metric, we immediately pinpointed the source of idleness to contention for locks protecting the shared queues. About 72% of the idleness derived from contexts where new 'stealable' work was pushed onto the shared queues. Almost all of the remaining idleness (27.5%) was attributed to successful steals of work by otherwise idle threads. Thus, a majority of this execution time was spent contending for the privilege of either providing or extracting work. One way to reduce this contention is to use larger granularity tasks.

6.5.3 SSCA #2

The last case study is from the Scalable Synthetic Compact Application (SSCA) benchmark suite [46]. SSCA #2 was designed to be a hard-to-parallelize, compute-intensive analysis program that stresses memory access using integer and character operations.

We profiled an implementation of SSCA #2 using Pthreads written by Bader and Madduri [20]. Interestingly, idleness is very unevenly distributed across threads. In particular, 99.9% of the idleness of the first thread derives from a coarse-grained lock protecting an update to the graph. Having one lock per graph vertex rather than one graph-wide lock would reduce contention for that critical section and could greatly speed the initialization phase. The post-initialization compute kernels contained no significant sources of lock contention.

6.6 Related Work

Performance Tools

Intel's Thread Profiler [26] (for Windows) has two ways to analyze multithreaded performance. First, it provides a measure of a routine's effective parallelism, a useful metric that is similar to Quartz [12] and the strategy of Hansen et al. [68]. Second, and more related to our work, it instruments synchronization objects with timers to further classify a thread's execution by its effects on other threads. Thread Profiler makes use of this information to (1) qualify a thread's execution and (2) to highlight synchronization objects that accumulate blocking time. To classify a thread's execution, Thread Profiler distinguishes between interaction effects such as cruise, impact and blocking time. *Cruise time* is time that a thread does not delay the next thread on the critical path while *impact time* is the opposite. If a thread on the critical path waits for some external event, it accumulates *blocking time.* While this is useful information, it requires substantial overhead to collect.

To highlight synchronization objects, Thread Profiler reports how much time was spent waiting for a particular object and the utilization of the system during that wait time [36]. It also shows the creation calling context of the synchronization object. If locks are statically allocated and have long lifetimes, this information can be very effective. However, additional information is needed if there is no direct line of sight from idleness at the lock to the source of contention. For example, only certain threads may be responsible for contention, locks may be dynamically created and destroyed (e.g., linked data structures), or contention may be related to context. Our approach is superior to that of Thread Profiler in two ways. First, we 'blame' lock contention on the offending thread's context rather than aggregating wait time at a synchronization object; this directs an analyst to the source of the problem. Second, our approach is able to deliver this insight with very low monitoring overhead ($< 5\%$).

Several current tools detect lock contention in Java. IBM's Lock Analyzer for Java [73] computes a metric that reflects the number of delayed lock acquisitions as a percentage of total lock acquisitions. Sun's JConsole [40] helps identify contention by timing idle and by counting the number of delayed lock acquisitions. Like Intel's Thread Profiler, these tools attribute these metrics to locks themselves rather than to calling contexts. Also, while these tools might be effective for programs with statically allocated and long-lived locks, they do not provide enough information to diagnose problems in applications with a large number of dynamically created and destroyed locks.

Dual-representation Locks

Bacon et al. use a dual representation for object locks in Java [19]. They use a 24-bit field in a Java object's header to implement a 'thin lock' for objects that (a) are not subject to contention, (b) do not have wait, notify, or notifyAll operations performed upon them, and (c) are not locked to a nesting depth of more than 255. Objects that do not meet these criteria have their locks implemented as out-of-band "fat" locks. As with our scheme, once locks are converted to an out-of-band representation, they remain in that state. Bacon et al. avoid the need for a compare-and-swap in unlock because in their protocol, once a thread acquires a lock, no other thread may modify the lock word. In our approach, a lock may be changed to its out-of-band representation at any time. Without this, we would be unable to attribute contention to any lock that was acquired before profiling was initiated.

Contention Managers for STM

In our work, we use auxiliary state associated with a lock to blame idleness resulting from contention for that lock on the lock holder and attribute the idleness to the calling context of the lock holder's unlock operation. Some contention managers for Software Transactional Memory (STM) use auxiliary state associated with transactional objects to notice and manage contention on the fly. For instance, the Eruption contention manager by Scherer and Scott [125] uses data associated with transactional objects not only to observe contention, but also to transfer priority from a blocked transaction to the transaction it is blocked behind. At an abstract level, both our profiler and the Eruption contention manager use state associated with synchronization objects to communicate information about contention between competing threads.

Hardware Support for Attributing Stalls Due to Contention

The Alpha 21264's ProfileMe hardware support for instruction-based sampling [47] measures and quantifies the impact of contention for registers or execution units by measuring stalls while waiting for resources. While ProfileMe identifies contention and quantifies its impact, it attributes stall cycles to the victim of a stall rather than the instruction on which it is waiting. This strategy of attributing contention to waiting instructions is similar in effect to the strategy we describe in Section 6.2, which directly attributes contention to waiting threads.

6.7 Discussion

Being able to quantify and attribute lock contention is important for understanding where a multithreaded program needs improvement.

We described three different approaches for quantifying lock contention that progressed from (1) attributing a thread's idleness to itself in the context in which it is idling (the victim); (2) then to the set of threads holding locks at the time (the suspects); and finally (3) to the thread holding the target lock (the perpetrator). Three underlying principles drove the development of our final blame shifting strategy. First, we strove to obtain a high degree of precision and detail in our measurements. Second, rather than sacrificing high overhead to obtain high precision, we developed extremely low overhead profiling methods. When using reasonable sampling frequencies (hundreds to thousands of samples/second), our overhead is typically $< 5\%$, even for an application that uses 65M distinct locks and an average of 30K lock acquisitions per second per thread. To prevent profiling itself from introducing serialization, we used a minimal amount of shared state and accessed it very rapidly. By using a sampling-based profiler that recovers call paths by unwinding a call stack, we were able to attribute idleness to its full static and dynamic context while maintaining extremely low overhead. We also used a form of sampling to amortize the cost of heavyweight instrumentation. Third, our aim was to develop a general method that enables tools to monitor unmodified programs. Doing this required solving subtle but complex problems such as how to maintain a dual-representation lock.

For future work, we would like to increase the precision of our results by recording the number of lock operations *within its calling context.* This would allow us to distinguish between a few highly contested long waits and many moderately contested short waits. A low-overhead way of doing this is by collecting return counts from sampled frames [60].

Our profiler is based on the general principle of using shared state to communicate information about performance losses due to resource contention between competitors. While in this chapter we apply this principle to attribute spin-waiting for a lock back to the calling context of the lock holder, we can imagine using variants of our strategy for other purposes. As one example, this same strategy could be used for reporting lock contention in multithreaded languages that provide locks such as Cilk. As another, in a lock-based software transactional memory system, transactions acquire locks associated with objects that they wish to modify transactionally. When another transaction needs an object that is already locked, a contention manager is invoked to decide which transaction to abort. Rather than just using using auxiliary object state to communicate information about contention and guide a contention manager's handling of competing transactional operations, our profiler could augment a transactional object with information that would enable us to attribute contention back to the transaction that holds an object lock and the calling context of the transaction.

Chapter 7

Analysis & Presentation of Petascale Executions

7.1 Introduction

A wide range of scientific applications require petascale computing to address problems at the frontier of computational science research. In 2009, the first petascale systems became available. Two of the most powerful 'leadership computing platforms' available for open science in the United States are Jaguar, a Cray XT4/XT5 at the National Center for Computational Sciences and Intrepid, an IBM BlueGene/P at the Argonne Leadership Computing Facility. Each system contains over 160,000 processor cores. Tackling grand challenge problems requires using such platforms effectively, which requires addressing two issues. First, an application must scale efficiently to large processor counts. Second, an application must make efficient use of individual processor nodes.

If an application contains significant scaling bottlenecks, it cannot productively use the large number of cores in leadership computing platforms. Unfortunately, it is extremely difficult for applications to effectively use computing resources at this scale because seemingly benign inefficiencies emerge as major bottlenecks on a large number of processors. Understanding why an application does not scale can be quite difficult. To date, approaches to analyze scalability on petascale systems have required laborious human effort [4–6, 72, 111, 152], used instrumentation-based measurement techniques that can significantly dilate execution time and distort the nature of performance measurements [48,74,129,151], or provide only qualitative information [149]. Moreover, at best these approaches only identify scaling bottlenecks at the procedure level because detailed instrumentation at a finer level (e.g., loops) is too costly. As a result, there is a critical need for better tools that can accurately measure and attribute performance information in ways that enable scientists to understand in detail how impediments to scaling arise in parallel applications. Without detailed information about where scaling losses occur, addressing their underlying causes can be difficult.

If an application loses a factor of two in node performance, that halves the amount of science that can be accomplished with a fixed allocation on a leadership computing platform. Understanding node performance inefficiencies in applications at full scale may require measuring performance at scale because it may be difficult to recreate the same conditions for study on a smaller number of processors.

The HPCToolkit project has developed low-overhead techniques for sampling-based performance measurement and analysis that make it possible to precisely quantify and attribute both scalability losses and node performance losses. HPCToolkit can attribute both kinds of losses to individual lines of source code, in their full static and dynamic contexts [41,141]. However, HPCToolkit's analysis relies on the accurate collection of precise performance measurements. Petascale platforms present two principal challenges to collecting such measurements.

Scale

The first challenge is that of scale. When analyzing data from many cores, reliance on serial algorithms is likely to be problematic. Also, one must take care to ensure that measurement approaches do not overly tax shared system resources, e.g., the network or file system. For instance, a measurement approach based on *tracing*, where performance information is distinguished by time, faces significant challenges at scale. Collecting traces at scale can burden file systems and interfere with application and system performance. Even with careful design, trace files can quickly become terabytes in size [154]. Some of these challenges are addressed by on-line data compression, but at the expense of coarser measurements [63]. Another approach for reducing trace data volume uses sampling to monitor only certain processes within the execution [64]. In our work, we avoid the problems of tracing by focusing on *profiling*. Since profiling collapses the time dimension of measurements, it more readily scales to long-running large-scale executions.

There are different ways to profile. A profiler that uses instrumentation — whether source code [129, 151], compiler-inserted [66, 129], static binary [48, 74], or dynamic binary [99] — can introduce significant measurement overhead in programs with small procedures. For instance, a previous study [60] showed that simple instrumentation for the Gprof [66] profiler introduced overhead with a geometric mean of 93% when monitoring the SPEC CPU2000 [136] integer benchmarks. The TAU performance tools [129] reduce instrumentation overhead at the expense of detail through the use of throttling and selective instrumentation [128]. However, selective instrumentation can be problematic because it introduces blind spots, often in critical places such as small, frequently executed routines that lie on hot paths. The alternative to instrumentation is asynchronous statistical sampling. With an appropriate choice of sampling frequency, sampling-based tools can deliver precise measurements with little overhead. The HPCToolkit performance tools use event-based sampling in combination with call stack unwinding to collect detailed call path profiles; experiments with the SPEC CPU2006 [134] benchmarks show that HPCToolkit's measurement overhead is only a few percent for reasonable sampling rates [141]. Sampling-based call path profiling is scalable because a call path profile does not grow with the number of samples, but only with the number of unique call paths observed during the samples.

Since HPCToolkit collects per-thread call path profiles, it must scalably analyze and present those measurements. To support performance analysis of large-scale executions, we have created a parallel version of HPCToolkit's analysis tool that scalably generates a database that can be scalably presented by HPCToolkit's presentation tool.

Microkernels

The second challenge that petascale systems had posed for measurement was that their compute node microkernels made asynchronous-sampling-based measurement impossible, in part because of a concern about unnecessary features within standard operating systems. Petrini et al. showed that for large systems, asynchronous operating system activity, such as periodically monitoring I/O, could cause serious performance problems [72,113]. As a result, minimizing interrupts to avoid operating system 'jitter' was a critical concern when designing the Catamount microkernel for the Cray XT3 [6]. As a side effect, it was not possible to use asynchronous sampling as a measurement approach on Catamount until we interceded with its developers at Sandia National Laboratory. In modern compute node kernels for the Cray XT and Blue Gene/P, the intent of their developers was to provide kernel support for sampling; however, before we exercised this capability with HPCToolkit, this support was non-functional in both kernels. In 2008, we engaged kernel developers at IBM and Cray to address the shortcomings of their implementations and in early 2009, kernel versions with working support for sampling were released and installed on the DOE's leadership computing platforms.

Our Approach

To support asynchronous-sampling-based call path profiling on emerging petascale platforms, including x86-64-based systems running Linux (e.g., the Ranger system at the University of Texas), x86-64-based Cray XT systems running Compute Node Linux, and PowerPC-based Blue Gene/P systems running IBM's compute node kernel, we added several new capabilities to HPCToolkit.[1] These capabilities include (1) technology for monitoring processes, threads, and dynamic loading; (2) on-the-fly binary analysis to support call path profiling of optimized and partially stripped executables; and (3) support for injecting a monitoring library into a statically linked executable. While support for statically linked binaries is needed for the Cray XT and Blue Gene/P platforms, support for dynamically loaded shared libraries is needed for dynamically linked binaries, which are typical on clusters that run more full-featured Linux kernels, e.g., the University of Texas's Ranger.

To support scalable analysis and presentation of call path measurements from petascale executions, we developed `hpcprof-mpi`, a parallel version of HPCToolkit's `hpcprof` tool. When given all per-thread measurements for a large-scale execution, `hpcprof-mpi` does two things. To scalably analyze and attribute measurements to source code, `hpcprof-mpi` creates a canonical call path profile that summarizes a whole execution. Then, to facilitate scalable presentation, it generates a database of thread-level data correlated with the canonical call path profile. The database is designed so that HPCToolkit's presentation tool `hpcviewer` can scalably present the summary data.

To scalably analyze and present HPCToolkit's petascale measurements using `hpcprof-mpi` and `hpcviewer`, we present solutions to three key problems:

[1]The Acknowledgments section recognizes the contributions of collaborators.

- We formally define `hpcviewer`'s three views: Calling Context, Callers and Flat. Earlier informal definitions were not fully correct for recursive programs.

- We show how to scalably compute summary metrics for the Calling Context view based on all thread-level profiles from a large-scale execution. Rather than assuming that all thread-level inputs are simultaneously available and fit within memory, we show how to create summary metrics by partitioning the thread-level inputs into chunks that can be processed in parallel — even when a summary metric relies on non-commutative and non-associative operators such as the square root in the formula for standard deviation.

- To generate as small a database as possible, we define the Callers and Flat views only in terms of a Calling Context view with summary metrics. This means that `hpcviewer` can compute its Callers and Flat views using only the Calling Context view — even when the Calling Context view only contains summary metrics defined with non-commutative and non-associative operators.

This chapter shows that it is possible, for little measurement overhead, to identify and quantify both scaling and node performance bottlenecks on petascale systems. Using asynchronous-sampling-based call path profiling, we show that HPCToolkit provides extremely detailed information about the performance of several emerging petascale applications on Cray XT and IBM BlueGene/P systems. Our tools pinpoint performance bottlenecks to source code lines, in their full static and dynamic context. Our analyses are rapid and their results are actionable. The effectiveness of our approach and our tools provides an argument that asynchronous sampling support is so beneficial that it should be included within microkernels for future extreme-scale systems.

The rest of this chapter is organized as follows. Section 7.2 describes HPCToolkit's approach to measurement, analysis and presentation and shows how it enables costs, including scalability bottlenecks, to be attributed to their full static and dynamic contexts. In Section 7.3, we use HPCToolkit to analyze the scaling of several applications slated for use on petascale systems. Section 7.4 compares our approach with related work and Section 7.5 discusses the chapter's main themes.

7.2 Scalable Measurement, Analysis and Presentation

This section explains HPCToolkit's measurement, analysis and presentation capabilities for scalably pinpointing and quantifying scalability bottlenecks.

7.2.1 Pinpointing Scaling Losses Using Call Path Profiling

To pinpoint scaling losses, we use call path profiling. HPCToolkit's sampling-based call path profiler, `hpcrun`, attributes execution costs of optimized executables to the full calling context in which they occur. To attribute metrics back to source code, HPCToolkit combines a call path profile with program structure information reconstructed by a post-mortem analysis of an application's object code and its debugging sections. Using this information, HPCToolkit attributes metrics to dynamic call paths fused with static context such as loops and inlined functions. HPCToolkit's measurement approach scales well to large executions because it is distributed (thread-based) and because profiles grow slowly over time. In particular, profiles grow only with the number of new calling contexts revealed on each sample (cf. Chapter 3).

To pinpoint and quantify scalability bottlenecks *in context*, we compute a metric that quantifies scaling loss by scaling and differencing call path profiles from a pair of executions [41].

Consider two parallel executions of an application, one executed on p processors and the second executed on $q > p$ processors. In a weak scaling scenario, processors in each execution compute on the same size data. If the application exhibits perfect weak scaling, then the execution times should be identical on both q and p processors. In fact, if every part of the application scales uniformly, then this equality should hold in *each scope* of the application.

Using `hpcrun`, we collect call path profiles on each of p and q processors to measure the cost associated with each calling context in each execution. `hpcrun` uses a data structure called a *calling context tree* (CCT) to record a call path profile. Each node in a CCT is identified by a code address. In a CCT, the path from any node to the root represents a calling context. Each node has a weight $w \geq 0$ indicating the exclusive cost attributed to the path from that node to the root. Given a pair of CCTs, one collected on p processors and another collected on q processors, with perfect weak scaling, the cost attributed to all pairs of corresponding CCT nodes[2] should be identical. Any additional cost for a CCT node on q processors when compared to its counterpart in a CCT for an an execution on p processors represents *excess work*. This process is shown pictorially in Figure 7.1. The fraction of excess work, i.e., the amount of excess work in a calling context in a q process execution divided by the total amount of work in a q process execution represents the scalability loss attributed to that calling context. By scaling the costs attributed in a CCT before differencing them to compute excess work, one can also use this strategy to pinpoint and quantify

[2]A node i in one CCT corresponds to a node j in a different CCT if the sequence of nodes along the path from i to root and the sequence of nodes from j to root are labeled with the same sequence of code addresses.

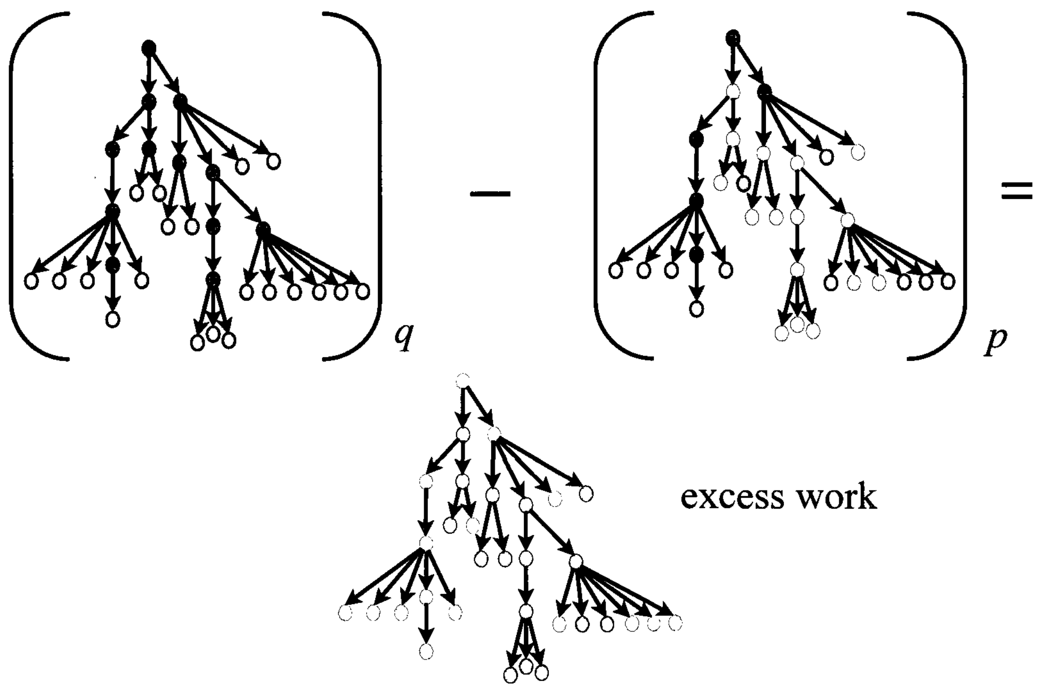


Figure 7.1: A pictorial representation of differencing call path profiles to pinpoint (weak) scaling bottlenecks.

strong scalability losses [41]. As long as the CCT's are expected to be similar, this analysis strategy is independent of the programming model and bottleneck cause.

Above, we described applying our scalability analysis technique *across* nodes in a cluster. This technique can also be used to pinpoint scaling bottlenecks within multicore nodes. For instance, one might want to understand how performance scales when using all of the cores in a node with multicore processors instead of just a single core. This can be accomplished by measuring an execution on a single core, measuring an execution on all cores, and then comparing the costs incurred by a core in each of the executions using the strategy described above for analysis of weak scaling. We have used this strategy to pinpoint and quantify scaling bottlenecks on multicore nodes at the loop level [138]. Measurements of L2 cache misses showed that contention in the memory hierarchy was the problem.

Our analysis is able to distinguish between different causes. For example, an analysis using standard time-based sampling is sufficient to precisely distinguish MPI communication bottlenecks from computational bottlenecks. With hardware performance counters, one can distinguish between different architectural bottlenecks such as floating point pipeline stalls, memory bandwidth, or memory latency.

7.2.2 Analyzing & Presenting Large-Scale Executions

To apply the scalability analysis summarized in the prior section to a large-scale execution, it is necessary to have a canonical calling context tree (CCT) that summarizes all of the individual thread-level call path profiles within an execution. Creating a canonical CCT requires unioning each thread-level CCT in the execution so that a context appears in the canonical CCT if and only if it appears in any thread-level CCT. If an execution contains n_T threads and there are n_m metrics per thread, then *each node* in that execution's canonical CCT will have $n_T \times n_m$ associated metric values. Because for large-scale executions n_T can be on the order of hundreds of thousands (currently) to millions (near future), it is neither reasonable to process each CCT sequentially nor feasible to store all thread-level metrics in memory. To handle large-scale measurements, it is critical that analysis and presentation itself be scalable.

To scalably analyze and present measurements from petascale executions, we developed `hpcprof-mpi`, a parallel version of `hpcprof`. Like `hpcprof`, `hpcprof-mpi` attributes measurements of executions to source code and creates a database that can be presented by `hpcviewer`. Unlike `hpcprof`, it is parallel and scalable. Additionally, it automatically creates metrics that summarize all per-thread CCT data. `hpcprof-mpi` is based on Single Program Multiple Data (SPMD) parallelism, which is implemented using MPI [98]. The high-level algorithm can be divided into three key phases. Assume an `hpcprof-mpi` job executes with P processes.

First, the master process divides the thread-level call path profiles into P groups, and assigns one group to each process. Each process is responsible for processing all the thread-level profiles assigned to it.

Second, `hpcprof-mpi` creates a canonical CCT that represents all of the thread-level CCTs' call path profiles. The canonical CCT contains a context — a path from a leaf to the root — if and only if it appears in any thread-level CCT. Although the canonical CCT represents a union of all the thread-level CCTs, in practice it does not grow linearly with the number of threads. The thread-level CCTs of SPMD scientific applications are often very similar. Even applications that model multiple physical systems usually do not induce more than a handful of distinct CCT groups, and the groups themselves have commonality between them. This means that with respect to its structure, we expect the canonical CCT to be no more than a small constant factor larger than the average thread-level CCT. To create the canonical CCT, `hpcprof-mpi` performs a parallel (tree-based) reduction on the thread-level CCTs. (Metric data is excluded from this reduction.) Then it uses a parallel (tree-based) broadcast to return the canonical CCT to each process. After this step is completed, each MPI process contains a copy of the canonical CCT. `hpcprof-mpi` aligns each thread-level CCT with the canonical CCT so that the canonical CCT can serve as an index for both the new summary metrics as well as all of the thread-level metrics.

Third, `hpcprof-mpi` creates summary metrics. To do this, `hpcprof-mpi` determines a useful set of derived metrics such as minimum, maximum, sum, mean, and standard deviation that summarize thread-level metrics. To compute a derived metric for a given canonical CCT node x, it is necessary to use the thread-level metric values from the thread-level CCT nodes that correspond to x. Since `hpcprof-mpi` cannot depend on storing all thread-level inputs to the derived metric computation in memory simultaneously, it computes the derived metric *incrementally*, a process that is discussed in the next section.

7.2.3 Scalably Computing Metrics

HPCToolkit was originally designed to compute derived metrics given all thread-level data simultaneously stored in memory. For example, consider the case of computing the arithmetic mean for a particular node in an execution's canonical CCT. If the execution contained n_T threads, then the arithmetic mean $m(x)$ for a given node x would be $\frac{1}{n_T}\sum_{t=1}^{n_T} m(x,t)$, where $m(x,t)$ represents node x's metric value for thread t. However, as we have seen, because the number of threads in an execution can be very large, it not feasible to require that all thread-level metric values for each node in the CCT reside in memory simultaneously. This implies that the computation of $m(x)$ must be broken into multiple steps.

To address this problem, we have developed an approach to scalably compute derived metrics. We call the approach *incremental* because it tolerates receiving inputs one at a time rather than all at once. Linford et al. [87] observe that covariance can be computed incrementally. Our contribution is to formalize the technique and show how it can be applied to computing several kinds of metrics for scalably analyzing and presenting call path profiles. Specifically, we partition the thread-level inputs into chunks that can be as small as one. Any metric that can be expressed as a function of polynomials over thread-level data can be computed incrementally. This approach can also be used to compute the minimum and maximum, though it is insufficient to compute order statistics in general.

To compute a given metric m incrementally for a CCT node x, we divide the computation into four stages, as shown in Figure 7.2. The key stages are accumulate

Stage	Function Prototype		
initialize	$\bigcirc_i()$	:	$\mapsto$ accumulator$_i$
accumulate	$\bigodot_i(a, x)$	: accumulator$_i$ $\times$ input	$\mapsto$ accumulator$_i$
combine	$\bigoplus_i(a_p, a_q)$	: accumulator$_i$ $\times$ accumulator$_i$	$\mapsto$ accumulator$_i$
finalize	●$(\langle a_1, \ldots, a_{n_A}\rangle, n)$	: accumulator-list $\times$ input-size	$\mapsto$ output

Figure 7.2: Function prototypes for an incrementally computed metric with accumulators $a_1, \ldots, a_{n_A}$. Observe that there is one initialize, accumulate and combine function for each accumulator.

and finalize. During the *accumulation* phase, each thread-level input is revealed one by one and in no particular order. To compute $m(x)$, we isolate portions of the computation that use commutative and associative operators from portions that do not; the latter operations are saved for the finalization stage. We associate two things with each isolated portion of the computation: a piece of state called an accumulator and an accumulate function. Recall that to compute the arithmetic mean for a particular CCT node x, we use the formula $\frac{1}{n_T}\sum_{t=1}^{n_T} m(x,t)$. To compute this metric incrementally, we associate the sum $\sum_{t=1}^{n_T} m(x,t)$ with an accumulator $m^*(x)$ and postpone the division by n_T to the finalization stage. Thus, the accumulate function is $m^*(x) + m(x,t)$ and the finalize function is $m^*(x)/n_T$. Then for each input value, we update each accumulator using its respective accumulate function. For the arithmetic mean example, the one accumulator simply maintains a running sum of the input values. Consequently, when the arithmetic mean metric is given a new input $m(x,t)$, we update the accumulator using the following computation: $m^*(x) \Leftarrow m^*(x) + m(x,t)$.

During the *finalization* phase, a finalize function takes all of a metric's accumulators and the number of inputs and applies any additional operations that are required to obtain the metric's final value. For arithmetic mean, the finalize function computes the final value $m(x)$ for node x using the operation $m^*(x)/n_T$, where n_T is the number of thread-level inputs. Before application of the finalize function, we say

Algorithm 7.1: incrementally-compute-metrics: Incrementally compute derived metrics in parallel.

```
Input: Metric descriptor $\mathbf{M}$, which includes the following (see Figure 7.2):
       initialize functions $\bigcirc_1, \ldots, \bigcirc_{n_A}$; accumulate functions $\bigodot_1, \ldots, \bigodot_{n_A}$;
       combine functions $\bigoplus_1, \ldots, \bigoplus_{n_A}$; and a finalize function $\bullet$.
Input: Input metric values $\mathbf{X} = \langle x_1, x_2, \ldots, x_n \rangle$ (which may not fit in memory).
Input: $P$ processes.
Result: The metric value when metric descriptor $\mathbf{M}$ is applied to $\mathbf{X}$.
1  Divide $\mathbf{X}$ into $P$ groups $\mathbf{X}_1, \ldots, \mathbf{X}_P$
2  In parallel at process p:
3      let $\mathbf{A}_p = \langle a_1, \ldots, a_{n_A} \rangle$ be accumulators for metric $\mathbf{M}$, where each $a_i = \bigcirc_i()$
4      let $\mathbf{X}_p = \langle x_1, \ldots, x_{n_p} \rangle$ be the metric values assigned to $p$
5      foreach $x_i$ in $\mathbf{X}_p$ do
6          foreach $a_j$ in $\mathbf{A}_p$ do
7              $a_j \Leftarrow \bigodot_j(a_j, x_i)$
8  In parallel, reduce accumulators $\mathbf{A}_2, \ldots, \mathbf{A}_P$ into $\mathbf{A}_1$. To reduce $\mathbf{A}_p$ and $\mathbf{A}_q$ into $\mathbf{A}_p$:
9      foreach $\langle a_{p,i}, a_{q,i} \rangle$ in make-pairs($\mathbf{A}_p, \mathbf{A}_q$) do
10         $a_{p,i} \Leftarrow \bigoplus_i(a_{p,i}, a_{q,i})$
11 return $\bullet(\langle a_1, \ldots, a_{n_A} \rangle, n)$
```

that the accumulator $m^*(x)$ is non-finalized; afterwards, it is finalized. It is easy to see that, instead of simultaneously requiring storage for all input values, this method only requires simultaneous storage of one input value and a set of accumulators.

To parallelize the computation of metric m at node x, we divide the per-thread input into several chunks. Assume we have a metric that requires only one accumulator. Then, one accumulator is associated with each chunk. We task each process in a parallel job with accumulating all the input values within a particular chunk. Then we use the metric's *combine* function (Figure 7.2) to reduce the per-chunk accumulators back into one accumulator. The combine function takes two non-finalized accumulator values $m_i^*(x)$ and $m_j^*(x)$ and combines them into another non-finalized value. The combine function for arithmetic mean is simply $m_i^*(x) + m_j^*(x)$.

Algorithm 7.1 shows a parallel algorithm for computing a derived metric incrementally. The algorithm takes as input a metric descriptor that includes the functions

Sum	Mean
$\bigcirc() = 0$	$\bigcirc() = 0$
$\bigodot(a, x) = a + x$	$\bigodot(a, x) = a + x$
$\bigoplus(a_p, a_q) = a_p + a_q$	$\bigoplus(a_p, a_q) = a_p + a_q$
●$(\langle a \rangle, n) = a$	●$(\langle a \rangle, n) = a/n$

Minimum	Maximum
$\bigcirc() = \infty$	$\bigcirc() = -\infty$
$\bigodot(a, x) = \mathsf{min}(a, x)$	$\bigodot(a, x) = \mathsf{max}(a, x)$
$\bigoplus(a_p, a_q) = \mathsf{min}(a_p, a_q)$	$\bigoplus(a_p, a_q) = \mathsf{max}(a_p, a_q)$
●$(\langle a \rangle, n) = a$	●$(\langle a \rangle, n) = a$

Figure 7.3: Computing sum, mean, minimum and maximum incrementally, using one accumulator a; see function prototypes in Figure 7.2.

prototyped in Figure 7.2 and a vector of values to which the metric descriptor should be applied. The algorithm separates the computation into four phases. Assume a parallel job executes with P processes.

First, line 1 divides the input metric values into P groups, one group for each process, to facilitate data parallelism. Second, the algorithm initializes a process-local set of accumulators using the metric's initialize functions (line 3) and then uses the metric's accumulate functions to form partially accumulated values for that process-local set of metric values (line 5). Although this algorithm operates over one vector $\mathbf{X}$ of inputs, it can easily be extended to operate over a set of vectors, such as a CCT that has one input vector per node. In this latter case, $\mathbf{X}$ would become an input matrix where row i is a vector of inputs for CCT node i. Third, the algorithm uses a parallel reduction to reduce all the local accumulators into one set (line 8). This phase relies on the metric's combine functions. Finally, line 11 applies the metric's finalize function to obtain the metric's final value.

Figures 7.3 and 7.4 show how to compute sum, mean, minimum, maximum and standard deviation incrementally. Each algorithm gives the number of accumulators

Standard Deviation		
$\bigcirc_i()$	$= 0$	
$\bigodot_1(a_1, x)$	$= a_1 + x^2$	a_1 : sum of squares
$\bigodot_2(a_2, x)$	$= a_2 + x$	a_2 : sum
$\bigoplus_i(a_p, a_q)$	$= a_p + a_q$	
$\bullet(\langle a_1, a_2 \rangle, n)$	$= \sqrt{(a_1/n) - (a_2/n)^2}$	

Figure 7.4: Computing standard deviation incrementally, using two accumulators a_1 and a_2; see function prototypes in Figure 7.2.

needed and the definitions for the corresponding initialize $\bigcirc$, accumulate $\bigodot$, combine $\bigoplus$ and finalize $\bullet$ functions.

While the algorithms for sum, mean, minimum, and maximum are straightforward (Figure 7.3), the algorithm for standard deviation (Figure 7.4) merits further explanation. The typical definition of the standard deviation for values $x_1, \ldots, x_n$ is $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$. This formula depends on previously computing the mean μ of the x_i. Such a dependency is undesirable because it implies that the computation for standard deviation requires at least two stages. With algebraic manipulation, we can turn this formula into one that does not depend on a previously computed value. During the derivation, we use the fact that because $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$, we have $\sum_{i=1}^{n} x_i = n\mu$. Beginning with the formula for standard deviation, we have:

$$\begin{aligned}
\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2x_i\mu + \mu^2)} \\
&= \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2 - 2\mu\sum_{i=1}^{n} x_i + n\mu^2\right)} \\
&= \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2 - 2n\mu^2 + n\mu^2\right)} \qquad (\text{subst. } \sum_{i=1}^{n} x_i = n\mu) \\
&= \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2 - n\mu^2\right)}
\end{aligned}$$

$$= \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right) - \mu^2}$$

With this rearranged formula, it is evident that standard deviation can be computed using two accumulators, as shown in Figure 7.4. The first accumulator tracks the sum of x_i^2. The second, behaving like the accumulator for arithmetic mean in Figure 7.3, tracks the sum of x_i on behalf of μ. The finalize function uses both accumulators to compute the final value for standard deviation.

The example of standard deviation illustrates the more general form of incrementally computed metrics. To compute standard deviation incrementally, we effectively transform the 'non-incremental' formula into a function F that has the form:

$$F(A_1(x_1, \ldots x_n), \ldots, A_{n_A}(x_1, \ldots x_n), n) \tag{7.1}$$

where F corresponds to a finalize function, each A_j is analogous to an accumulate function (cf. Figure 7.2), and the $x_1, \ldots x_n$ are the per-thread input values. Although the value of n changes for each set of input values, the number of accumulators n_A is fixed for a given F. Each A_j has the form:

$$A_j(x_1, \ldots x_n) = \square g(x_i) \tag{7.2}$$

where $\square$ is a commutative and associative operator over all $g(x_i)$. Often, A_j is a polynomial where $\square = \sum$ and each $g(x_i)$ forms a term in the polynomial; frequent examples are $g(x_i) = x_i$ or $g(x_i) = x_i^2$. However, A_j is not necessarily polynomial; for example, $\square$ can be min or max.

After transforming a 'non-incremental' formula to have the forms of Equations 7.1 and 7.2, it is straightforward to associate an accumulator with each A_j and to compute

F in an incremental fashion. To see how A_j is analogous with an accumulate function, consider the case where A_j receives only one input value x_i at a time. A_j takes this input value x_i, applies the function g to that input, and then uses $\square$ to fold the result into the accumulator. The requirement that $\square$ be commutative and associative is due to the fact that inputs arrive in no particular order and also permits the accumulation step to be parallelized. Observe that because it is not generally possible to save all inputs ($n_A \ll n$), it is not in general possible to compute order statistics using the incremental method.

7.2.4 Scalably Presenting Call Path Profiles

To enable insightful presentation of performance data, we wish to present contextual measurements in multiple views: the Calling Context, Callers and Flat views. A Calling Context view attributes performance metrics to their full calling context. If the Calling Context view looks down a call chain, the Callers view looks up a call chain to apportion metrics of a callee on behalf of its calling contexts. A Flat view organizes performance data according to an application's static structure so that all costs incurred in any calling context by a procedure are aggregated together. Each view is important in analysis. However, the Calling Context view is foundational in the sense that it can be used to define both the Callers and Flat views, but not vice versa.

In this section, we develop formal definitions for metric values in `hpcviewer`'s Calling Context, Callers and Flat views. We designed these definitions for two purposes. First, with appropriate definitions, it is possible to compute the Callers and Flat views not just from thread-level CCT metrics but also from *derived* CCT metrics. The significance of this is that to create all three views with derived metrics, `hpcprof-mpi` only needs to generate one view — a Calling Context view with derived metrics. Second, our definitions correct deficiencies of the informal definitions used in prior versions of HPCToolkit. For example, our definitions correctly account for recursion when computing metrics for the Callers and Flat view.

Overview of the Calling Context view

The Calling Context view is represented by the canonical calling context tree (CCT) that `hpcprof-mpi` generates. An important feature of the canonical CCT is that it is a fusion of dynamic calling contexts and static program structure. Each node in the CCT can be classified as representing either a dynamic or static scope. A *dynamic* node is either a call site (CallSite) or a statement (Stmt), where a statement is a sample point. A *static* node is either a procedure frame (ProcFrame), loop (Loop), or alien code (Alien), where the latter usually represents inlined procedures. There are three important invariants that govern the structure of this CCT. First, every CallSite node has one or more ProcFrame nodes as children. Conversely, except for the root, every ProcFrame has a CallSite node for a parent. The second invariant is that every CallSite, Stmt, Loop, and Alien node is a descendant of a ProcFrame node. Third, a Stmt node is always a leaf.

We define two types of Calling Context metrics: inclusive and exclusive. Inclusive metrics for a particular node reflect costs for the entire subtree rooted at that node. This suggests that exclusive metrics for a node x do *not* include costs for the entire subtree. While this is true, it does not precisely distinguish between two reasonable definitions. The two definitions are distinguished by whether exclusive metric values for a node x should be computed with respect to dynamic call chains or static hierarchy:

1. Dynamic: sum every Stmt *descendant* of x that is not across a CallSite.

Procedure structure	Exclusive work			Inclusive work
	Dynamic	Static	Hybrid	
ProcFrame	11	0	11	111
Loop$_x$	11	0	0	111
Loop$_y$	11	1	1	111
Stmt	1	1	1	1
Loop$_z$	10	10	10	110
Stmt	5	5	5	5
Stmt	5	5	5	5
CallSite	0	0	0	100

Figure 7.5: Comparing different definitions for exclusive Calling Context metrics.

2. Static: sum every Stmt *child* of x.

These definitions are illustrated in Figure 7.5, which shows exclusive and inclusive metric values for a procedure frame with unremarkable structure. The inclusive metric values are intuitive. For example, the metric value at Loop$_z$ is 110. By including the cost of all Loop$_z$'s children (5 + 5 + 100), the value of 110 reflects the fact that the loop contains a relatively costly call site (100). The exclusive metric values require more discussion.

As the figure shows, the Exclusive/Dynamic definition is quite natural when applied to a procedure frame. The frame's three Stmt nodes are responsible for 11 units of work (1 + 5 + 5). The metric value at the ProcFrame (11) captures the fact that computation within the frame itself, excluding callees, is responsible for 11 units of work. In contrast, the Exclusive/Static definition unhelpfully reports that the ProcFrame is directly responsible for 0 units of work. Unfortunately, although the Exclusive/Dynamic definition is preferable for the ProcFrame, when applied to the loop nest rooted at Loop$_x$, this definition is less than satisfactory. To see this, observe that the Exclusive/Static column shows that there is no direct work in Loop$_x$, only 1 unit of direct work in Loop$_y$, and 10 units of direct work in Loop$_z$. Yet, unhelpfully, the Exclusive/Dynamic metric value for Loop$_x$ (11) is equal to that of Loop$_y$ (11), and nearly equal to that of Loop_z (10). To preserve the strengths of each of these definitions, we adopt a hybrid definition, shown in the Exclusive/Hybrid column, that applies the Dynamic definition to ProcFrame nodes and the Static definition to all other structure in a procedure frame. This hybrid definition makes sense when we consider that although we often think of procedure frames in the context of call chains, it is natural to think of loops in the context of a procedure.

Per-thread Calling Context view metrics

Now we are ready to precisely define exclusive and inclusive metrics for the Calling Context view. Initially, a thread-level CCT contains metric values only at Stmt nodes, or sample points (leaves). We define these values to be exclusive metrics for Stmt nodes. Specifically, for a Stmt node x at thread t, the exclusive value $m_E(x,t)$ for metric m is defined to be the number of samples at x multiplied by the sample period. For any non-Stmt node x, we initialize $m_E(x,t) = 0$. We then compute exclusive values for each node x using the formula:

$$m_E(x,t) = \begin{cases} \sum\limits_{s \in \mathsf{desc\text{-}Stmt}(x)} m_E(x_s,t) & x\text{: }\mathsf{ProcFrame} \\ \sum\limits_{s \in \mathsf{child\text{-}Stmt}(x)} m_E(x_s,t) & x\text{: other static} \\ m_E(x,t) & x\text{: dynamic} \end{cases} \tag{7.3}$$

The three cases in the formula preserve the Exclusive/Hybrid definition discussed above. When x is a dynamic node, case three simply returns the metric's initial value. When x is a static node that is not a ProcFrame, the second case uses the Static definition. Here, the function child-Stmt(x) returns every Stmt that is a child of x. When x is a ProcFrame, the first case applies the Dynamic definition. The function desc-Stmt(x) returns every Stmt s that is a descendant of x and for which the path between x and s contains no CallSite.

Using the per-thread exclusive metric values from Equation 7.3, we define per-thread inclusive values for metric m at node x and thread t as:

$$m_I(x,t) = \begin{cases} \sum_{c=1}^{n_C(x)} m_I(x_c,t) + m_E(x,t) & x\text{: interior} \\ m_E(x,t) & x\text{: leaf} \end{cases} \tag{7.4}$$

This simple inductive definition computes an interior node's inclusive metric value from its children's inclusive values and its own exclusive value. The function $n_C(x)$ refers to the number of children for node x.

Before turning to *derived* Calling Context view metrics, we note that that the definition of exclusive metrics above treats Alien nodes just like Loop nodes. Because Alien nodes usually represent inlined procedures, an alternative definition might be:

1. ProcFrame and Alien nodes: sum every Stmt *descendant* of x that is not across a CallSite or Alien node.

2. Other structural hierarchy: sum every Stmt *child* of x.

This is a defensible definition, but for reasons that will become apparent below, it should only be adopted only if Alien nodes are instantiated within the Flat view. Currently, `hpcstruct` does not recover Alien scopes with enough precision to do this well [139].

Derived Calling Context view metrics

When all thread-level inputs are available in memory simultaneously, it is possible to compute a derived Calling Context metric m for a node x by applying m's formula across all inputs. We use $m(x)$ to represent metric m's value for node x over all n_T threads. Given m's formula f, to compute derived exclusive ('E') and inclusive ('I') values for metric m at node x, we simply evaluate the following formulas, which are based on Equations 7.3 and 7.4, respectively:

$$m_E(x) = f_{t=1}^{n_T} m_E(x,t) \tag{7.5}$$

$$m_I(x) = f_{t=1}^{n_T} m_I(x,t) \tag{7.6}$$

Although computing derived metrics in this way is easy, it is not scalable.

To scalably compute Equations 7.5 and 7.6 we use the method of Algorithm 7.1. This algorithm incrementally computes metrics in parallel. To apply the algorithm to a CCT, we extend it to take not just a vector of inputs, but a vector for each CCT node. By using a dense CCT node numbering, a CCT's metric values can be represented as a dense matrix, and thus can be easily partitioned for parallel computation. Since this parallelism is straightforward, to simplify the remaining discussion, we will only focus on computing derived metrics incrementally.

Recall that Algorithm 7.1 divides the computation of a derived metric into four stages, where the two critical stages are accumulation and finalization. Let m be a derived metric m with one accumulator m^*. To compute the value of m for CCT node x, the algorithm accumulates each new thread-level input $m(x,t)$ into the accumulator using the accumulate function $\odot$. Then it applies the finalize function $\bullet$ to return m's final value $m(x)$. Recall that before application of the finalize function, we say the accumulator $m^*(x)$ is non-finalized; afterwards, it is finalized.

Thus, to incrementally compute exclusive and inclusive derived Calling Context metric values for m at a CCT node x, we must define corresponding accumulate and finalize functions. In doing this, we will depart from the notation in Figure 7.2. Although the prototypes in Figure 7.2 show precisely how the accumulate and finalize functions interact with Algorithm 7.1, they tend to emphasize the *means* of performing the computation rather than it *results*. To focus on the results of the accumulate and finalize functions, we use definitions that resemble Equations 7.5 and 7.6 rather than the prototypes of the figure. Our formulas are as follows. First, we apply the accumulate function $\bigodot$ to all thread-level inputs for a node x (see Equations 7.3 and 7.4) to obtain a non-finalized accumulator for that node:

$$m_E^*(x) = \bigodot_{t=1}^{n_T} m_E(x,t) \tag{7.7}$$

$$m_I^*(x) = \bigodot_{t=1}^{n_T} m_I(x,t) \tag{7.8}$$

Recall that the subscripts 'E' and 'I' signify 'exclusive' and 'inclusive,' respectively. Then, to obtain the final values for each node, we apply the finalize function $\bullet$:

$$m_E(x) = \bullet m_E^*(x) \tag{7.9}$$

$$m_I(x) = \bullet m_I^*(x) \tag{7.10}$$

These formulas are trivially extended to apply to metrics that use more than one accumulator.

The problem of accumulating or combining from children to a parent

For thread-level metrics, it is possible to compute inclusive Calling Context metrics from exclusive Calling Context metrics and vice versa. For example, Equation 7.4 was defined in terms of Equation 7.3 because for every Stmt x and thread t, $m_E(x,t) = m_I(x,t)$. Combining this equality with Equations 7.5 and 7.6 implies that for every Stmt x, it is also the case that $m_E(x) = m_I(x)$. Does this mean that for derived

CCT	Inclusive standard deviation of ProcFrame x based on: Per-thread values at node (correct)	Derived values from children
ProcFrame x	$\sigma_x = \sqrt{\frac{1}{n_T} \sum_1^{n_T} ((m(a,t) + m(b,t) + m(c,t)) - \mu_x)}$	$\sigma_x \neq \sigma_a + \sigma_b + \sigma_c$
Stmt a	$\sigma_a = \sqrt{\frac{1}{n_T} \sum_1^{n_T} (m(a,t) - \mu_a)}$	[e.g.: $\sigma_a = 1.0$]
Stmt b	$\sigma_b = \sqrt{\frac{1}{n_T} \sum_1^{n_T} (m(b,t) - \mu_b)}$	[e.g.: $\sigma_b = 2.0$]
Stmt c	$\sigma_c = \sqrt{\frac{1}{n_T} \sum_1^{n_T} (m(c,t) - \mu_c)}$	[e.g.: $\sigma_c = 1.0$]

Figure 7.6: Example showing that it is, in general, impossible to compute derived metric values given finalized derived metric values.

metrics it is also possible to compute inclusive Calling Context metric values from their exclusive counterparts? If so, `hpcprof-mpi` could generate its canonical CCT with only one version of each metric's values, rather than two, resulting in a smaller database.

Given *finalized* derived metrics, it is impossible in general to compute inclusive metrics from exclusive metrics. To see this, consider computing the inclusive value for a metric like standard deviation for an interior node x, where x's children are all leaves, as in Figure 7.6. The column in the figure labelled 'Per-thread values at node' shows values for standard deviation (correctly) computed according to Equations 7.5 and 7.6. In contrast, the column labelled 'Derived values from children' attempts to compute the value of x from its children. Since inclusive and exclusive metric values are identical at leaves, this column first computes values for x's children and then sums those values to form the metric value σ_x at node x (cf. Equation 7.4). Unfortunately, simply computing σ_x from the individual standard deviation values of its children is invalid and does not yield the correct answer in general. For example, if σ_a, σ_b, σ_c have the values 1.0, 2.0, and 1.0, respectively, we cannot conclude that $\sigma_x = 1.0 + 2.0 + 1.0$.

Clearly, part of the problem in this example was using finalized metric values that were the result of non-commutative and non-associative operators such as square root. However, what if we are given *non-finalized* derived exclusive metrics? Is it then possible to compute non-finalized derived inclusive metrics? In other words, is it possible to define Equation 7.8 in terms of Equation 7.7 rather than Equation 7.4? If possible, it would mean that `hpcprof-mpi` need only generate non-finalized exclusive metrics rather than both inclusive and exclusive versions.

Unfortunately, it is impossible in general to compute non-finalized derived inclusive metrics from their exclusive counterparts and vice versa. To see this, we attempt to define Equation 7.8 in terms of Equation 7.7:

$$m_I^*(x) = \begin{cases} \displaystyle\sum_{c=1}^{n_C(x)} m_I^*(x_c) + m_E^*(x) & x\text{: interior} \\ m_E^*(x) & x\text{: leaf} \end{cases} \tag{7.11}$$

(Recall that the function $n_C(x)$ refers to the number of children for node x.) We now show that Equations 7.8 and 7.11 are not equivalent in general:

$$\begin{aligned} m_I^*(x) &= \bigodot_{t=1}^{n_T} m_I(x,t) && \text{(Eq. 7.8)} \\ &= \begin{cases} \displaystyle\bigodot_{t=1}^{n_T} \left(\sum_{c=1}^{n_C(x)} m_I(x_c,t) + m_E(x,t) \right) & x\text{: interior} \\ \displaystyle\bigodot_{t=1}^{n_T} m_E(x,t) & x\text{: leaf} \end{cases} && \text{(subst. Eq. 7.4)} \\ &= \begin{cases} \displaystyle\bigodot_{t=1}^{n_T} \sum_{c=1}^{n_C(x)} m_I(x_c,t) + m_E^*(x) & x\text{: interior} \\ m_E^*(x) & x\text{: leaf} \end{cases} && \text{(subst. Eq. 7.7)} \end{aligned}$$

$$= \begin{cases} \sum_{c=1}^{n_C(x)} \bigodot_{t=1}^{n_T} m_I(x_c, t) + m_E^*(x) & x\text{: interior} \\ m_E^*(x) & x\text{: leaf} \end{cases} \qquad (\text{iff } \bigodot = \sum\text{!!!})$$

$$= \begin{cases} \sum_{c=1}^{n_C(x)} m_I^*(x_c) + m_E^*(x) & x\text{: interior} \\ m_E^*(x) & x\text{: leaf} \end{cases} \qquad (\text{subst. Eq. 7.8})$$

This derivation shows that Equation 7.8 is equivalent to Equation 7.11 if and only if the accumulate function $\bigodot$ is $\sum$. As Figure 7.3 shows, this condition does not hold for the metrics minimum and maximum.

One might wonder if the result would change with a different definition of Equation 7.11. The answer is no. To see this, observe that *any* attempt to compute non-finalized inclusive metrics for an interior node from their exclusive counterparts must include some form of accumulation (with $\bigodot$) or combination (with $\bigoplus$) from that node's children. This is because inclusive and exclusive metric values are only identical at a Stmt node (leaf). Thus, any alternative definition that includes accumulation or combination from children to parents will meet the same problem.

The fact that it is in general invalid to accumulate or combine non-finalized derived metric values from children to parents will restrict the forms that the algorithms for computing the Callers and Flat views can take. This is particularly true for computing the Flat view, which is discussed next.

Flat view metrics

`hpcviewer`'s Flat view organizes performance data according to an application's static structure. This means that all costs incurred by a procedure in any calling context are aggregated together. As with the Calling Context view, it is trivial to

Algorithm 7.2: make-flat-view: Given a Calling Context view with non-finalized derived metric values, make a Flat view.

```
Input: Metric descriptor M, which includes the following (see Figures 7.2 and 7.7):
       combine functions $\bigoplus_1, \ldots, \bigoplus_{n_A}$ and a finalize function ●.
Input: cct, a Calling Context view with non-finalized exclusive ('E') and inclusive
       ('I') metric values for M.
Result: A Flat view flat with finalized exclusive and inclusive metric values for M.
1  let flat be an empty Flat view
2  foreach Stmt or CallSite x in cct do
3      let $\pi_{\text{cct}} = \langle\langle \text{caller}_n, \text{callsite}_n\rangle \rightsquigarrow \langle \text{caller}_1, \text{callsite}_1\rangle \rightarrow x\rangle$ be the path from cct's
           root to $x$
4      let $\kappa_{\text{cct}} = \langle \text{ProcFrame}, (\text{Loop}|\text{Alien})^*, x\rangle$ be the static context of $x$ within cct
5      let $\kappa_{\text{flat}}$ be the corresponding context in flat
6      foreach $\langle y_{cct}, y_{flat}\rangle$ in make-pairs($\kappa_{cct}$, $\kappa_{flat}$) do
7          $m_E^*(y_{\text{flat}}) \Leftarrow \bigoplus(m_E^*(y_{\text{flat}}), m_E^*(y_{\text{cct}}))$          // for each $\bigoplus_i$
8          if is-outermost-instance($\pi_{cct}$, $\kappa_{cct}$'s ProcFrame) or $y_{cct}$ is a Stmt then
9              $m_I^*(y_{\text{flat}}) \Leftarrow \bigoplus(m_I^*(y_{\text{flat}}), m_I^*(y_{\text{cct}}))$      // for each $\bigoplus_i$
10 Apply finalize function ● to each node in flat.
```

compute derived metrics for the Flat view by first creating all thread-level (Flat view) metrics (cf. Equations 7.5 and 7.6). However, creating all thread-level metrics for petascale executions is both time- and space-consuming. This section shows how to compute a Flat view with derived metrics using a Calling Context view with non-finalized derived metrics.

Algorithm 7.2 shows the process for building a Flat view. Assume we have a metric descriptor that defines an incrementally computed metric (see Figure 7.2). Given this metric descriptor and a Calling Context view with exclusive and inclusive metric values, the algorithm shows how to compute a Flat view with both exclusive and inclusive metric values.

This algorithm does two things: build the structure of the Flat view and attribute metrics to it. Lines 4–5 build the structure. Line 4 obtains the static structure of the Stmt or CallSite x within the Calling Context view. This static structure includes all of the enclosing scopes between x and the immediately enclosing ProcFrame. Line 5 creates or finds this corresponding structure in the Flat view. In this way, the Flat view aggregates metrics according to the static structure of ProcFrame nodes, regardless of its calling context.

The second part of the algorithm, which is located in the loop beginning at line 6, computes metric values. The loop considers each portion of x's static context κ_{cct} and attributes metric values from the instance in the Calling Context view to the corresponding instance in the Flat view. (The function make-pairs takes κ_{cct} and its counterpart in the Flat view and makes pairs of corresponding nodes.) Recall that the metric values in the Calling Context view are non-finalized. Since attributing metric values from the Calling Context to the Flat view requires aggregating multiple non-finalized values, lines 7 and 9 use the metric's combine operator $\bigoplus$. To account for metric descriptors that have multiple combine operators, the left-margin comments on these lines indicate that the respective statements should be considered vector operations. There are two important subtleties related to combining metric values. We discuss each in turn.

The first subtlety is the use of two tests on line 8 to conditionally attribute inclusive metrics. These tests are designed to correctly attribute metrics in the presence of recursion. There are two cases to consider, depending on the result of is-outermost-instance. For the first case, let us refer to the context κ_{cct}'s ProcFrame as $\mathcal{F}$. We know that the path π_{cct} from the Calling Context view's root to x contains at least one instance of $\mathcal{F}$; for recursive programs there may be more than one instance. Let $\mathcal{F}'$ be the first instance of $\mathcal{F}$ encountered along this path. The function is-outermost-instance returns true if and only if $\mathcal{F} = \mathcal{F}'$. That is, it returns true if and only if $\mathcal{F}$'s inclusive metric values have not been folded into an ancestor instance. If $\mathcal{F}$ is an outermost instance, then every portion of the context (including every Stmt) is combined into the corresponding context in the Flat view by line 9. If, on the other hand, $\mathcal{F}$ is not an outermost instance, then only its Stmt nodes (leaves) can contain metric values that are not already reflected in the inclusive costs of the Flat view's structure κ_{flat}. Consequently, the test ensures any such Stmt in the Calling Context view is added to its corresponding Stmt in the Flat view.

The second subtlety is that using a metric's combine function affects the value of the number of inputs n that must be passed to that metric's finalize function in the Flat view. For example, assume the Calling Context view contains two instances of a ProcFrame p. In the Calling Context view, summary metric values for each instance of p are computed over all n_T per-thread input values. In contrast, the Flat view's metrics reflect values over *all* instances of p. This means that the Flat view's metric values for p have $2 \times n_T$ input values. In general, if a node x appears k times in the Calling Context view, its combine operator will be applied $k - 1$ times to create non-finalized values for x in the Flat view; and the number of inputs for x's Flat view metrics will be $k \times n_T$. Thus, when making the Flat view, it is necessary to track the appropriate number of inputs for each node that should be passed to the finalize function. As shown in Figure 7.7, this is easy to do by extending the definitions of metrics that depend on the number of inputs to use an additional accumulator. This additional accumulator tracks the number of Calling Context view instances that the algorithm combines to form a given node in the Flat view.

It is worth observing that there is an alternative way to compute Flat view metrics for non-derived (thread-level) metric values. This alternative method first accumulates values at Stmt and CallSite nodes within the Flat view and then, within the Flat view, uses combine functions to aggregate those (leaf) values to the interior static structure of ProcFrame nodes. Unfortunately, this method is incorrect for derived metrics. It is an instance of the previously discussed problem of combining non-finalized derived metric values from children to parents.

Function Prototypes		
initialize	$\bigcirc_i()$	$: \qquad \mapsto$ accumulator$_i$
combine	$\bigoplus_i(a, a_{\text{cct}})$	$:$ accumulator$_i$ $\times$ CCT-accumulator$_i$ $\mapsto$ accumulator$_i$
finalize	$\bullet(\langle a_1, \ldots, a_{n_A} \rangle, n)$	$:$ accumulator-list $\times$ CCT-input-size $\mapsto$ output

Mean	
$\bigcirc_1() = 0$	
$\bigcirc_2() = 1$	
$\bigoplus_1(a, a_{\text{cct}}) = a + a_{\text{cct}}$	a_1 : sum
$\bigoplus_2(a, a_{\text{cct}}) = a + 1$	a_2 : scope's instances within CCT
$\bullet(\langle a_1, a_2 \rangle, n) = a_1 / a_2 n$	

Standard Deviation	
$\bigcirc_{1,2}() = 0$	
$\bigcirc_3() = 1$	
$\bigoplus_1(a, a_{\text{cct}}) = a + a_{\text{cct}}$	a_1 : sum of squares
$\bigoplus_2(a, a_{\text{cct}}) = a + a_{\text{cct}}$	a_2 : sum
$\bigoplus_3(a, a_{\text{cct}}) = a + 1$	a_3 : scope's instances within CCT
$\bullet(\langle a_1, a_2, a_3 \rangle, n) = \sqrt{(a_1/a_3 n) - (a_2/a_3 n)^2}$	

Figure 7.7: Computing metrics incrementally for a Flat or Callers view; cf. Figures 7.2, 7.3, and 7.4. Each metric uses accumulators $a_1, \ldots, a_{n_A}$. The finalize function assumes n is the number of inputs for the corresponding Calling Context view.

Finally, it is sometimes useful to include load module and file information in the Flat view structure. To compute metric values for these scopes while avoiding the problem of combining non-finalized metric values from children to parents, it would be necessary to include metric values for load modules and files in the Calling Context view or use `hpcprof-mpi` to precompute metrics for those two layers of the Flat view.

Callers view metrics

If the Calling Context view looks down a call chain, the Callers view looks up a call chain to apportion metrics of a callee (in its context) on behalf of its caller. As with the Calling Context and Flat views, it is trivial to compute derived metrics for the Callers view by first creating all thread-level (Callers view) metrics (cf. Equations 7.5 and 7.6). However, if this was undesirable for the Flat view, it is even more undesirable

Algorithm 7.3: make-callers-view: Given a Calling Context view with non-finalized derived metric values, make a Callers view.

Input: Metric descriptor **M**, which includes the following (see Figures 7.2 and 7.7): combine functions $\bigoplus_1, \ldots, \bigoplus_{n_A}$ and a finalize function ●.
Input: *cct*, a Calling Context view with non-finalized exclusive ('E') and inclusive ('I') derived metric values for **M**.
Result: A Callers view with finalized exclusive and inclusive metric values for **M**.

```
1 let callers be an empty Callers view
2 foreach ProcFrame $x$ in cct do
3     let $\pi_{\text{cct}} = \langle\langle \text{caller}_n, \text{callsite}_n\rangle \rightsquigarrow \langle \text{caller}_1, \text{callsite}_1\rangle \rightarrow x\rangle$ be the path from cct's
          root to $x$. ($x \rightsquigarrow y \equiv x \rightarrow \ldots \rightarrow y$, where it may be that $x = y$.)
4     let $\pi_{\text{callers}}$ be the corresponding path in callers. $\pi_{\text{callers}}$ is formed by reversing
          $\pi_{\text{cct}}$ and projecting out all non-ProcFrame nodes.
5     foreach $\langle y_{cct}, y_{callers}\rangle$ in make-pairs($\pi_{cct}$, $\pi_{callers}$) do
          // Given path $y_{\text{cct}} \rightarrow x' \rightsquigarrow x$, attribute $x' \rightsquigarrow x$ to $y_{\text{cct}}$
6         $m^*_E(y_{\text{callers}}) \Leftarrow \bigoplus (m^*_E(y_{\text{callers}}), m^*_E(x))$            // for each $\bigoplus_i$
7         if is-outermost-instance($\pi_{cct}$, $y_{cct} \rightarrow x' \rightsquigarrow x$) then
8             $m^*_I(y_{\text{callers}}) \Leftarrow \bigoplus (m^*_I(y_{\text{callers}}), m^*_I(x))$        // for each $\bigoplus_i$
9 Apply finalize function ● to each node in callers.
```

for the Callers view. Creating all Callers-view thread-level metrics is especially time- and space-consuming because the Callers view is quadratic in terms of the Calling Context view. That is, if the Calling Context view has n nodes, then the Callers-view has $O(n^2)$ nodes. To avoid such behavior, this section shows how to compute a Callers view with derived metrics using a Calling Context view with only non-finalized derived metrics.

Algorithm 7.3 describes the process for building a Callers view. Assume we have a metric descriptor that defines an incrementally computed metric (see Figure 7.2). Given this metric descriptor and a Calling Context view with exclusive and inclusive metric values, the algorithm shows how to compute a Callers view with both exclusive and inclusive metric values.

Like Algorithm 7.2, this algorithm does two things: build the structure of the Callers view and attribute metrics to it. Lines 3–4 build the structure. As a notational note, the algorithm uses an arrow ($\rightarrow$) to denote a call and a squiggly arrow ($\rightsquigarrow$) to denote a (possibly-empty) path of calls, i.e., $x \rightsquigarrow y \equiv x \rightarrow \ldots \rightarrow y$, where it may be that $x = y$. To build the view's structure, line 3 obtains the full path from the Calling Context view's root to the ProcFrame x. Then, line 4 creates or finds the corresponding structure in the Callers view. This corresponding structure is rooted at x. Observe that the Callers view is a forest, with a root for each ProcFrame. Using this structure, for every ProcFrame x in the Calling Context view, the Callers view shows all the paths (contexts) that x was called from and attributes x's metrics, in context, to its various callers.

The second and critical part of the algorithm is to compute metric values for the Callers view. This occurs in the loop beginning at line 5. Recall that line 3 defines π_{cct}, the path from the Calling Context view's root to the ProcFrame x. The loop at line 5 effectively considers all calling contexts for x within π_{cct} — i.e., all paths in π_{cct} for which x is a sink — and attributes the metric values of x to the corresponding caller in the Callers view. To do this, the function make-pairs takes the path π_{cct} and its counterpart in the Callers view and makes pairs of corresponding nodes, where y_{cct} acts as a cursor in the path π_{cct}. From y_{cct} and x, we form the path $y_{\text{cct}} \rightarrow x' \rightsquigarrow x$, where it may be that $y_{\text{cct}} = x$. Then, the algorithm attributes the metric values of x in context ($x' \rightsquigarrow x$) to that context's caller, y_{cct} (lines 6 and 8). There are two things that require further discussion: the use of is-outermost-instance and the process of attributing metric values. We discuss them in turn.

As with Algorithm 7.2, it is necessary to use is-outermost-instance (line 7) to correctly attribute inclusive metrics in the presence of recursion. However, whereas in that algorithm it was only necessary to consider whether a single ProcFrame appeared in the path π_{cct}, in this algorithm it is necessary for is-outermost-instance to determine whether a *path* (context) appears in the path π_{cct}. This is because whereas the Flat view attributes the cost of a particular procedure regardless of calling context, the Callers view attributes the cost of a particular procedure *in its calling context* ($x' \rightsquigarrow x$) to the context's caller (y_{cct}). Consequently, this version of is-outermost-instance returns true if and only if there is no instance of $y_{\text{cct}} \rightarrow x' \rightsquigarrow x$ that is a (strict) ancestor of y_{cct} in the path π_{cct}. If so, inclusive metric values are updated.

The process of attributing the metric values of x to y_{cct} (lines 6 and 8) can be divided into two cases. The first case is the special case of $y_{\text{cct}} = x' = x$. In this case, x is a root in the Callers view and its metrics are computed without context, as in the Flat view. In the second case, $y_{\text{cct}} \neq x$. Here, it is necessary to attribute metrics for x's calling context to that context's caller, y_{cct}. The calling context for x is $x' \rightsquigarrow x$, where x' may or may not equal x. The metric values that must be attributed to y_{cct} are precisely those at x, since these values are with respect to the context $x' \rightsquigarrow x$ as it is called by y_{cct}. Because the outer loop (line 2) considers all additional instances of ProcFrame x with the Calling Context view, the algorithm correctly apportions the metrics of the procedure x, in its various calling contexts, on behalf of that context's caller.

The second point of further discussion relates to metric values. Recall that the metric values in the Calling Context view are non-finalized. Since attributing metric values from the Calling Context to the Callers view requires aggregating multiple non-finalized values, lines 6 and 8 of the algorithm use the metric's combine operator $\bigoplus$. To account for metric descriptors that have multiple combine operators, the left-margin comments on these lines indicate that the respective statements should be considered vector operations. As with the Flat view, using a metric's combine function affects the value of the number of inputs n that must be passed to that metric's finalize function. This means, e.g., that if the Calling Context view contains two instances of a ProcFrame p and has n_T input metric values, the Caller's view root for p has $2 \times n_T$ input values. Thus, it is necessary to use the revised metric definitions of Figure 7.7 to track the number of Calling Context view instances that this algorithm combines into a given node in the Callers view.

The Callers view potentially contains a full copy of the canonical CCT for every ProcFrame in the Calling Context view. That is, if the CCT has n nodes, then *each* root x in the Callers view potentially has $O(n)$ nodes, because ProcFrame x could appear in every calling context within the CCT. Because of this, the Callers view has the undesirable property that it requires space that is quadratic in terms of the input CCT. To prevent this, it is possible to trade space for time and build the view incrementally on demand. Two things are necessary to do this. First, we modify Algorithm 7.3 to initially compute only the top (first) level of the Callers view by setting π_{cct} to x on line 3. When a user attempts to expand any particular subtree, we can build only that subtree. Second, before finalizing the metric values in the Calling Context view, it is necessary to keep a copy of the non-finalized values for use in the demand-driven Callers view algorithm.

7.3 Application Studies

To demonstrate the utility of HPCToolkit for performance analysis of applications on emerging petascale applications, we apply it to study the performance of three codes: PFLOTRAN, FLASH, and MILC. We studied these applications on core counts up to 8192.[3] Our performance studies were performed on two systems: Jaguar — a Cray XT system at Oak Ridge National Laboratory's National Center for the Computational Sciences — and Intrepid — a Blue Gene/P at Argonne National

[3]We could have used larger core counts for our study, but opted to limit the scale of our executions to limit our resource consumption.

Laboratory's Leadership Computing Facility. We describe these machines as they exist in Spring 2010.

Jaguar consists of 84 Cray XT4 racks and 200 Cray XT5 racks linked together. There are 7,832 XT4 compute nodes and 18,688 XT5 compute nodes for a total of 255,584 cores. Each XT4 node contains a quad-core 2.1 GHz Opteron (Budapest), 8 GB memory and a SeaStar2 network interface card, for a total of 31,328 cores. Each XT5 node contains two hex-core 2.6 GHz Opterons (Istanbul), twice the memory and twice the memory bandwidth, but with one SeaStar2+ interface card, for a total of 224,256 cores. Nodes in the system are arranged in a 3-D torus topology. Compute nodes run Cray's Compute Node Linux (CNL) microkernel. In early February 2009, CNL version 2.1 was installed which corrects bugs that inhibited asynchronous sampling in prior versions.

Intrepid is a BlueGene/P system with 163,840 compute cores divided into 40 racks. Each rack consists of 1024 compute nodes (and is thus more densely populated than a Cray XT). Each node is a custom system-on-a-chip design that contains four 850 MHz PowerPC 450 cores, each with a dual floating point unit, and 2 GB of off-chip shared memory. Multiple networks connect each node by attaching directly to the SoC, including a 3-D torus, a global collective network (for broadcasts and reductions), and a global barrier network. Compute nodes run IBM's Compute Node Kernel for BG/P. In late January 2009, patches were installed to correct bugs in kernel version V1R3M0 that inhibited asynchronous sampling.

We collected Jaguar data on XT4 or XT5 nodes in which an MPI process was assigned to each core. Similarly, we collected BG/P data using 'virtual node' mode in which an MPI process was assigned to each core.

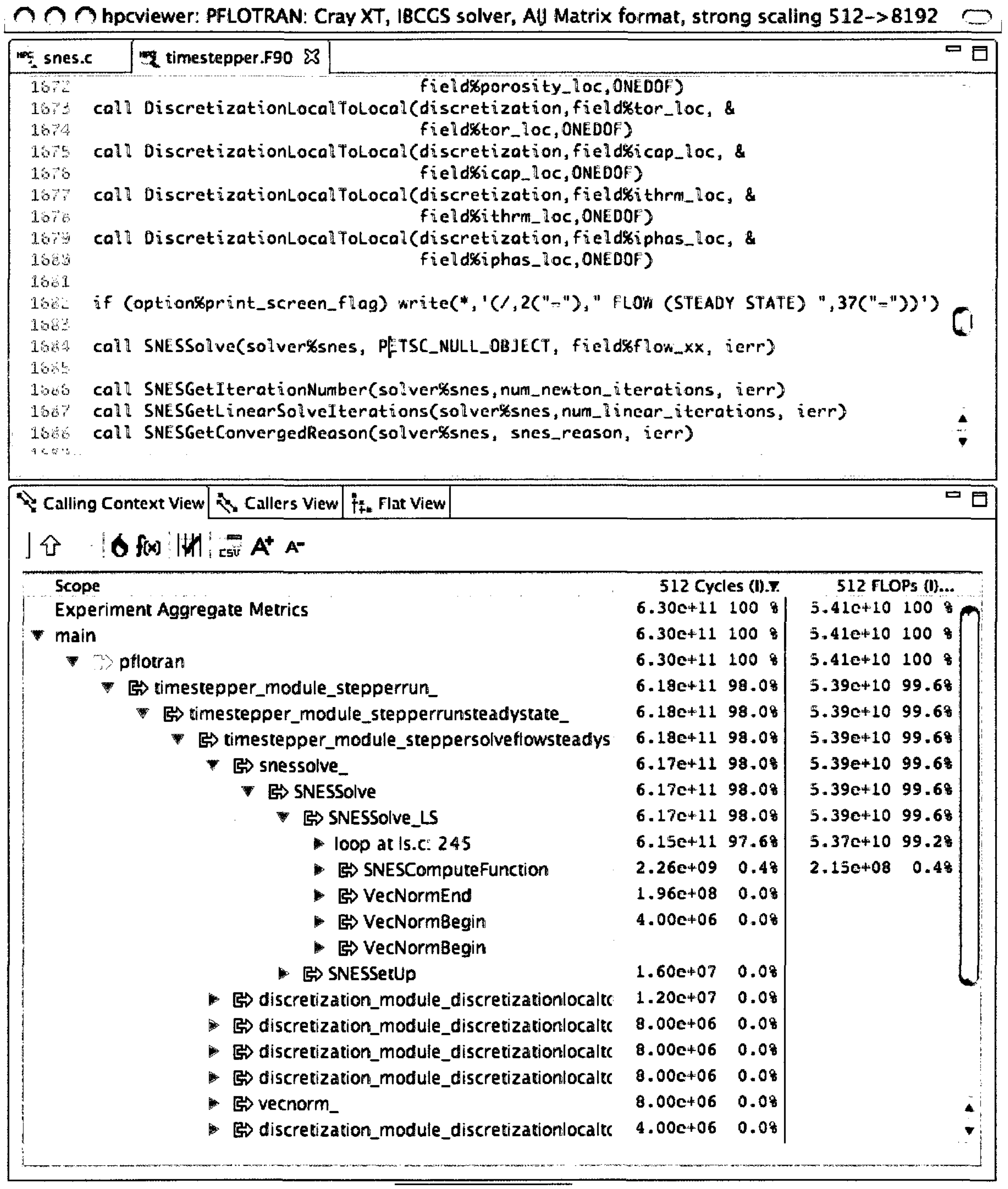
7.3.1 PFLOTRAN

PFLOTRAN is a code for modeling multi-phase, multi-component subsurface flow and reactive transport using massively parallel computers [86, 100]. The code is designed to predict the migration of contaminants underground. "PFLOTRAN solves a coupled system of mass and energy conservation equations for multiple compounds and phases including H_2O, supercritical CO_2, black oil, and a gaseous phase" [100]. With support from the DOE SciDAC program, the authors of PFLOTRAN plan to use it to understand radionuclide migration at the DOE Hanford facility and model sequestration of CO_2 in deep geologic formations. Typical simulations involve massive computation due to ten or more chemical degrees of freedom on a grid of millions of nodes. PFLOTRAN employs the PETSc library's Newton-Krylov solver framework.

Analyzing scaling losses on a Cray XT4

In this section, we use HPCToolkit to examine study the performance of PFLOTRAN when strong scaling from 512 to 8192 cores of a Cray XT4. (A strong scaling study employs different numbers of cores on the same test problem.) The test problem used for this section is a steady-state groundwater flow problem in heterogeneous porous media on a 512^3 element discretization. It uses PETSc's IBCGS (Improved Stabilized version of BiConjugate Gradient Squared) solver [114,155] to solve for flow.

Figure 7.8 shows a screen snapshot from HPCToolkit's `hpcviewer` user interface displaying a top-down Calling Context view of how PFLOTRAN spends its time on 512 processors. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, partially expanded. One can see several procedure instances along the call paths in the calling context tree. Each entry in the navigation pane is associated with metric values in the metric pane to its right. The line selected in the navigation pane



hpcviewer: PFLOTRAN: Cray XT, IBCGS solver, AIJ Matrix format, strong scaling 512->8192

snes.c | timestepper.F90

```
                                 field%porosity_loc,ONEDOF)
  call DiscretizationLocalToLocal(discretization,field%tor_loc, &
                                 field%tor_loc,ONEDOF)
  call DiscretizationLocalToLocal(discretization,field%icap_loc, &
                                 field%icap_loc,ONEDOF)
  call DiscretizationLocalToLocal(discretization,field%ithrm_loc, &
                                 field%ithrm_loc,ONEDOF)
  call DiscretizationLocalToLocal(discretization,field%iphas_loc, &
                                 field%iphas_loc,ONEDOF)

  if (option%print_screen_flag) write(*,'(/,2("="),"  FLOW (STEADY STATE) ",37("="))')

  call SNESSolve(solver%snes, PETSC_NULL_OBJECT, field%flow_xx, ierr)

  call SNESGetIterationNumber(solver%snes,num_newton_iterations, ierr)
  call SNESGetLinearSolveIterations(solver%snes,num_linear_iterations, ierr)
  call SNESGetConvergedReason(solver%snes, snes_reason, ierr)
```

Calling Context View | Callers View | Flat View

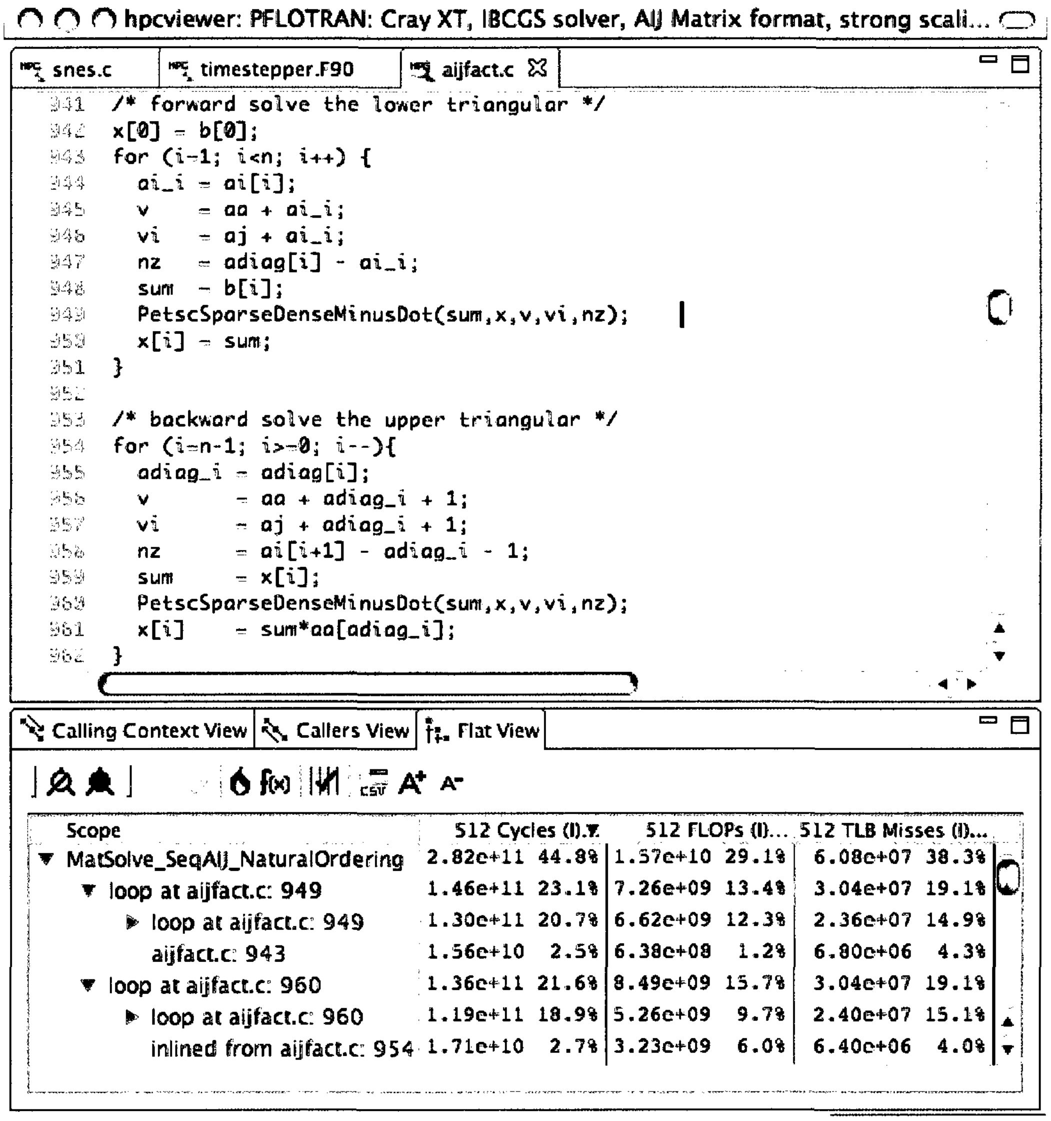
```
Scope                                                   512 Cycles (I)      512 FLOPs (I)...
Experiment Aggregate Metrics                            6.30e+11  100 %     5.41e+10  100 %
▼ main                                                  6.30e+11  100 %     5.41e+10  100 %
  ▼ pflotran                                            6.30e+11  100 %     5.41e+10  100 %
    ▼ timestepper_module_stepperrun_                    6.18e+11  98.0%     5.39e+10  99.6%
      ▼ timestepper_module_stepperrunsteadystate_       6.18e+11  98.0%     5.39e+10  99.6%
        ▼ timestepper_module_steppersolveflowsteadys    6.18e+11  98.0%     5.39e+10  99.6%
          ▼ snessolve_                                  6.17e+11  98.0%     5.39e+10  99.6%
            ▼ SNESSolve                                 6.17e+11  98.0%     5.39e+10  99.6%
              ▼ SNESSolve_LS                            6.17e+11  98.0%     5.39e+10  99.6%
                ▶ loop at ls.c: 245                     6.15e+11  97.6%     5.37e+10  99.2%
                ▶ SNESComputeFunction                   2.26e+09   0.4%     2.15e+08   0.4%
                ▶ VecNormEnd                            1.96e+08   0.0%
                ▶ VecNormBegin                          4.00e+06   0.0%
                ▶ VecNormBegin
              ▶ SNESSetUp                               1.60e+07   0.0%
          ▶ discretization_module_discretizationlocaltc 1.20e+07   0.0%
          ▶ discretization_module_discretizationlocaltc 8.00e+06   0.0%
          ▶ discretization_module_discretizationlocaltc 8.00e+06   0.0%
          ▶ discretization_module_discretizationlocaltc 8.00e+06   0.0%
          ▶ vecnorm_                                    8.00e+06   0.0%
          ▶ discretization_module_discretizationlocaltc 4.00e+06   0.0%
```

Figure 7.8: hpcviewer's Calling Context view of PFLOTRAN on a Cray XT4.

is displayed in the source pane (top sub-pane). For the steady state flow problem measured, on 512 processors the selected line shows that PFLOTRAN spends 98% of its time (measured as inclusive processor cycles using the PAPI [28] interface to hardware counters) inside PETSc's `SNESolve` procedure, called from PFLOTRAN's `StepperSolveFlowSteadyState` procedure in module `Timestepper_module`. Comparing the cycles spent in `SNESolve` with the floating point operations performed (shown in the rightmost column), we see that the solver executes only one floating point operation about every 11 cycles. This low performance bears further investigation.

Figure 7.9 shows a Flat view of the most costly procedure, PETSc's `MatSolve_SeqAIJ_NaturalOrdering`, where the 512 processor execution of PFLOTRAN spent 44.8% of the total execution time when executing the steady state flow problem. A strength of HPCToolkit is that it attributes costs not only at the routine level, but at the loop level too. The second line of the metric pane shows the most costly loop in the aforementioned routine: a forward solve of a lower triangular matrix, which accounts for 23.1% of the total cycles during execution. Almost all of the loop's costs are attributed to line 949 of file `aijfact.c` since the PGI compiler only associates one source line number with each basic block. By comparing the cycles with the second column, floating point operations, we see that the loop executes only about one floating point operation every 20 cycles. The fact that we can pinpoint and quantify the nature of this performance loss demonstrates HPCToolkit's abilities for locating node performance bottlenecks.

In the loop highlighted in Figure 7.9, L2 misses (elided in the figure) are lower than average: the loop accounts for only 8.0% of the L2 misses even though it accounts for 23.1% of the cycles. Execution time for the loop correlates more closely with TLB misses: 19.1% of TLB misses and 23.1% of the program cycles. Comparing the



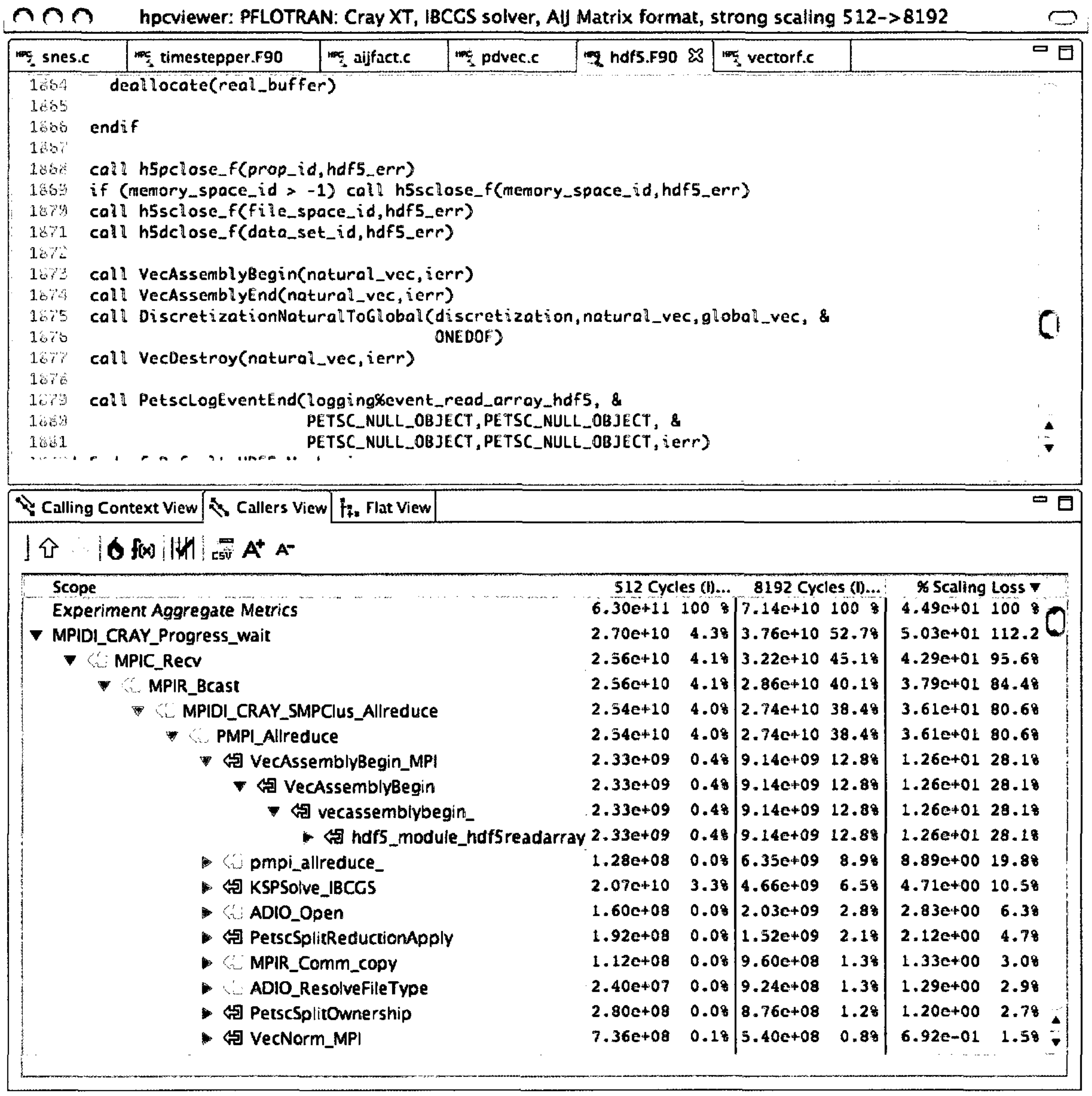
hpcviewer: PFLOTRAN: Cray XT, IBCGS solver, AIJ Matrix format, strong scali...

```
941  /* forward solve the lower triangular */
942  x[0] = b[0];
943  for (i=1; i<n; i++) {
944    ai_i = ai[i];
945    v    = aa + ai_i;
946    vi   = aj + ai_i;
947    nz   = adiag[i] - ai_i;
948    sum  = b[i];
949    PetscSparseDenseMinusDot(sum,x,v,vi,nz);
950    x[i] = sum;
951  }
952
953  /* backward solve the upper triangular */
954  for (i=n-1; i>=0; i--){
955    adiag_i = adiag[i];
956    v       = aa + adiag_i + 1;
957    vi      = aj + adiag_i + 1;
958    nz      = ai[i+1] - adiag_i - 1;
959    sum     = x[i];
960    PetscSparseDenseMinusDot(sum,x,v,vi,nz);
961    x[i]    = sum*aa[adiag_i];
962  }
```

Scope	512 Cycles (I).▼		512 FLOPs (I)...		512 TLB Misses (I)...	
▼ MatSolve_SeqAIJ_NaturalOrdering	2.82e+11	44.8%	1.57e+10	29.1%	6.08e+07	38.3%
▼ loop at aijfact.c: 949	1.46e+11	23.1%	7.26e+09	13.4%	3.04e+07	19.1%
▶ loop at aijfact.c: 949	1.30e+11	20.7%	6.62e+09	12.3%	2.36e+07	14.9%
aijfact.c: 943	1.56e+10	2.5%	6.38e+08	1.2%	6.80e+06	4.3%
▼ loop at aijfact.c: 960	1.36e+11	21.6%	8.49e+09	15.7%	3.04e+07	19.1%
▶ loop at aijfact.c: 960	1.19e+11	18.9%	5.26e+09	9.7%	2.40e+07	15.1%
inlined from aijfact.c: 954	1.71e+10	2.7%	3.23e+09	6.0%	6.40e+06	4.0%

Figure 7.9: hpcviewer's Flat view of PFLOTRAN on a Cray XT4.

number of TLB misses to the number of floating point operations shows that there is a TLB miss for every 239 floating point operations. These measurements suggest that the performance on the Opteron architecture might be improved by reducing TLB misses. To reduce the TLB miss rate, we tried using 2MB jumbo pages; however, we found that this change had little effect on overall run time. This suggests that



```
  deallocate(real_buffer)

endif

call h5pclose_f(prop_id,hdf5_err)
if (memory_space_id > -1) call h5sclose_f(memory_space_id,hdf5_err)
call h5sclose_f(file_space_id,hdf5_err)
call h5dclose_f(data_set_id,hdf5_err)

call VecAssemblyBegin(natural_vec,ierr)
call VecAssemblyEnd(natural_vec,ierr)
call DiscretizationNaturalToGlobal(discretization,natural_vec,global_vec, &
                                   ONEDOF)
call VecDestroy(natural_vec,ierr)

call PetscLogEventEnd(logging%event_read_array_hdf5, &
                      PETSC_NULL_OBJECT,PETSC_NULL_OBJECT, &
                      PETSC_NULL_OBJECT,PETSC_NULL_OBJECT,ierr)
```

Scope	512 Cycles (I)...		8192 Cycles (I)...		% Scaling Loss ▼	
Experiment Aggregate Metrics	6.30e+11	100 %	7.14e+10	100 %	4.49e+01	100 %
▼ MPIDI_CRAY_Progress_wait	2.70e+10	4.3%	3.76e+10	52.7%	5.03e+01	112.2
▼ MPIC_Recv	2.56e+10	4.1%	3.22e+10	45.1%	4.29e+01	95.6%
▼ MPIR_Bcast	2.56e+10	4.1%	2.86e+10	40.1%	3.79e+01	84.4%
▼ MPIDI_CRAY_SMPClus_Allreduce	2.54e+10	4.0%	2.74e+10	38.4%	3.61e+01	80.6%
▼ PMPI_Allreduce	2.54e+10	4.0%	2.74e+10	38.4%	3.61e+01	80.6%
▼ VecAssemblyBegin_MPI	2.33e+09	0.4%	9.14e+09	12.8%	1.26e+01	28.1%
▼ VecAssemblyBegin	2.33e+09	0.4%	9.14e+09	12.8%	1.26e+01	28.1%
▼ vecassemblybegin_	2.33e+09	0.4%	9.14e+09	12.8%	1.26e+01	28.1%
▶ hdf5_module_hdf5readarray	2.33e+09	0.4%	9.14e+09	12.8%	1.26e+01	28.1%
▶ pmpi_allreduce_	1.28e+08	0.0%	6.35e+09	8.9%	8.89e+00	19.8%
▶ KSPSolve_IBCGS	2.07e+10	3.3%	4.66e+09	6.5%	4.71e+00	10.5%
▶ ADIO_Open	1.60e+08	0.0%	2.03e+09	2.8%	2.83e+00	6.3%
▶ PetscSplitReductionApply	1.92e+08	0.0%	1.52e+09	2.1%	2.12e+00	4.7%
▶ MPIR_Comm_copy	1.12e+08	0.0%	9.60e+08	1.3%	1.33e+00	3.0%
▶ ADIO_ResolveFileType	2.40e+07	0.0%	9.24e+08	1.3%	1.29e+00	2.9%
▶ PetscSplitOwnership	2.80e+08	0.0%	8.76e+08	1.2%	1.20e+00	2.7%
▶ VecNorm_MPI	7.36e+08	0.1%	5.40e+08	0.8%	6.92e-01	1.5%

Figure 7.10: `hpcviewer`'s Callers view of scaling losses for PFLOTRAN on a Cray XT4.

we should use other hardware counters to further investigate the reason for the low performance.

Figure 7.10 shows a bottom-up Callers view of the losses when scaling from solving the test problem on 512 cores to 8192 cores (strong scaling). The Callers view apportions the cost of a procedure (in context) to each call site in each of its callers. For inclusive costs (as shown in this figure), `hpcviewer`'s bottom-up view attributes costs incurred within. For each calling context c in the program executions in this scaling study, we compute the percent of scaling losses as $100(16\ \mathrm{T}_{c,8192} - \mathrm{T}_{c,512})/(16\ \mathrm{T}_{r,8192})$, where r is the root of the calling context tree, and $T_{i,n}$ represents the time spent in context i in an n core execution. In English, the quantity $(16\ \mathrm{T}_{c,8192} - \mathrm{T}_{c,512})$ calculates the difference in parallel work performed by the executions on 512 and 8192 cores for a particular calling context c. The factor of 16 arises because when strong scaling from 512 to 8192 processors, the amount of work per processor is a factor of 16 smaller on the larger number of processors. We divide through by $16\ \mathrm{T}_{r,8192}$, the total amount of work performed on 8192 cores, to compute the relative fraction of the execution that corresponds to parallel overhead. We multiply through by 100 to express this number in percent. In Figure 7.10, the percent relative scaling loss in the 8192-core execution is represented using scientific notation. The percentages shown in that column show the percentage of the total scaling loss that is associated with each line in the display.

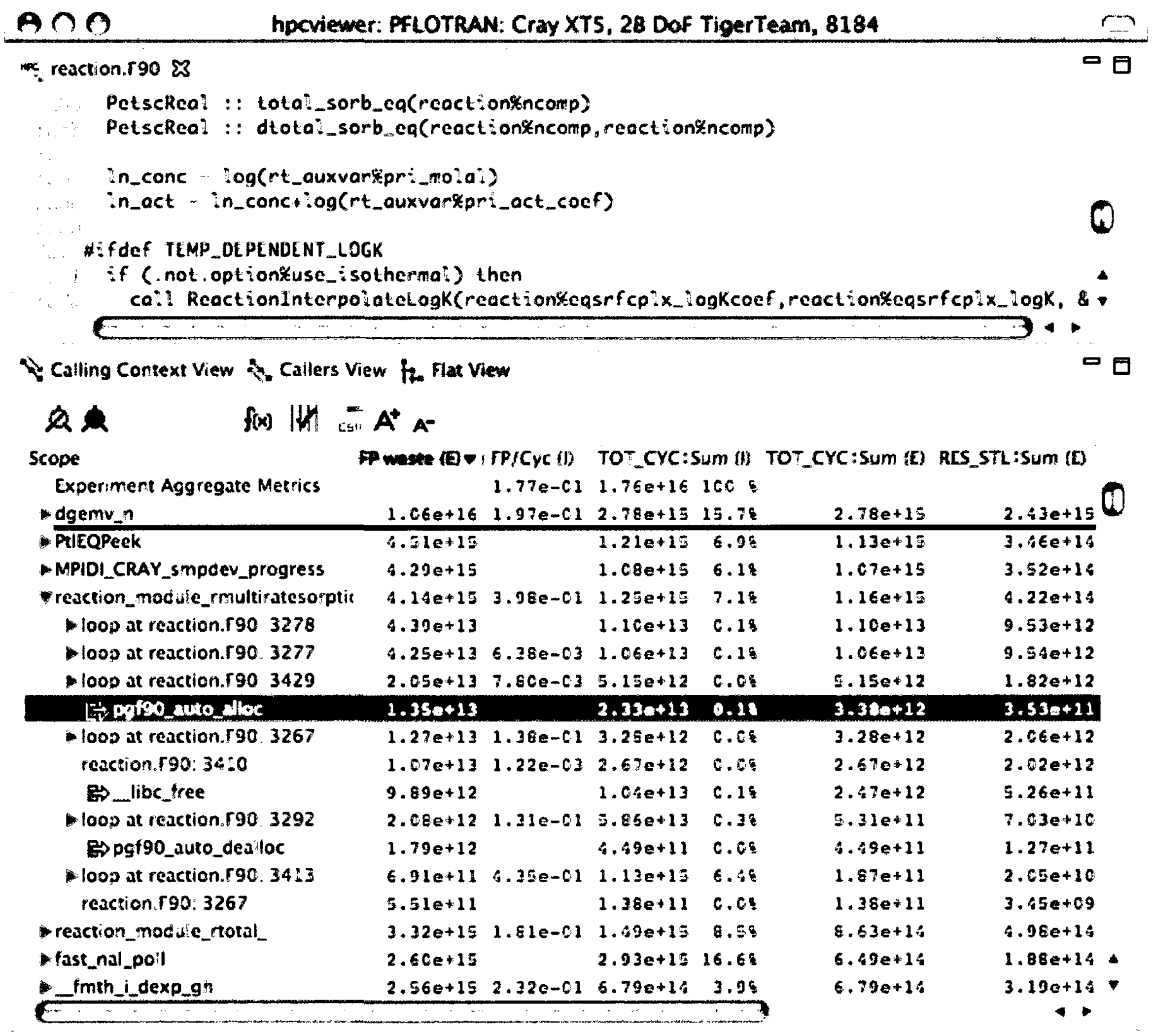
Figure 7.10 shows that 112.2% of the scaling loss in the application is attributed to the routine `MPIDI_CRAY_Progress_wait` and the routines that it calls. Percentage losses in any individual context are relative to total losses in the execution. While a scaling loss greater than 100% for a particular context might seem odd, it just means that there were scaling gains elsewhere in the execution that offset losses here. By looking up the call chain to see what calling sequence caused the program to incur scalability losses in `MPIDI_CRAY_Progress_wait`, we see that 80.6% of the scaling losses in the application can be traced to the use of `MPI_AllReduce`. Looking at the number of cycles spent in `MPI_AllReduce` in the 512 core and 8192 core executions, the poor scalability is clear: the 8192 core execution spends more time in `MPI_AllReduce` than in the 512-core execution.

Our bottom-up Callers view enables us to identify how losses associated with `MPI_AllReduce` are apportioned across various calling contexts that use this primitive. Looking two levels further up the call chain, we see that 28.1% of the total scaling losses come from the use of `MPI_AllReduce` on behalf of `VecAssemblyBegin` (a PETSc routine), which in turn was called to create a distributed vector out of an array read from an HDF5 file. In this case, the losses seem unavoidable and represent a fundamental limit to strong scalability. Other lines in the display show the breakdown of other scaling losses due calls to `MPI_AllReduce` from other contexts. Here, we have shown that HPCTOOLKIT's sampling-based measurements provide quantitative information about scaling losses and enable attribution of these losses to the full calling contexts in which they occur. Understanding scalability losses at this level of precision is essential if one's aim is to ameliorate them so that a code can scale well to full configurations of petascale systems.

Analyzing a large-scale execution on a Cray XT5

In this section, we use summary metrics from HPCTOOLKIT's `hpcprof-mpi` to analyze an 8184-core execution of PFLOTRAN on a Cray XT5. The test problem used for this section is a steady-state groundwater flow problem in heterogeneous porous media on an $850 \times 1000 \times 80$ element discretization with 15 chemical species per cell. We used HPCTOOLKIT to simultaneously collect four hardware counter metrics: cycles, floating point operations, resource stalls and L1 data cache misses. The effective sampling rate was about 925 samples/second and the overhead was less than 1%.

We first assess the overall floating point efficiency of PFLOTRAN's execution. Figure 7.11 shows a Flat view of PFLOTRAN's static structure with two metrics that highlight floating point utilization. The first metric, which is the sort key, provides



hpcviewer: PFLOTRAN: Cray XT5, 28 DoF TigerTeam, 8184

reaction.F90

```
    PetscReal :: total_sorb_eq(reaction%ncomp)
    PetscReal :: dtotal_sorb_eq(reaction%ncomp,reaction%ncomp)

    ln_conc = log(rt_auxvar%pri_molal)
    ln_act = ln_conc+log(rt_auxvar%pri_act_coef)

  #ifdef TEMP_DEPENDENT_LOGK
    if (.not.option%use_isothermal) then
      call ReactionInterpolateLogK(reaction%eqsrfcplx_logKcoef,reaction%eqsrfcplx_logK, &
```

Calling Context View   Callers View   Flat View

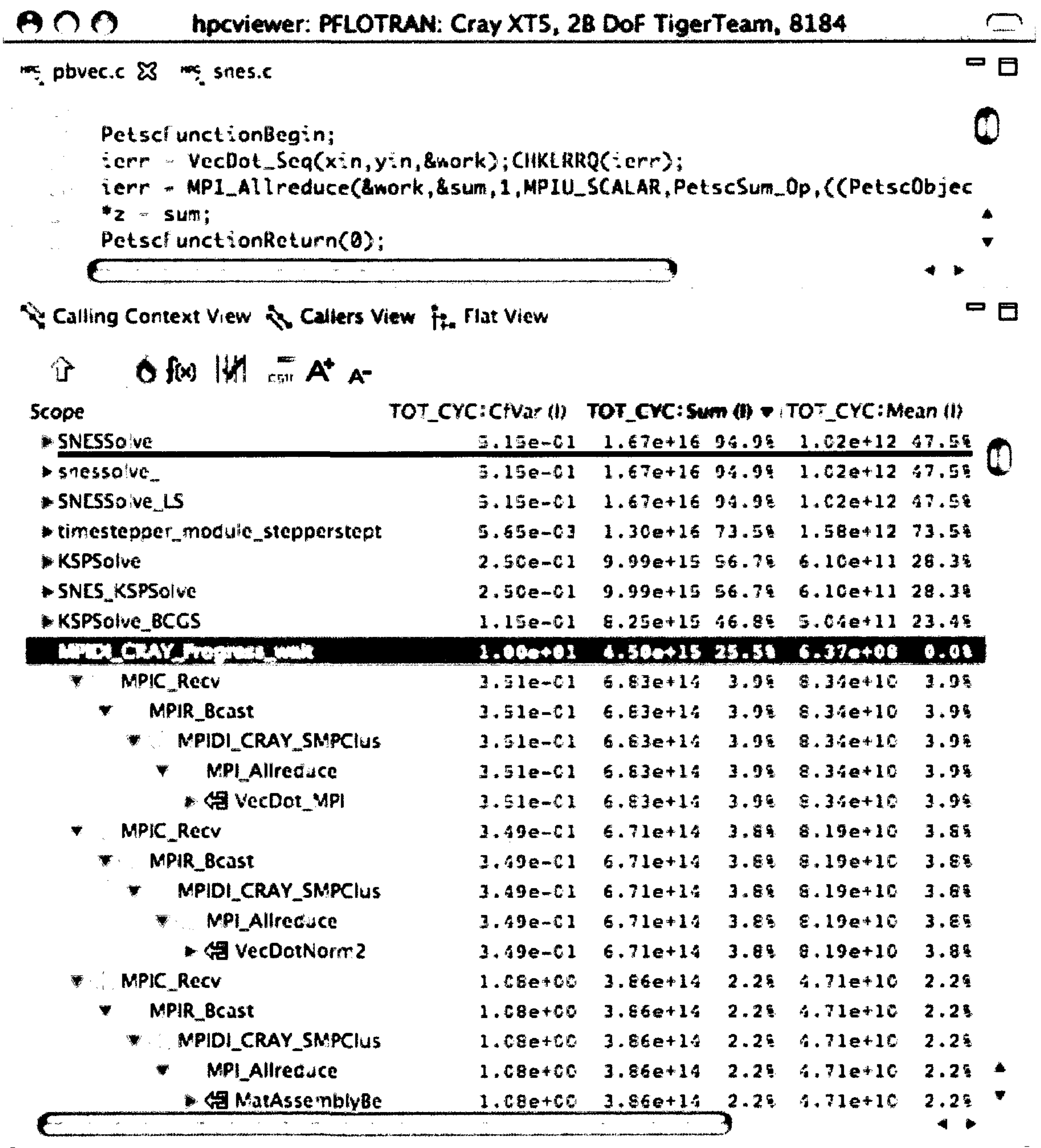
Scope	FP waste (E) ▾	FP/Cyc (I)	TOT_CYC:Sum (I)	TOT_CYC:Sum (E)	RES_STL:Sum (E)
Experiment Aggregate Metrics		1.77e-01	1.76e+16 100 %		
▶ dgemv_n	1.06e+16	1.97e-01	2.78e+15 15.7%	2.78e+15	2.43e+15
▶ PtlEQPeek	4.51e+15		1.21e+15 6.9%	1.13e+15	3.46e+14
▶ MPIDI_CRAY_smpdev_progress	4.29e+15		1.08e+15 6.1%	1.07e+15	3.52e+14
▼ reaction_module_rmultiratesorptic	4.14e+15	3.98e-01	1.25e+15 7.1%	1.16e+15	4.22e+14
▶ loop at reaction.F90 3278	4.39e+13		1.10e+13 0.1%	1.10e+13	9.53e+12
▶ loop at reaction.F90 3277	4.25e+13	6.38e-03	1.06e+13 0.1%	1.06e+13	9.54e+12
▶ loop at reaction.F90 3429	2.05e+13	7.80e-03	5.15e+12 0.0%	5.15e+12	1.82e+12
pgf90_auto_alloc	1.35e+13		2.33e+13 0.1%	3.38e+12	3.53e+11
▶ loop at reaction.F90 3267	1.27e+13	1.38e-01	3.28e+12 0.0%	3.28e+12	2.06e+12
reaction.F90: 3410	1.07e+13	1.22e-03	2.67e+12 0.0%	2.67e+12	2.02e+12
__libc_free	9.89e+12		1.04e+13 0.1%	2.47e+12	5.26e+11
▶ loop at reaction.F90 3292	2.08e+12	1.31e-01	5.86e+13 0.3%	5.31e+11	7.03e+10
pgf90_auto_dealloc	1.79e+12		4.49e+11 0.0%	4.49e+11	1.27e+11
▶ loop at reaction.F90 3413	6.91e+11	4.35e-01	1.13e+15 6.4%	1.87e+11	2.05e+10
reaction.F90: 3267	5.51e+11		1.38e+11 0.0%	1.38e+11	3.45e+09
▶ reaction_module_rtotal_	3.32e+15	1.81e-01	1.49e+15 8.5%	8.63e+14	4.98e+14
▶ fast_nal_poll	2.60e+15		2.93e+15 16.6%	6.49e+14	1.88e+14
▶ __fmth_i_dexp_gh	2.56e+15	2.32e-01	6.79e+14 3.9%	6.79e+14	3.19e+14

Figure 7.11: `hpcviewer`'s Flat view of floating point efficiency for PFLOTRAN on a Cray XT5.

a measure of floating point waste ('FP waste'). Each Opteron core on an XT5 node has a maximum peak performance of four double-precision floating point operations (FLOPs) per cycle. Therefore, we can compute floating point waste by subtracting actual floating point throughput from ideal throughput as follows: (4 × cycles) − FLOPs. The presentation tool computes this 'FP waste' metric using the cycle and FLOPs summary metrics that `hpcprof-mpi` generates by summing over all processes in the execution. This metric is exclusive, meaning that it excludes callees (hence the 'E' modifier). The second metric is inclusive FLOPs per cycle. Overall, this execution of PFLOTRAN performed 0.177 floating point operations per cycle, which is only 4.43% of peak.

The first routine that the 'FP waste' metric highlights is `dgemv_n`, which is underlined in Figure 7.11's lower pane. According to the TOT_CYC:Sum columns, this matrix-vector multiply routine consumes 15.7% of the execution's cycles, but has a floating point efficiency of 0.197 FLOPs/cycle. For comparison, the matrix-matrix multiply routine `dgemm_kernel` (not shown) delivers 2.27 floating point operations per cycle. The 'RES_STL:Sum' metric column shows the exclusive cycles that a processor core was stalled on any resource, over all processes in the execution. According to this metric, 87% ($2.43 \times 10^{15}/2.78 \times 10^{15}$) of the cycles spent in `dgemv_n` were attributed to resource stalls. Although not shown, `dgemv_n` accounts for 20.9% of the L1 data cache misses. This low efficiency bears further investigation.

Figure 7.11 also shows static structure within the `reaction_module_rmultiratesorption` routine, which accounts for 7.1% of the total cycles. At 0.398 FLOPs/cycle, this routine has better floating point throughput than `dgemv_n`. The static structure recovered by HPCToolkit exposes two important compiler transformations. The first is shown in the highlighted call site to `pgf90_auto_alloc`. This call site indicates that the Portland Group (PGI) compiler automatically allocated a temporary vector to implement the highlighted Fortran 90 statement shown in the source code (top) pane. The right hand side of this statement performs a vector logarithm and then adds the result to another vector. Although the whole statement could have been implemented with a loop and without a temporary, HPCToolkit shows that the compiler allocated an unnecessary temporary vector. The second transformation to notice is that the four top loops in this routine are actually compiler-generated scalarization loops to implement Fortran 90 vector operations.



```
PetscFunctionBegin;
ierr = VecDot_Seq(xin,yin,&work);CHKERRQ(ierr);
ierr = MPI_Allreduce(&work,&sum,1,MPIU_SCALAR,PetscSum_Op,((PetscObjec
*z = sum;
PetscFunctionReturn(0);
```

Scope	TOT_CYC:CfVar (I)	TOT_CYC:Sum (I) ▼		TOT_CYC:Mean (I)	
▶ SNESSolve	3.15e-01	1.67e+16	94.9%	1.02e+12	47.5%
▶ snessolve_	3.15e-01	1.67e+16	94.9%	1.02e+12	47.5%
▶ SNESSolve_LS	3.15e-01	1.67e+16	94.9%	1.02e+12	47.5%
▶ timestepper_module_stepperstept	5.65e-03	1.30e+16	73.5%	1.58e+12	73.5%
▶ KSPSolve	2.50e-01	9.99e+15	56.7%	6.10e+11	28.3%
▶ SNES_KSPSolve	2.50e-01	9.99e+15	56.7%	6.10e+11	28.3%
▶ KSPSolve_BCGS	1.15e-01	8.25e+15	46.8%	5.04e+11	23.4%
MPIDI_CRAY_Progress_wait	1.00e+01	4.50e+15	25.5%	6.37e+08	0.0%
▼ MPIC_Recv	3.51e-01	6.83e+14	3.9%	8.34e+10	3.9%
▼ MPIR_Bcast	3.51e-01	6.83e+14	3.9%	8.34e+10	3.9%
▼ MPIDI_CRAY_SMPClus	3.51e-01	6.83e+14	3.9%	8.34e+10	3.9%
▼ MPI_Allreduce	3.51e-01	6.83e+14	3.9%	8.34e+10	3.9%
▶ VecDot_MPI	3.51e-01	6.83e+14	3.9%	8.34e+10	3.9%
▼ MPIC_Recv	3.49e-01	6.71e+14	3.8%	8.19e+10	3.8%
▼ MPIR_Bcast	3.49e-01	6.71e+14	3.8%	8.19e+10	3.8%
▼ MPIDI_CRAY_SMPClus	3.49e-01	6.71e+14	3.8%	8.19e+10	3.8%
▼ MPI_Allreduce	3.49e-01	6.71e+14	3.8%	8.19e+10	3.8%
▶ VecDotNorm2	3.49e-01	6.71e+14	3.8%	8.19e+10	3.8%
▼ MPIC_Recv	1.08e+00	3.86e+14	2.2%	4.71e+10	2.2%
▼ MPIR_Bcast	1.08e+00	3.86e+14	2.2%	4.71e+10	2.2%
▼ MPIDI_CRAY_SMPClus	1.08e+00	3.86e+14	2.2%	4.71e+10	2.2%
▼ MPI_Allreduce	1.08e+00	3.86e+14	2.2%	4.71e+10	2.2%
▶ MatAssemblyBe	1.08e+00	3.86e+14	2.2%	4.71e+10	2.2%

Figure 7.12: `hpcviewer`'s Callers view of variance within PFLOTRAN on a Cray XT5.

We next perform two preliminary assessments of load balance. The first one uses a Callers view and the second one uses a Calling Context view.

Figure 7.12 shows a Callers view of PFLOTRAN sorted by total inclusive cycles, summed over all processes ('TOT_CYC:Sum (I)'). The left-most metric column labelled 'TOT_CYC:CfVar (I)' shows the corresponding coefficient of variation of cycles across all processes. The coefficient of variation is defined as standard deviation divided by mean and thus presents a relative measure of the standard deviation of cycles across all processors. For instance, a value of 2.0 means that the standard deviation has a magnitude of two means.

To find the most time-consuming routine with the most variation, we sort by inclusive summed cycles and then highlight the first routine with a large coefficient of variation.[4] The routine `SNESSolve` (underlined) is the first item in the list with a coefficient of variation larger than 0.02. Because this routine consumes 94.9% of the total cycles, has a large mean and a coefficient of variation of 0.515, it is a prime candidate for further study. Nevertheless, we defer discussing it for the moment and move down the list to the highlighted routine, which has an extremely large coefficient of variation of 10.0. This routine, `MPIDI_CRAY_Progress_wait`, is part of the low-level implementation of the Cray XT's MPI library. Although the routine has a relatively small mean, it accounts for 25.5% of the total (inclusive) cycles in the execution. The figure partially expands the top three of the many calling contexts from which this routine is called. Each call chain passes through `MPI_Allreduce` into the PETSc library (shown) and then into PFLOTRAN source-level routines (not shown). Nearly all the other contexts from which `MPIDI_CRAY_Progress_wait` is called also pass through `MPI_Allreduce` (not shown).

It is not surprising that a low-level communication routine would have a high coefficient of variation on a large-scale execution. Nevertheless, two things make the highlighted routine interesting. First, `MPI_Allreduce` is an MPI collective, which

[4]To find the highest-level routine with the most variation, we could sort by inclusive *mean* cycles.

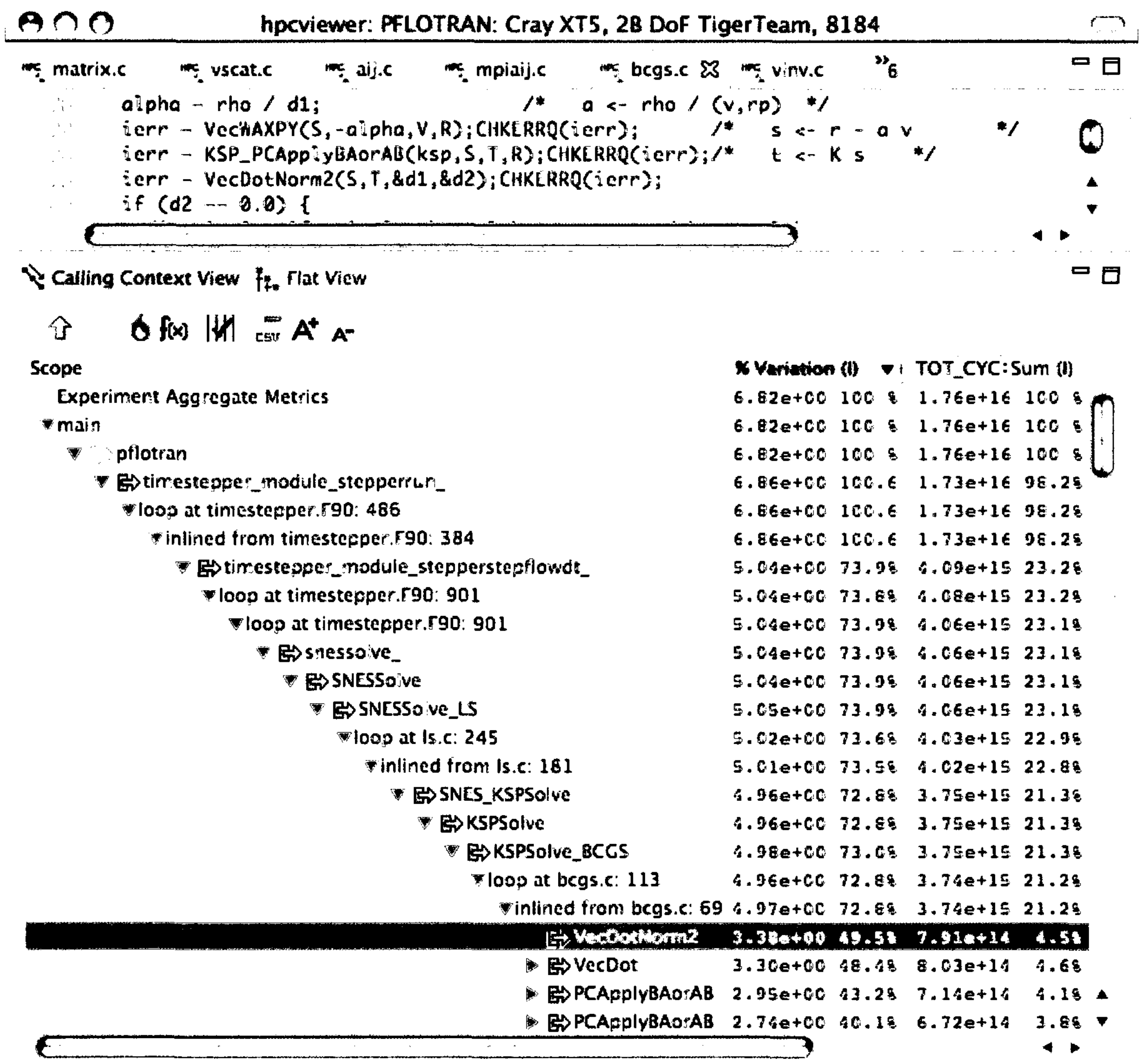
means that it must be invoked by all processes in the execution.[5] In the general case, the coefficient of variation statistic is unable to distinguish between variations caused by (1) some routine instances completing more slowly than others and (2) certain processes invoking a routine many more times than other processes. The fact that `MPI_Allreduce` is a collective means that it must be invoked the same number of times by each process, eliminating the ambiguity for this case. (By collecting return counts [60] we could resolve this ambiguity in the general case.) Second, as the figure shows, the Cray XT's implementation of `MPI_Allreduce` has several layers. The Callers view highlights these layers and shows that the important calling contexts of `MPIDI_CRAY_Progress_wait` all have a much lower coefficient of variation, but a much larger mean. HPCTOOLKIT's is able to show how variation and mean abruptly change precisely at this wait routine — even though it is within a vendor-supplied binary only library.

We next turn to the Calling Context view shown in Figure 7.13. We create a simple measure of variability by creating an inclusive '% Variation' metric based on cycles. For a given node in the Calling Context view, we take the maximum and minimum per-process cycle value of that node. The difference between the two represents the maximum variability for any given node. We then display the result as a percentage of the execution's total mean cycles:

$$\frac{\text{cycles}_{\text{max}} - \text{cycles}_{\text{min}}}{\text{total-cycles}_{\text{mean}}} \times 100\%$$

Although this metric can exaggerate the potential for improvement between back-to-back communication and computation, it provides a quick and effective assessment of variability. In the future, we plan to compute a more precise load imbalance met-

[5]The `MPI_Allreduce` is performed on the global MPI process group instead of a subgroup.



```
hpcviewer: PFLOTRAN: Cray XT5, 28 DoF TigerTeam, 8184

matrix.c   vscat.c   aij.c   mpiaij.c   bcgs.c   vinv.c   »6

      alpha = rho / d1;                    /*   a <- rho / (v,rp)  */
      ierr = VecWAXPY(S,-alpha,V,R);CHKERRQ(ierr);      /*   s <- r - a v        */
      ierr = KSP_PCApplyBAorAB(ksp,S,T,R);CHKERRQ(ierr);/*   t <- K s     */
      ierr = VecDotNorm2(S,T,&d1,&d2);CHKERRQ(ierr);
      if (d2 == 0.0) {
```

```
Calling Context View   Flat View

Scope                                                % Variation (I)   TOT_CYC:Sum (I)
 Experiment Aggregate Metrics                        6.82e+00 100 %    1.76e+16 100 %
 main                                                6.82e+00 100 %    1.76e+16 100 %
  pflotran                                           6.82e+00 100 %    1.76e+16 100 %
   timestepper_module_stepperrun_                    6.86e+00 100.6    1.73e+16 98.2%
    loop at timestepper.F90: 486                     6.86e+00 100.6    1.73e+16 98.2%
     inlined from timestepper.F90: 384               6.86e+00 100.6    1.73e+16 98.2%
      timestepper_module_stepperstepflowdt_          5.04e+00 73.9%    4.09e+15 23.2%
       loop at timestepper.F90: 901                  5.04e+00 73.6%    4.08e+15 23.2%
        loop at timestepper.F90: 901                 5.04e+00 73.9%    4.06e+15 23.1%
         snessolve_                                  5.04e+00 73.9%    4.06e+15 23.1%
          SNESSolve                                  5.04e+00 73.9%    4.06e+15 23.1%
           SNESSolve_LS                              5.05e+00 73.9%    4.06e+15 23.1%
            loop at ls.c: 245                        5.02e+00 73.6%    4.03e+15 22.9%
             inlined from ls.c: 181                  5.01e+00 73.5%    4.02e+15 22.8%
              SNES_KSPSolve                          4.96e+00 72.8%    3.75e+15 21.3%
               KSPSolve                              4.96e+00 72.8%    3.75e+15 21.3%
                KSPSolve_BCGS                        4.98e+00 73.0%    3.75e+15 21.3%
                 loop at bcgs.c: 113                 4.96e+00 72.8%    3.74e+15 21.2%
                  inlined from bcgs.c: 69            4.97e+00 72.6%    3.74e+15 21.2%
                   VecDotNorm2                       3.38e+00 49.5%    7.91e+14  4.5%
                   VecDot                            3.30e+00 48.4%    8.03e+14  4.6%
                   PCApplyBAorAB                     2.95e+00 43.2%    7.14e+14  4.1%
                   PCApplyBAorAB                     2.74e+00 40.1%    6.72e+14  3.8%
```

Figure 7.13: hpcviewer's Calling Context view of PFLOTRAN's variability on a Cray XT5.

ric that, like CrayPAT [48], distinguishes between computation and communication routines.

After computing the '% Variation' metric, we use hpcviewer's 'Hot path' button to automatically expand the unambiguous portion of the hot path with respect to the metric. The resulting path goes through PFLOTRAN's main time-stepper loop and into PETSc's SNESSolve routine, the routine that we passed over when discussing the Callers view. From `SNESSolve`, the call path descends several more layers into PETSc's `KSPSolve_BCGS` routine. The figure shows the four key call sites within this routine, the first of which is highlighted. Taken together, the four call sites are responsible for about 17% of the inclusive cycles in the execution. Although each call site has several descendants, in each case the variation is concentrated in a few spots. For the call sites to `VecDotNorm2` and `VecDot`, the variation exclusively derives from `MPI_Allreduce`. For the two call sites to `PCApplyBAorAB`, the variation is concentrated in computation from the `PetscSparseDenseMinusDot` and `PetscSparseDensePlusDot` macros and in delays manifested in `MPI_Waitany`. This result shows that HPCToolkit's automatically computed summary metrics can help a performance analyst quickly identify portions of a computation that contribute to the most important performance variations, even if those areas are within third party (PETSc) and vendor-supplied (MPI) libraries.

7.3.2 FLASH

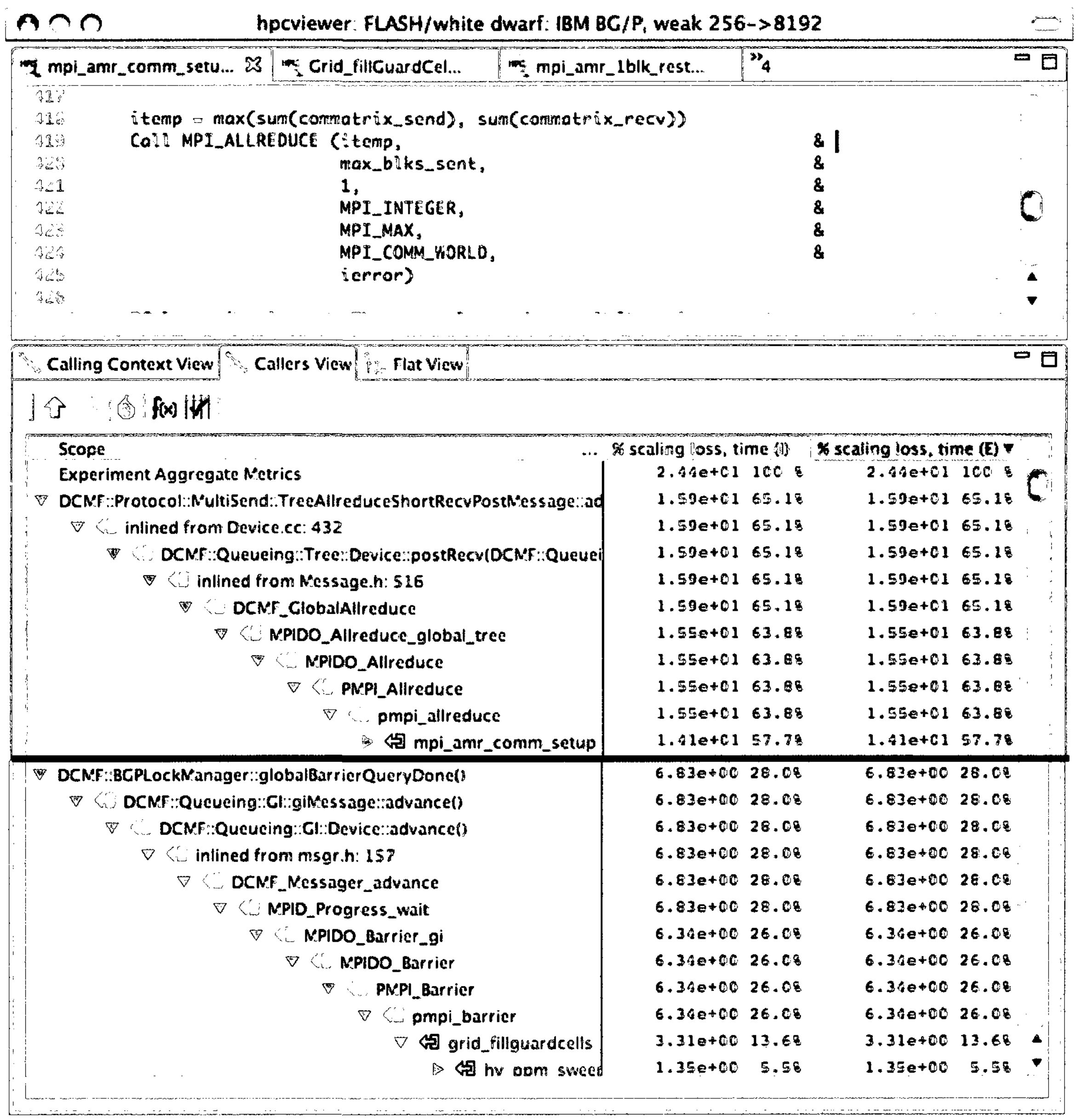
Next we consider FLASH [52], a code for modeling astrophysical thermonuclear flashes. We performed a weak scaling study of a white dwarf explosion by executing 256-core and 8192-core simulations on both Jaguar (Cray XT4) and Intrepid (IBM BlueGene/P). Both the input and the number of cores are 32x larger for the 8192-core execution. With perfect scaling, we would expect identical run times and call path profiles for both configurations.

A glance at the Calling Context view (top-down) for each scaling study (not shown) quickly reveals some differences between application scaling on the two systems. On BG/P there was a 24.4% loss of parallel efficiency (i.e., scaling loss), whereas on the XT4 the loss was larger, 32.5%. An execution of FLASH is divided into three phases, initialization (`Driver_initFlash`), simulation (`Driver_evolveFlash`), and finalization (Driver_finalizeFlash). In our benchmark runs, on BG/P 42.9% of the scaling loss (10.5% of the run time) came from initialization while the remaining 57.1% of the scaling loss (13.9% of the run time) came from simulation. In contrast, on the XT4, the initialization and simulation phases account for 54% and 46% of the scaling loss (about 17.6% and 15% of the run time), respectively. We consider the differences between the BG/P and XT4 in turn.

IBM BG/P

To quickly understand where the scaling losses for the initialization and simulation phases are aggregated, we turn to the bottom-up Callers view. Recall that the Callers view apportions the cost of a procedure (in context) to its callers. We sort the Callers 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: TreeAllreduce::advance and globalBarrierQueryDone.[6] To determine how these library calls relate to source-level code, we look up their call chains; the result is shown in Figure 7.14. When we look up the first call chain, we find calls to 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 MPI_Allreduce is a global max reduce for a scalar that occurs in code managing the adaptive mesh. HPCToolkit is uniquely able to pinpoint this one crucial call to MPI_Allreduce and distinguish it from several others that occur in the application.

[6]The full names are DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostMessage::advance and DCMF::BGPLockManager::globalBarrierQueryDone.



hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

mpi_amr_comm_setu... | Grid_fillGuardCel... | mpi_amr_1blk_rest... | »4

```
      itemp = max(sum(commatrix_send), sum(commatrix_recv))
      Call MPI_ALLREDUCE (itemp,                                &
                          max_blks_sent,                        &
                          1,                                    &
                          MPI_INTEGER,                          &
                          MPI_MAX,                              &
                          MPI_COMM_WORLD,                       &
                          ierror)
```

Calling Context View | Callers View | Flat View

Scope	% scaling loss, time (I)	% scaling loss, time (E) ▼
Experiment Aggregate Metrics	2.44e+01 100 %	2.44e+01 100 %
▽ DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostMessage::ad	1.59e+01 65.1%	1.59e+01 65.1%
▽ inlined from Device.cc: 432	1.59e+01 65.1%	1.59e+01 65.1%
▽ DCMF::Queueing::Tree::Device::postRecv(DCMF::Queuei	1.59e+01 65.1%	1.59e+01 65.1%
▽ inlined from Message.h: 516	1.59e+01 65.1%	1.59e+01 65.1%
▽ DCMF_GlobalAllreduce	1.59e+01 65.1%	1.59e+01 65.1%
▽ MPIDO_Allreduce_global_tree	1.55e+01 63.8%	1.55e+01 63.8%
▽ MPIDO_Allreduce	1.55e+01 63.8%	1.55e+01 63.8%
▽ PMPI_Allreduce	1.55e+01 63.8%	1.55e+01 63.8%
▽ pmpi_allreduce	1.55e+01 63.8%	1.55e+01 63.8%
mpi_amr_comm_setup	1.41e+01 57.7%	1.41e+01 57.7%
▽ DCMF::BGPLockManager::globalBarrierQueryDone()	6.83e+00 28.0%	6.83e+00 28.0%
▽ DCMF::Queueing::GI::giMessage::advance()	6.83e+00 28.0%	6.83e+00 28.0%
▽ DCMF::Queueing::GI::Device::advance()	6.83e+00 28.0%	6.83e+00 28.0%
▽ inlined from msgr.h: 157	6.83e+00 28.0%	6.83e+00 28.0%
▽ DCMF_Messager_advance	6.83e+00 28.0%	6.83e+00 28.0%
▽ MPID_Progress_wait	6.83e+00 28.0%	6.83e+00 28.0%
▽ MPIDO_Barrier_gi	6.34e+00 26.0%	6.34e+00 26.0%
▽ MPIDO_Barrier	6.34e+00 26.0%	6.34e+00 26.0%
▽ PMPI_Barrier	6.34e+00 26.0%	6.34e+00 26.0%
▽ pmpi_barrier	6.34e+00 26.0%	6.34e+00 26.0%
▽ grid_fillguardcells	3.31e+00 13.6%	3.31e+00 13.6%
▷ hv_ppm_swee	1.35e+00 5.5%	1.35e+00 5.5%

Figure 7.14: `hpcviewer`'s Callers view of scaling losses (wallclock) for FLASH on an IBM BG/P.

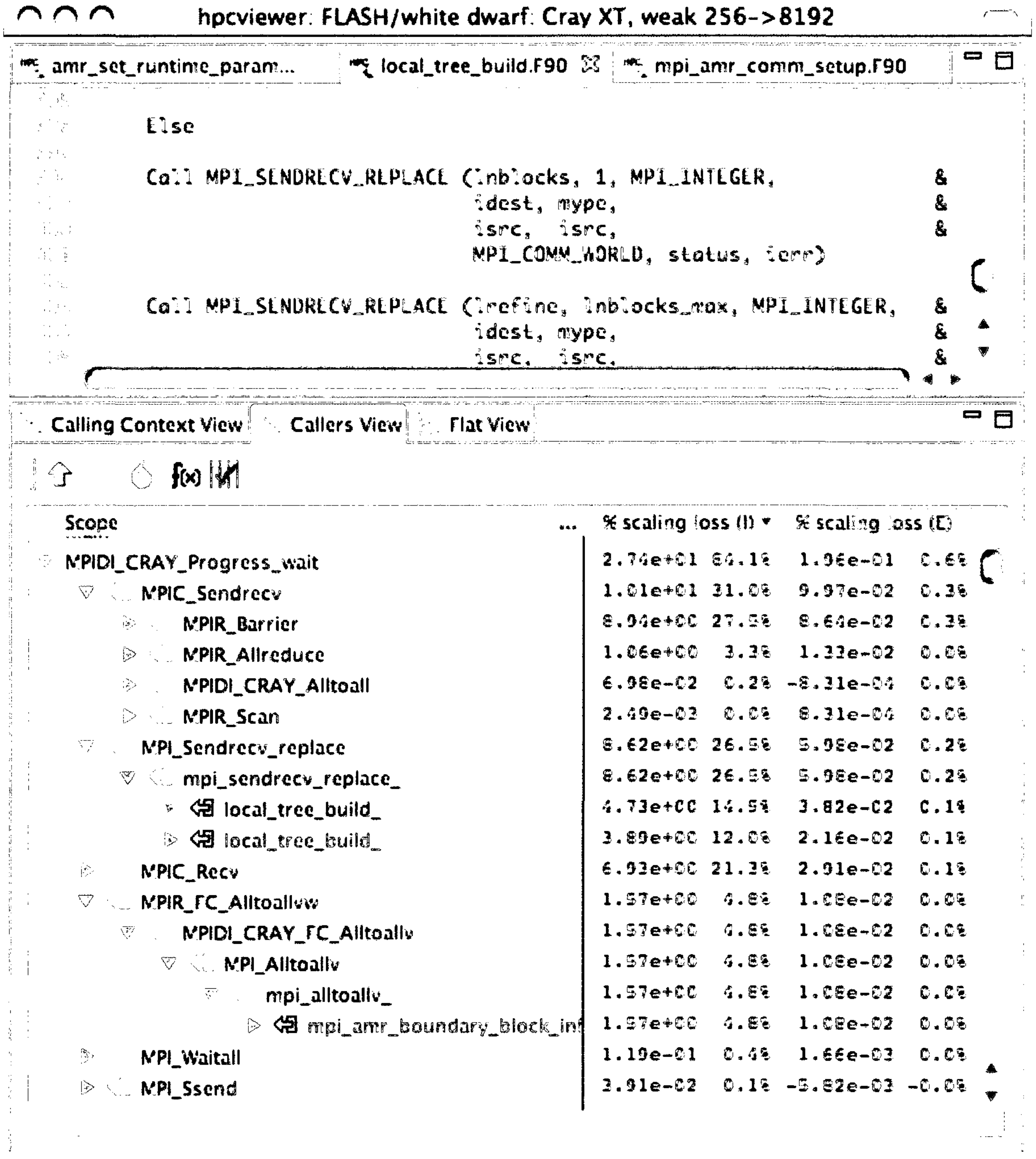
Next, we peer up the `globalBarrierQueryDone` call chain. The 'Hot path' button automatically expands the unambiguous portion of the hot path. By expanding this hot path automatically, we hone in on the one call to `MPI_Barrier` that disproportionately affects scaling. The call site is within `Grid_fillGuardCells` and is visible at the bottom of Figure 7.14; it accounts for 13.6% of the scaling loss (or 3.31% of the run time).

HPCToolkit enables us to quickly pinpoint exactly two calls that account for about 70% of FLASH's scaling loss on BG/P. It is interesting to note that the these two calls relate to two of BG/P specialized networks: the `MPI_Allreduce` to the global collective network and the `MPI_Barrier` to the global barrier network.

Cray XT4

In Figure 7.15, we turn to the same bottom-up Callers view that we used to analyze scaling losses on BG/P. We first sort by the exclusive scaling loss metric. However, because losses are more finely distributed than on BG/P, we sort by inclusive losses, which includes losses for callees. Since 100% of the scaling loss occurs in or below FLASH's 'main' routine, it appears at the top. The next procedure, `MPIDI_CRAY_Progress_Wait`, accounts for 84.1% of the scaling loss, is related to MPI communication, is shown in Figure 7.15. By inspecting the callers of this procedure, we see the breakdown of scaling losses among different types of communication. When using the `hpcviewer` interface interactively, one can expand the tree further to show the full context in the user program where these losses originate.

By inspecting the callers of `MPIC_Sendrecv`, one can see that 27.5% of the losses are due to barrier synchronization. Exploring a few levels deeper in the subtree rooted at `MPIR_Barrier`, we find that 12.1% of the scaling losses are due to barrier synchronization in the routine `amr_setup_runtime_parameters`. This routine contains a loop that iterates over each of the processor IDs. On each iteration of the loop, the processor whose ID is equal to the loop induction variable opens the input file, reads a set of program input parameters, and then closes the file. All processors meet at the bottom of the loop at a barrier. This represents a scaling bottleneck



hpcviewer: FLASH/white dwarf: Cray XT, weak 256->8192

```
      Else

      Call MPI_SENDRECV_REPLACE (lnblocks, 1, MPI_INTEGER,              &
                                 idest, mype,                            &
                                 isrc,  isrc,                            &
                                 MPI_COMM_WORLD, status, ierr)

      Call MPI_SENDRECV_REPLACE (lrefine, lnblocks_max, MPI_INTEGER,    &
                                 idest, mype,                            &
                                 isrc,  isrc,                            &
```

```
Scope                                           ...  % scaling loss (I) ▾  % scaling loss (E)
MPIDI_CRAY_Progress_wait                             2.74e+01 84.1%  1.96e-01  0.6%
  MPIC_Sendrecv                                      1.01e+01 31.0%  9.97e-02  0.3%
    MPIR_Barrier                                     8.94e+00 27.5%  8.64e-02  0.3%
    MPIR_Allreduce                                   1.06e+00  3.3%  1.33e-02  0.0%
    MPIDI_CRAY_Alltoall                              6.98e-02  0.2% -8.31e-04  0.0%
    MPIR_Scan                                        2.49e-03  0.0%  8.31e-04  0.0%
  MPI_Sendrecv_replace                               8.62e+00 26.5%  5.98e-02  0.2%
    mpi_sendrecv_replace_                            8.62e+00 26.5%  5.98e-02  0.2%
      local_tree_build_                              4.73e+00 14.5%  3.82e-02  0.1%
      local_tree_build_                              3.89e+00 12.0%  2.16e-02  0.1%
  MPIC_Recv                                          6.93e+00 21.3%  2.01e-02  0.1%
  MPIR_FC_Alltoallvw                                 1.57e+00  4.8%  1.08e-02  0.0%
    MPIDI_CRAY_FC_Alltoallv                          1.57e+00  4.8%  1.08e-02  0.0%
      MPI_Alltoallv                                  1.57e+00  4.8%  1.08e-02  0.0%
        mpi_alltoallv_                               1.57e+00  4.8%  1.08e-02  0.0%
          mpi_amr_boundary_block_inf                 1.57e+00  4.8%  1.08e-02  0.0%
  MPI_Waitall                                        1.19e-01  0.4%  1.66e-03  0.0%
  MPI_Ssend                                          3.91e-02  0.1% -5.82e-03 -0.0%
```

Figure 7.15: hpcviewer's Callers view of scaling losses (cycles) for FLASH on a Cray XT4.

whose severity increases with the number of processors. Fortunately, it has a remedy: one processor can open the input file and broadcast its contents to the rest of the processors; this change transforms the operation from $O(p)$ time to $O(\log p)$ time. Implementing and testing this solution on the Cray XT4 reduced the scaling loss due to `amr_setup_runtime_parameters` on 8192 cores to almost zero.

The highlighted line in Figure 7.15 shows one of two call sites for `local_tree_build`. This routine is part of the PARAMESH library [89] used by FLASH. Together, the function's two call sites account for 26.5% of the scaling losses and 8.62% of execution time on 8192 processors. This function builds an oct-tree as part of the structured adaptive mesh refinement. It scales poorly as the number of processors is increased. `local_tree_build` uses a communication pattern known as a *digital orrery* [14], in which all-to-all communication is implemented by circulating content from each processor around a ring of all processors. The communication phase takes $O(p)$ time. By consulting the Calling Context view (not shown) we found that `local_tree_build` is called both within FLASH's initialization and simulation phases. In the initialization phase it accounts for 18.5% of the scaling loss; in simulation it accounts for about 7.9%. We have had preliminary discussions with the FLASH team about how to improve the scaling of `local_tree_build`.

Figure 7.15 shows that 21.3% of the scaling loss results from `MPI_Recv`. Expanding the subtree rooted at that point, one discovers that almost all of these costs are due to calls to `MPI_AllReduce`. 15.5% of the total scaling loss is for `MPI_AllReduce` calls that are used to exchange information about blocks to set up communication prior to guard cell filling and flux conservation. In contrast, the same max reduction on BG/P accounts for 40.6% of the scaling loss.

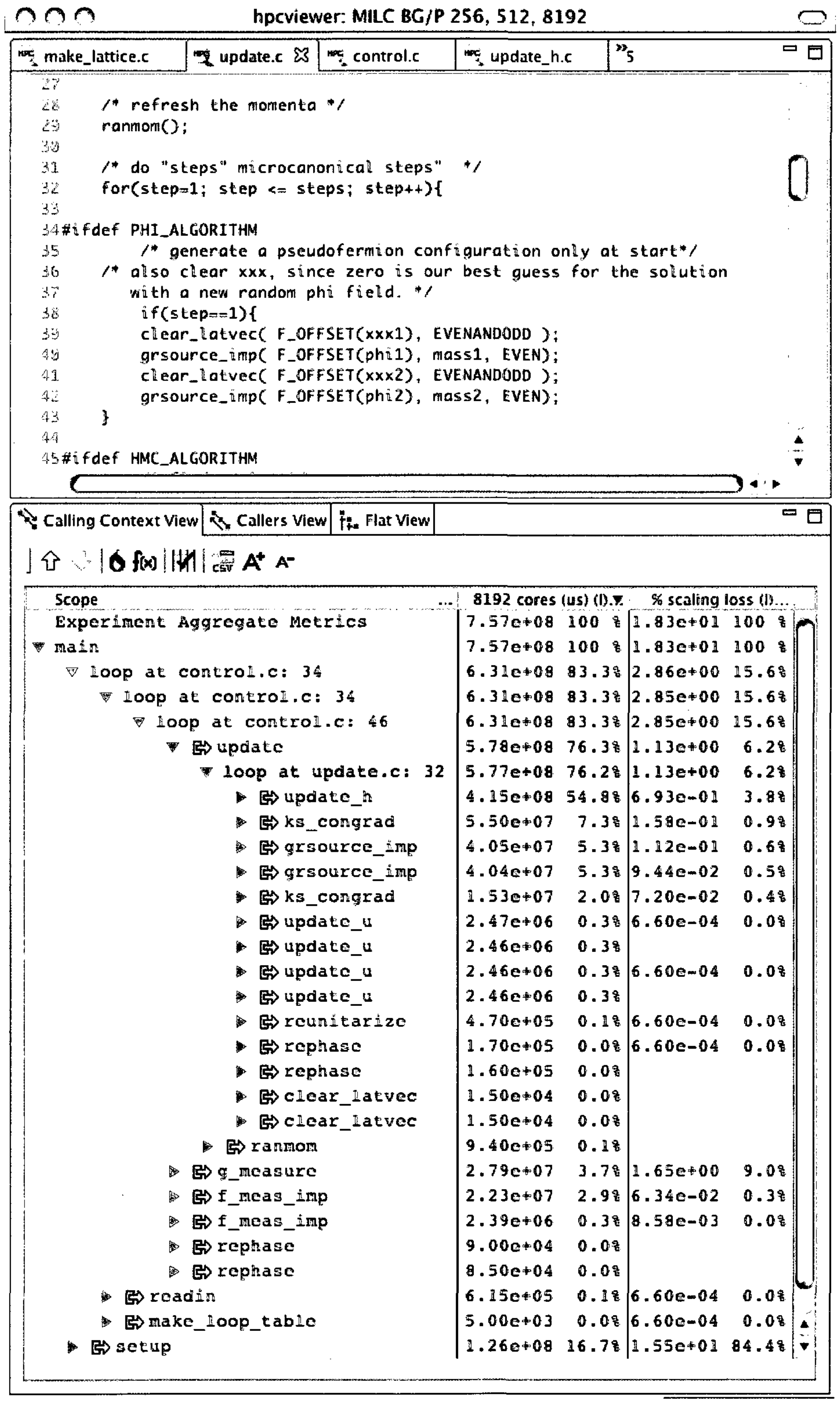
Summary

In the span of minutes, we have used HPCToolkit to pinpoint and quantify the scaling losses in each system deriving from just a few crucial call sites. HPCToolkit enables us to focus on the key areas and ignore the other losses, which are more finely distributed. Moreover, HPCToolkit obtains accurate call paths and precise measurements despite several layers of communication library calls for which no source code is available to application developers. The static program structure information computed by HPCToolkit even reports inlining within these layers.

7.3.3 MILC

The third application we analyze is a lattice quantum chromodynamics (QCD) simulation with dynamical Kogut-Susskind fermions from MILC, or MIMD Lattice Computation package [22]. MILC is a Lattice Quantum Chromodynamics code that is one of six application benchmarks in a suite used to evaluate bids for an NSF-funded petascale computer. We performed a weak scaling study by profiling 512-core and 8192-core simulations on both Jaguar (Cray XT4) and Intrepid (IBM Blue Gene/P). To keep execution time for the scaling study reasonable, we altered the default NSF problem size by decreasing the number of trajectories. In our scaling study, the input data and the number of cores are scaled by a factor of 16 so if scaling is ideal we should expect identical run times and call path profiles for both core counts.

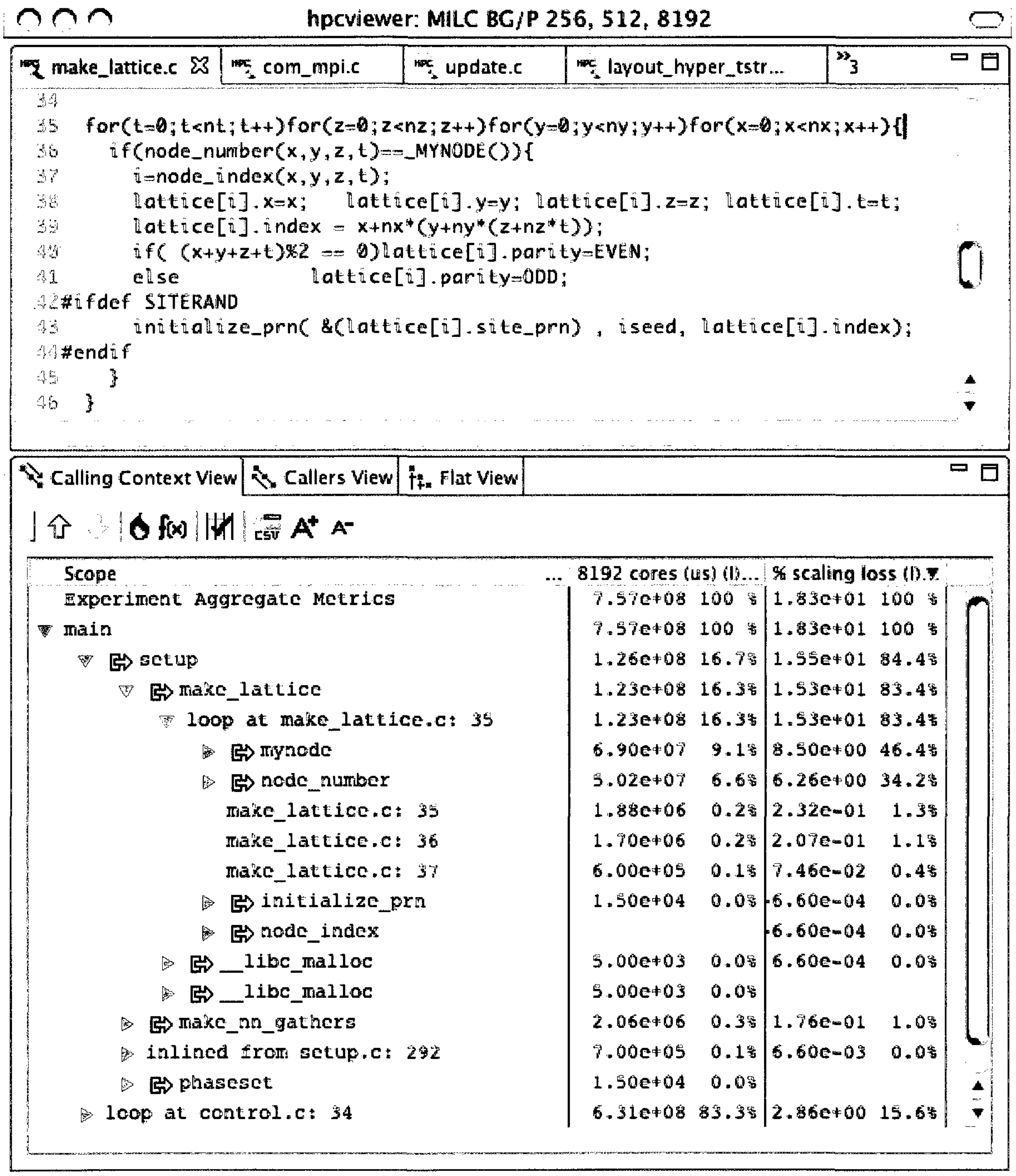
Figures 7.16 and 7.17 respectively focus on the breakdown of execution time and scaling losses (relative to a 512-core execution) for MILC in an 8192-core execution on a BG/P. The most time-consuming part of the code is the lattice update. In Figure 7.16, we can see that this phase accounts for 76.3% of the time on BG/P in an 8192-core execution; in an execution on a Cray XT4, this phase accounted for 83.3%



```
27
28     /* refresh the momenta */
29     ranmom();
30
31     /* do "steps" microcanonical steps"  */
32     for(step=1; step <= steps; step++){
33
34#ifdef PHI_ALGORITHM
35         /* generate a pseudofermion configuration only at start*/
36     /* also clear xxx, since zero is our best guess for the solution
37        with a new random phi field. */
38         if(step==1){
39         clear_latvec( F_OFFSET(xxx1), EVENANDODD );
40         grsource_imp( F_OFFSET(phi1), mass1, EVEN);
41         clear_latvec( F_OFFSET(xxx2), EVENANDODD );
42         grsource_imp( F_OFFSET(phi2), mass2, EVEN);
43     }
44
45#ifdef HMC_ALGORITHM
```

```
Scope                                        8192 cores (us) (I)  % scaling loss (I)...
Experiment Aggregate Metrics                 7.57e+08 100 %   1.83e+01 100 %
▼ main                                       7.57e+08 100 %   1.83e+01 100 %
  ▽ loop at control.c: 34                    6.31e+08 83.3%   2.86e+00 15.6%
    ▼ loop at control.c: 34                  6.31e+08 83.3%   2.85e+00 15.6%
      ▽ loop at control.c: 46                6.31e+08 83.3%   2.85e+00 15.6%
        ▼ update                             5.78e+08 76.3%   1.13e+00  6.2%
          ▼ loop at update.c: 32             5.77e+08 76.2%   1.13e+00  6.2%
            ▶ update_h                       4.15e+08 54.8%   6.93e-01  3.8%
            ▶ ks_congrad                     5.50e+07  7.3%   1.58e-01  0.9%
            ▶ grsource_imp                   4.05e+07  5.3%   1.12e-01  0.6%
            ▶ grsource_imp                   4.04e+07  5.3%   9.44e-02  0.5%
            ▶ ks_congrad                     1.53e+07  2.0%   7.20e-02  0.4%
            ▶ update_u                       2.47e+06  0.3%   6.60e-04  0.0%
            ▶ update_u                       2.46e+06  0.3%
            ▶ update_u                       2.46e+06  0.3%   6.60e-04  0.0%
            ▶ update_u                       2.46e+06  0.3%
            ▶ reunitarize                    4.70e+05  0.1%   6.60e-04  0.0%
            ▶ rephase                        1.70e+05  0.0%   6.60e-04  0.0%
            ▶ rephase                        1.60e+05  0.0%
            ▶ clear_latvec                   1.50e+04  0.0%
            ▶ clear_latvec                   1.50e+04  0.0%
          ▶ ranmom                           9.40e+05  0.1%
        ▶ g_measure                          2.79e+07  3.7%   1.65e+00  9.0%
        ▶ f_meas_imp                         2.23e+07  2.9%   6.34e-02  0.3%
        ▶ f_meas_imp                         2.39e+06  0.3%   8.58e-03  0.0%
        ▶ rephase                            9.00e+04  0.0%
        ▶ rephase                            8.50e+04  0.0%
    ▶ readin                                 6.15e+05  0.1%   6.60e-04  0.0%
    ▶ make_loop_table                        5.00e+03  0.0%   6.60e-04  0.0%
  ▶ setup                                    1.26e+08 16.7%   1.55e+01 84.4%
```

Figure 7.16: hpcviewer's Calling Context view of scaling losses (cycles) for MILC on a BG/P.



```
34
35  for(t=0;t<nt;t++)for(z=0;z<nz;z++)for(y=0;y<ny;y++)for(x=0;x<nx;x++){
36    if(node_number(x,y,z,t)==_MYNODE()){
37      i=node_index(x,y,z,t);
38      lattice[i].x=x;   lattice[i].y=y; lattice[i].z=z; lattice[i].t=t;
39      lattice[i].index = x+nx*(y+ny*(z+nz*t));
40      if( (x+y+z+t)%2 == 0)lattice[i].parity=EVEN;
41      else           lattice[i].parity=ODD;
42#ifdef SITERAND
43      initialize_prn( &(lattice[i].site_prn) , iseed, lattice[i].index);
44#endif
45    }
46  }
```

```
Scope                                    8192 cores (us) (I)...   % scaling loss (I)
Experiment Aggregate Metrics             7.57e+08 100 %           1.83e+01 100 %
▼ main                                   7.57e+08 100 %           1.83e+01 100 %
    ▼ setup                              1.26e+08 16.7%           1.55e+01 84.4%
        ▼ make_lattice                   1.23e+08 16.3%           1.53e+01 83.4%
            ▼ loop at make_lattice.c: 35 1.23e+08 16.3%           1.53e+01 83.4%
                ▶ mynode                 6.90e+07  9.1%           8.50e+00 46.4%
                ▶ node_number            5.02e+07  6.6%           6.26e+00 34.2%
                  make_lattice.c: 35     1.88e+06  0.2%           2.32e-01  1.3%
                  make_lattice.c: 36     1.70e+06  0.2%           2.07e-01  1.1%
                  make_lattice.c: 37     6.00e+05  0.1%           7.46e-02  0.4%
                ▶ initialize_prn         1.50e+04  0.0%           6.60e-04  0.0%
                ▶ node_index                                      6.60e-04  0.0%
            ▶ __libc_malloc              5.00e+03  0.0%           6.60e-04  0.0%
            ▶ __libc_malloc              5.00e+03  0.0%
        ▶ make_nn_gathers                2.06e+06  0.3%           1.76e-01  1.0%
        ▶ inlined from setup.c: 292      7.00e+05  0.1%           6.60e-03  0.0%
        ▶ phaseset                       1.50e+04  0.0%
    ▶ loop at control.c: 34              6.31e+08 83.3%           2.86e+00 15.6%
```

Figure 7.17: A closer look at scaling losses for MILC on a BG/P.

of the execution time. Within the update phase, execution time is distributed among routines called from the loop on line 32 in `update` and routines they call.

The total inclusive scaling loss for the application is shown in the yellow highlighted line as a percentage written in scientific notation. As shown in both figures, MILC has 18.3% total scaling loss on a BG/P. The lattice update phase scales relatively well and only has a 6.2% scaling loss. Most of the scaling losses in the update phase are due to waiting for scatter-gather communication to complete. For the short execution studied, Figure 7.17 shows that MILC's `setup` phase accounts for most of the scaling losses.

In Figure 7.17, the highlighted loop on line 35 in `make_lattice` accounts for 83.4% of the scaling loss and 16.3% of the run time. The reason that this loop causes a scaling loss is that it initializes local data for an MPI process by having each processor iterate over the *entire* lattice (all possible x, y, z, and t values), test each lattice point to see if it belongs to the current process, and then perform initialization only when the test succeeds. To avoid this kind of scaling loss, the application would need to be reworked to iterate only over a process's local lattice points rather than over the entire domain. Without a deeper understanding of the application, it is unclear whether this is feasible. Furthermore, it is not clear that losses due to initialization will be significant for production executions. The point of this example is not to focus on a shortcoming of the MILC code; rather, it is to show that HPCToolkit is capable of pinpointing and quantifying losses of this nature. Scaling losses need not be caused by communication.

7.4 Related Work

Most studies of application scaling on petascale systems have relied on manual analysis rather than sophisticated performance tools to understand scalability [4–6, 72]. Usually the analysis consists of (1) measuring key system performance characteristics using micro-benchmarks; (2) isolating scaling bottlenecks by creating scaling curves for different phases or procedures within the application; and (3) determining causes of bottlenecks by comparing an application's expected performance with its actual performance. Oliker et al. performed an early and insightful evaluation of application scaling on candidate petascale systems [111]. Even though they invested considerable effort in manual analysis, they had difficulty pinpointing and quantifying bottlenecks, and were only able to offer educated guesses such as "[the scalability loss] is probably due to the increase in [Allreduce operations]." HPCTOOLKIT could could directly pinpoint which operations were problematic and quantify the scaling loss for each. While the focus of these prior studies was to characterize system performance rather than advocate a method for pinpointing scaling bottlenecks, it was still necessary to understand such bottlenecks as part of their work.

Current performance tools for petascale systems identify scaling bottlenecks at the procedure level at best. The most important reason for this is that it is not feasible to make fine-grained measurements using instrumentation. Moreover, most of these tools require additional effort to analyze scaling. For example, Wright et al. used IPM [131] to distinguish between scaling bottlenecks in the communication or computation portions of an application [152]. To achieve low overhead ($< 5\%$), they collected profiles of instrumented MPI routines. These coarse measurements — only at the (MPI) procedure level, and without calling context — resulted in two deficiencies. First, because the application's computational component was not directly measured, the authors had to manually correct for communication-computation overlap to understand computational scaling. Second, to achieve further insight, the authors supplemented the measurements with labor-intensive analytical analysis.

mpiP [149] synchronously monitors MPI routines and collects a stack trace for each call. It qualitatively evaluates MPI scaling problems by using a rank-based correlation strategy. Because of this selective instrumentation, it incurs low overhead. However, it misses scaling problems in computational and non-MPI code.

Although other tools measure more comprehensively than IPM and mpiP, their measurements are still relatively coarse, typically at the procedure level. For example, tools such as TAU [90,129], SCALASCA [151,154], Cray's CrayPAT [48] and IBM's HPC Toolkit [74] collect the calling context of procedures rather than of statements. Because these tools collect calling context information using procedure-level instrumentation, their measurements are subject to distortion from measurement overhead associated with small procedures. By using asynchronous sampling, HPCToolkit is able to attribute costs to their full static and dynamic context with overhead of only a few percent [141], which in most cases is significantly less than procedure-level instrumentation [60]. HPCToolkit has the ability to collect the full calling context of any sample point, even exposing layers of calls in communication and math libraries for which source code is unavailable.

HPCToolkit's approach to computing scalability losses is similar to differential profiling support in other systems, e.g. [133]. However HPCToolkit is unique in its capability to attribute scalability losses to their full calling context, including inlined functions, loops and even individual statements. Furthermore, by providing Calling Context (top-down), Callers (bottom-up), and Flat views of scalability losses in context, HPCToolkit offers several different ways of analyzing the data. Different views provide different perspectives on bottlenecks that can make them easier to understand.

Although application traces can be very valuable (e.g., for identifying load imbalance), the volume of trace information makes scaling difficult. SCALASCA [154] selectively traces based on information from a prior profile. Others have explored (manual) selective tracing based on application characteristics [39]. Gamblin et al. have explored techniques for dynamically reducing the volume of trace information [63,64]. They report impressively low overheads, but they also, in part, use selective instrumentation that results in coarse measurements.

The STAT tool has been used on BG/L to sample call paths to aid parallel debugging at scale [83]. This tool uses third-party sampling mechanism that relies on daemons, running on I/O nodes, to periodically collect trace samples. In contrast, we use first-party sampling (in which the application samples itself), which requires no communication and permits much higher sampling rates.

7.5 Discussion

The key metric for parallel performance is scalability, either weak or strong. This is especially true at the petascale. Consequently, there is an acute need for application scientists to understand and address scaling bottlenecks in codes targeted for petascale systems. We have shown that it is possible, for minimal overhead, to pinpoint and quantify scaling bottlenecks on petascale systems to source code lines, in their full static and dynamic context using HPCToolkit. The analysis is rapid and its results are actionable.

Our results depend upon (1) accurate and precise asynchronous-sampling-based call path profiles — a form of measurement that until now has been unavailable on petascale systems; and (2) scalable analysis and presentation of those call path profiles. These two things enable us to apply Coarfa et al.'s powerful and elegant method for rapidly pinpointing and quantifying scaling bottlenecks [41] to emerging petascale applications. Past scaling analyses for petascale systems are either laborious, inaccurate (with respect to measurement), imprecise (with respect to bottleneck detection), or, in the case of the analysis we adopted, relied on tools that did not yet exist (i.e., tools for scalably analyzing and presenting asynchronous-sampling-based call path profiles).

It is a truism that a microkernel for a petascale platform should include what is necessary but dispense with excess: "just enough, but not too much!" The difficulty is in deciding what actually is necessary. We believe our results provide strong evidence that asynchronous-sampling-based performance analysis is so useful on these systems, that future microkernels for large-scale parallel systems should find a way to support it. Because petascale systems are designed for performance, it makes little sense to invest in computing resources that are powerful on paper but that cannot be exploited in practice.

HPCToolkit's support for sample-based performance analysis can provide insight into scalability and performance problems both within and across nodes. Gaining insight into node performance bottlenecks on large-scale parallel systems is a problem of growing importance. Today, parallel systems typically have between 4–16 cores per node. In emerging systems, we expect the core count per node to be higher. By sampling on hardware performance counters, one can distinguish between node performance bottlenecks caused by a variety of factors including inadequate instruction-level parallelism, memory latency, memory bandwidth, and contention.

In the near future, we plan to address a likely impediment to measuring full-system executions on petascale platforms. The file systems on petascale machines usually restrict the number of files per directory to less than the potential number of cores, which means that HPCTOOLKIT cannot open one file per thread to record its profiling data. We plan to address this by using a parallel I/O library such as SIONlib [59]. We also plan to rework HPCTOOLKIT's presentation tool to provide not only summary statistics for overall system performance, but also to preserve the ability to drill down into the details of performance on individual nodes. This will require managing thread-level metric data out-of-core.

Chapter 8

Conclusions

We claimed that it was possible to achieve unique, accurate, and actionable insight into the performance of fully optimized parallel programs by (1) measuring them with asynchronous-sampling-based call path profiles; (2) attributing the resulting binary-level measurements to source code structure; (3) analyzing measurements on-the-fly and post-mortem to highlight performance inefficiencies; and (4) presenting the resulting context-sensitive metrics in three complementary views. By actionable insight, we meant insight into an application's performance that justifies concrete actions such as determining how to resolve a performance bottleneck or deciding that there are no significant and worthwhile opportunities for performance improvement. To support this claim, we described several techniques for pinpointing performance problems in fully optimized serial, multithreaded and petascale programs. First, we provided a coherent framework for these techniques by sketching a unique and comprehensive performance analysis methodology. Second, we described the process of attributing very precise (instruction-level) measurements to full source-level static and dynamic calling contexts in two important execution environments — fully optimized applications and work-stealing run times — all for a run-time overhead of less than a few percent. Third, we described techniques for pinpointing and quantifying parallel inefficiencies such as parallel idleness, parallel overhead and lock contention in multithreaded executions. Finally, we showed how to diagnose scalability bottlenecks in petascale applications by scaling our measurement, analysis and presentation tools to support large-scale executions.

Measurement. Our work has striven to provide novel and actionable insight into the performance of parallel programs. To enable such insight, we have argued that it is essential to focus on accurate and precise performance measurements, because without such measurements analysis is unproductive. However, there is a natural tension between accuracy and precision: more precise measurements usually generate more overhead; and high overhead nearly always translates into high distortion and less accuracy. We observed that this trade-off is particularly acute for instrumentation-based strategies. For example, the dynamic-binary-instrumentation-based tool within Intel's Performance Tuning Utility toolkit collects less precise information than HPCToolkit but for an average overhead of over 400% on the SPEC 2006 integer benchmarks. We also noted that source code instrumentation, even when inducing low overhead through low precision, can introduce unintended blind spots that can obscure problems and interfere with compiler optimizations. Consequently, we grounded our work upon asynchronous-sampling-based measurement with controllable sampling rates.

To provide a theoretical foundation for our work, Appendix A shows how sampling-based measurement relates to standard statistical theory. We derive a simple formula to compute bounds for the measurement error within any particular code context and provided guidelines for choosing reasonable sampling periods. With reasonable sample periods, important regions of code receive enough samples to yield tight error bounds. For instance, it only takes 20 samples within a context over the course of an execution to obtain an error bound of $\pm 5\%$ for the cost of that context.

Because contextual measurements are often necessary for actionable insight into modular programs and because we are interested in the performance of real applications, we developed techniques for enabling asynchronous-sampling-based call path profiling on fully optimized parallel programs. Achieving both highly accurate and highly precise measurements for fully optimized binaries is a challenging problem. Nevertheless, by using novel on-the-fly binary analysis to enable stack unwinding, we demonstrated a capable call stack unwinder for fully optimized applications that induced average run-time overheads of 1–2%.

Thus, we have shown that it is possible to obtain highly precise call path profiles — statements in their full static and dynamic context — for very low overhead. Without extenuating circumstances, low overhead results in low distortion. For this reason, we argue that our techniques enable both highly accurate and highly precise contextual measurement. Our work shows that asynchronous sampling is an extremely useful measurement technique that can significantly mitigate the inelastic tension between accuracy and precision that instrumentation-based measurement approaches face.

To obtain the same quality of measurements for Cilk computations as we obtained for standard C, C++ and Fortran applications, we generalized call path profiling to recover logical call paths. We showed how to use logical call path profiling to relate an execution of a work-stealing-based multithreaded program back to its source-level representation. Although we focused on Cilk, logical call path profiling is applicable to any execution model for which native stack frames cannot serve as a proxy for a source-level call path. Such measurement capability will become imperative as programming models based on managed dynamic parallelism become more widespread.

Attribution. Once accurate measurements have been obtained, it is necessary to attribute them to source code. We desired an effective mechanism for projecting measurements at the level of machine code to higher levels of abstraction. Because we could not rely on standard compiler-generated information, we developed a binary analysis tool to recover static program structure directly from an application's binary. With this mapping from object code to source code, we correlated call path profiles to source code and enriched procedure instances with static context such as loop nests and inlined procedure instances. Because this process occurs post-mortem, it induces no run-time overhead.

Our binary analyses for enabling call path profiling and for recovering program structure uniquely complement asynchronous-sampling-based profiling of fully optimized binaries. Asynchronous-sampling-based profiling naturally observes *any* portion of the (user-level) execution. Indeed, this very property made unwinding difficult, motivating our binary analysis for stack unwinding. By joining both of these binary analyses with sampling-based profiling, we have demonstrated the ability to observe the behavior of vendor-only math and communication libraries and important compiler-inserted copy loops, in their full calling context. In other words, we have been able to measure *what actually executes* — as opposed to what one might assume executes given source code — and have then correlated those binary-level execution details as much as possible with source code structure.

Analysis & Presentation. To effectively understand the performance of multithreaded and petascale executions, we grounded our analysis and presentation upon our call path profiling technology.

For multithreaded applications, we focused on developing techniques for what we call blame shifting. That is, rather than pinpointing source-level contexts that simply exhibit parallel idleness (victims), we identified those that are responsible for causing it (perpetrators). We showed how to quantify and pinpoint idleness blame for applications based on both work-stealing and locks. We also showed how to quantify and pinpoint parallel overhead using a post-mortem analysis that induces no run-time overhead. For work-stealing-based applications, we showed that attributing parallel idleness and overhead to *logical* calling contexts enables one to quickly obtain unique insight into the run-time performance of Cilk programs. Our techniques demonstrated the importance of *third-party* metrics, i.e., metrics that reflect information about the execution state of other threads. To maintain the integrity of our measurements, we developed techniques that did not cause HPCToolkit itself to become a non-negligible source of contention and overhead.

For petascale executions, we showed how to apply the powerful technique of differencing call path profiles to petascale applications. Doing this required solving two problems. First, we demonstrated the ability to collect asynchronous-sampling-based call path profiles on petascale architectures. Second, we showed how to scalably analyze and present HPCToolkit's performance data. We argued that our results provide a compelling argument that because sampling-based measurement is so useful, petascale microkernels should support it.

Actionable Insight. We claim that these measurement, attribution, analysis, and presentation techniques result in novel and actionable insight into the performance of real-world applications executing on real architectures. With respect to *applications*, we have demonstrated insight across several different parallel programming models such as explicit threading (Pthreads), work stealing (Cilk), and distributed-memory Single Program Multiple Data (MPI). Moreover, we have focused on techniques that obtain precise measurements, incur low overhead, and which usually result in very high accuracy, even on fully optimized unmodified applications. With respect to *architectures*, we have developed techniques that can be applied to both multicore and petascale platforms. The fact that there are significant differences between these applications and architectures shows that our work has broad application.

Our techniques apply not only to current programming models but should adapt to the more dynamic models of parallelism that will likely become dominant in the future. For example, besides developing techniques for pinpointing parallel idleness in applications that use locks, we also targeted high-level programming models based on work stealing, an influential and practical dynamic scheduler.

Influence. Although HPCToolkit is an academic research project, it has been the recipient of growing interest and use by research groups, national labs and even industry. This is in large part due to the publication and dissemination of the various results described in this dissertation. Given the wide availability of other tools, both vendor-supplied and open source, this usage provides evidence that we are achieving our goal of providing unique and actionable insight.

For instance, within industry, the French computer company Bull is now shipping HPCToolkit as part of its software stack [30]. Samara Technology Group, like SiCortex before it, has adopted HPCToolkit as part of its core performance tool stack [108]. A group within IBM is currently evaluating HPCToolkit and, through personal communication, has provided very positive reviews. A group within WesternGeco (a division of Schlumberger) has used HPCToolkit to assess the performance of their proprietary software for analyzing seismic waves. They, also, through personal communication, were impressed with its feedback.

HPCToolkit is being actively used in other research projects. Researchers at the University of Texas are using HPCToolkit's performance data as input to an expert system that automatically diagnoses performance bottlenecks [31]. Rice University's Platform-Aware Compilation Environment (PACE) project [121] is using

HPCToolkit's performance data for automatically partitioning application source code and for feedback-directed optimization.

The HPCToolkit group has recently helped train HPC application scientists, both from national labs and from industry, in analyzing their applications using HPC-Toolkit. Examples of workshops include the 2009 Rice HPC Summer Institute [120], the 2009 CScADS Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies [35], and a 2010 workshop at Argonne National Laboratory [15]. The comments of a participant of this last workshop illustrate positive reception to the work we described in Chapters 3 and 7. This participant, a researcher in the area of computational molecular dynamics, commented that the overhead of HPCToolkit was very low. When asked how he knew, he responded that he saw no noticeable difference between a monitored and unmonitored run of his application. He added that he had been using a well-known instrumentation-based tool. With that tool, he had seen overheads of about 1000%. When he had tried to reduce this overhead by using selective instrumentation and throttling, he had found it to be labor intensive and ineffective.

In addition, the HPCToolkit group has recently been in contact with representatives from the National Renewable Energy Laboratory, STFC Daresbury Laboratory (UK) and the Swiss National Supercomputing Centre, among others. The University of Texas has independently included HPCToolkit in one of its own workshops [146].

Looking Forward. To obtain actionable insight into an application's performance, we have striven to make accurate and precise measurements. It is difficult to overestimate the importance of such measurements for systems that depend on performance analysis. For instance, accurate and detailed measurements are prerequisites for both successful feedback-directed optimization and automatic performance tuning. Similarly, although modeling can be extremely useful for performance prediction, it is necessary to validate a model's accuracy at small and large scales. Accurate fine-grain measurement provides this capability.

Despite the foundational nature of accurate and precise measurements, there is still a large and important gap that must be bridged to realize the goal of making performance tools widely useful to those who are not performance analysis experts. From the perspective of an application scientist, obtaining actionable performance insight currently requires wielding performance tools with expert control. We believe there are many ways to reduce the effort of performance analysis and tuning. We briefly discuss some ideas and open problems within the context of two very broad categories.

The first broad category is that of automatically presenting an insightful description of an execution's performance. Although our present work has fallen exclusively into this category, there are still many ways in which HPCToolkit is insufficient for making insightful high-level conclusions. One important area that our work does not address is transient behavior. To achieve low-overhead measurement, we have exclusively focused on profiling-based measurements — precisely because call path profiles do not grow with time but only with the number of unique contexts that a sample reveals. However, in large-scale parallel applications, some scalability problems are related to patterns of waiting that are not readily distinguishable with only a profile. To distinguish between different types of temporal bottlenecks, it is necessary to incorporate time into HPCToolkit's measurements. One approach we are investigating is collecting asynchronous-sampling-based call path traces [1]. To collect such a trace, one simply maintains both a calling context tree and a series of small (12 bytes) time-stamped records representing samples. We expect this to enable HPCToolkit to collect extremely rich trace information at large scales for much less overhead than instrumentation-based approaches.

Scale introduces many challenging problems. As a simple example, to scale HPC-Toolkit's measurement ability to hundreds of thousands of cores, it will be necessary to write profiles using a parallel I/O library such as SIONlib [59]. As another, to effectively present performance data, it will be necessary to develop ways to insightfully present more data than fits on computer displays. Currently, we are reworking HPCToolkit's presentation tool to provide not only summary statistics for overall system performance, but also to preserve the ability to drill down into the details of performance on individual nodes. In addition, very large-scale executions will probably cause problems for the sampling-based tracing described above. To address this, it will almost certainly be necessary to find ways to effectively compress temporal measurements. We expect that to effectively analyze the performance of very large-scale executions, tools will need to exploit statistical techniques more thoroughly. One possible approach is to employ statistical sampling at several levels instead of just within a thread.

Another challenge to automatically describing an execution's performance is that of node-level architecture. Multicore processors share many resources. For instance, most contemporary processors share at least one level of cache and a memory controller; some use hardware multithreading to share pipeline resources and hide latency; and a BlueGene/P chip contains shared network controllers. With shared caches and network controllers, assigning blame for resource contention difficult. For instance, frequent demand for a shared L3 cache by one thread may cause idling in another thread — a thread that might *not* be idling if located on another socket. With hardware multithreading, all functional units can be operating at peak efficiency even though one thread is stalled. In other words, certain types of contention may not be a problem!

Recently, there has been a surge of interest in heterogeneous architectures, primarily as a way to improve a node's performance per watt. With NVIDIA's recent introduction of Fermi, much more attention has focused on general purpose GPUs. With improved double-precision floating point support and ECC memory, Fermi addresses many of the drawbacks of prior GPU accelerators [110]. We are exploring ways to extend HPCToolkit's performance analysis to cover applications that use GPU accelerators.

Whereas the first set of open problems related to automatically describing a complete picture of an execution's performance, the second broad category is that of translating this basically *descriptive* information into *prescriptive* recommendations. In other words, if this dissertation has primarily focused on *obtaining* insight, then we would like to develop techniques that move toward automatically *applying* that insight. As an example, we would like a tool to highlight an important bottleneck and provide an explicit and targeted list of suggestions for resolving it. Such functionality is exactly what is needed to enable average developers to resolve most bottlenecks without the assistance of an expert performance analyst.

With the microprocessor industry's increasing reliance on parallel architectures, performance analysis is becoming more important outside the realm of high performance computing. Since processor-core clocks are not becoming appreciably faster — and even slowing — there are essentially two ways to improve an application's performance: create additional parallelism and optimize serial code regions. Both ways currently require manual performance tuning. This dissertation advances the performance analysis state-of-the-art to support both of these activities.

Appendix A

Theory of Sampling-Based Measurement

Since the the act of measuring an application's performance usually interferes with its execution and since interference usually distorts measurements, it is critical to minimize measurement interference. When instrumentation is applied to frequently executed program constructs, it often induces a proportionally large amount of overhead. In contrast, the overhead of sampling is proportional to the sampling frequency and not to execution frequency. Because overhead is nearly always combined with distortion, our methodology uses sampling to minimize measurement overhead.

This appendix, which especially complements Chapter 2, develops a foundation for our methodology by relating sampling-based measurement to statistical theory. In particular, it formalizes the concept of a profile gathered using statistical sampling. It also provides more than an intuitive justification for the claim that in most cases, sampling-based measurement can yield both high accuracy and precision.

The intention of this appendix is to set the practice of sampling-based measurement in an appropriate theoretical context. Consequently, it merely summarizes some complicating details of current hardware. For instance, one issue that often arises in current practice is the imprecision of hardware that assists in collecting sampling-based measurements. Therefore, a valuable question to ask is, given a particular set of hardware characteristics, can we make precise statements about the expected result or error of projecting low-level measurements to higher levels of source-level abstraction? This appendix leaves these questions to future work. However, it is worth noting that because there are commercial hardware solutions for the most important aspects of this imprecision, there is a possibility that the practical importance of these open questions will diminish in the future.

A.1 A Sampling-based Measurement Strategy

Perhaps the most well-known use of statistical sampling is for surveys and opinion polling. In this context, sampling is used to estimate general characteristics of a population from a small sample. The primary motivation for sampling is usually that working with a small sample is much less costly and time-consuming than canvassing an entire population. One difference between surveys and program measurement is that in the latter, measurement directly and immediately changes the target population by interrupting program execution and thereby increasing execution time.[1] We hope to use sampling to interrupt a program relatively infrequently and to collect a relatively samll amount of representative data.

We state our goal precisely as follows. Given program thread P with input I, use statistical sampling to estimate resource metrics for resource R over the important static and dynamic calling contexts of P's execution. We will focus on profiling, but our discussion also applies to sampling-based tracing.

Sampling theory is concerned with describing how well a sample characterizes the population from which it was drawn. Therefore, we first define two relevant populations:

- $\mathcal{P}_R$: $x_1, x_2, \ldots, x_{N_R}$. Given a resource R, this population represents monotonically increasing values for R, quantized into discrete units, where each unit is

[1]Opinion surveyors using tendentious questions may also wish to nudge a respondent's opinions, but this seems to be less direct and immediate.

1_R. For instance, the population $1, 2, 3, 4, 5$ could represent an execution that consumes 5 units of resource R. This population is finite but N_R cannot be known until P's execution completes.

- $\mathcal{P}_P$: $y_1, y_2, \ldots, y_N$. This population represents the consumption of resource R for each instruction in the (dynamic) instruction stream of P's execution. Thus, y_j gives the the number of units of R that the j^{th} instruction in the dynamic stream consumes. For example, the population $0, 4, 1$ could represent an execution of three instructions that consumes a total of 5 units of resource R. This population is finite but cannot be known until after P's execution.

  This population is stratified by dynamic calling context. Thus, each $y_j \in \mathcal{P}_P$ belongs to exactly one dynamic calling context given by $\mathcal{C}(j)$. At times, it will be useful to speak only of the instruction instance indices within $\mathcal{P}_P$. We can think of this as a projection and represent it as $\mathcal{P}_{P|\mathcal{I}}$.

Now, based on these populations, we define metric values for resource R over P's execution. Let Y be the total resource usage of R during P's execution. We have

$$Y = \sum_{j=1}^{N} y_j = N_R$$

where each y_j is from population $\mathcal{P}_P$. To define total resource usage Y_c for any context c during P's execution, we let $\mathbf{y}_c = \{y_j | \mathcal{C}(j) = c\}$. Then,

$$Y_c = \sum_{y_j \in \mathbf{y}_c} y_j$$

We can now restate our goal more precisely, which is to derive an estimator $\hat{Y}_c$ of the actual resource metric total Y_c for any given static or dynamic context c that is part of P's execution. To compute these estimates, we need a sample of population

$\mathcal{P}_P$. We can obtain this sample in two ways. The first is to directly obtain a sample using instruction-based sampling. The second is to indirectly obtain a sample by using event-based sampling of $\mathcal{P}_R$ and then mapping that to population $\mathcal{P}_P$.

A.1.1 Instruction-based sampling

To use instruction-based sampling, we *systematically sample* population $\mathcal{P}_P$ with period p to obtain a *simple random sample.* We pick a random starting point y_i (where $1 \leq i \leq p$) and then select every p^{th} item thereafter to obtain the sample $\mathbf{y} = \{y_i, y_{i+p}, y_{i+2p}, \ldots, y_n\}$, where subscripts are relative to population $\mathcal{P}_P$. Assume that there is no correlation between the sample period p and the sample points within population $\mathcal{P}_P$. Since each sampled instruction tracks resource usage of the sampled instruction j, we can directly compute its consumption y_j. To obtain an estimate $\hat{Y}$ for the total resource consumption Y of the program, we sum every y_j in the sample $\mathbf{y}$, and scale the result by p, the ratio of unsampled to sampled instruction instances:

$$\hat{Y} = p \sum_{y_j \in \mathbf{y}} y_j \tag{A.1}$$

Similarly, to estimate $\hat{Y}_c$ for a given program context c, we let $\mathbf{y}_c$ refer to all the y_j in context c, sum the result, and scale by p:

$$\hat{Y}_c = p \sum_{y_j \in \mathbf{y}_c} y_j \tag{A.2}$$

For small contexts, p may not be an accurate estimate of the ratio of unsampled to sampled instruction instances. We will address the concern in more detail in the context of event-based sampling.

A.1.2 Event-based sampling

For event-based sampling, we first *systematically sample* population $\mathcal{P}_R$ with period p to obtain a *simple random sample.* We pick a random starting point x_i (where $1 \leq i \leq p$) and then select every p^{th} item thereafter to obtain the sample $x_i, x_{i+p}, x_{i+2p}, \ldots, x_{n_R}$, where subscripts are relative to population $\mathcal{P}_R$. Observe that $n_R = i + (n-1)p \leq N_R$, where n is the number of samples.

To obtain a sample of population $\mathcal{P}_P$, we rely on a mapping $\mathcal{M} : \mathcal{P}_R \mapsto \mathcal{P}_{P|\mathcal{I}}$ that associates any given member of population $\mathcal{P}_R$ with its corresponding instruction instance in $\mathcal{P}_P$. Thus, for each x_i in the sample, we obtain a corresponding $y_j \in \mathcal{P}_P$ such that $\mathcal{M}(x_i) = j$. This yields a sample $\mathbf{y} = \{y_{\mathcal{M}(x_i)}, y_{\mathcal{M}(x_{i+p})}, \ldots, y_{\mathcal{M}(x_{n_R})}\}$ of population $\mathcal{P}_P$, where subscripts are relative to $\mathcal{P}_P$. Assume that there is no correlation between the sample period p and the sample points within population $\mathcal{P}_P$.

The next step is to define the value of each y_j in the sample $\mathbf{y}$. In theory, with a very precise and exhaustive mapping $\mathcal{M}$ we could obtain a very precise value for each y_j, as with instruction-based sampling. For example, given any $y_j \in \mathcal{P}_P$, we would precisely know the set of resource units $\mathbf{x}_j$ that were consumed during the execution of instruction instance j: $\mathbf{x}_j = \{x_i | \mathcal{M}(x_i) = j\}$. Then, to compute the value y_j — the total number of resource units consumed during instruction instance j's execution — we say $y_j \equiv (\mathsf{max}(\mathbf{x}_j) - \mathsf{min}(\mathbf{x}_j)) + 1_R$, where 1_R represents 1 unit of resource R. However, this is not practical because it would require that some combination of hardware and software ensure that $\mathcal{M}$ is exhaustive. Consequently, we use the sampling period p as an estimator for the value of each y_j in the sample $\mathbf{y}$. That is, when we sample population $\mathcal{P}_R$ and use the mapping $\mathcal{M}$ to obtain the associated instruction instance j, we assign p units of resource R to y_j. This results in the following resource metric total estimator $\hat{Y}$ for the sample $\mathbf{y}$ with n sample points:

$$\hat{Y} = \sum_{y_j \in \mathbf{y}} y_j = \sum_{1}^{n} p = p \sum_{1}^{n} 1 = pn \tag{A.3}$$

Similarly, the estimator $\hat{Y}_c$ for context c is

$$\hat{Y}_c = \sum_{y_j \in \mathbf{y}_c} y_j = \sum_{1}^{n_c} p = p \sum_{1}^{n_c} 1 = pn_c \tag{A.4}$$

where $\mathbf{y}_c = \{y_j | \mathcal{C}(y_j) = c\}$ represents the sample points in context c and $n_c = |\mathbf{y}_c|$.

It may initially appear that this estimator is inaccurate because any given instruction may not have consumed p units of the resource under consideration. Although this may be true at the precision of an instruction, recall that our primary goal is obtaining an accurate estimator $\hat{Y}_c$ for the resource metric total Y_c of a program context c, where c is a statement, loop or procedure in its calling context. In addition, we are usually interested in aggregating multiple instances of the same context to create a profile, which naturally tends to improve the estimator $\hat{Y}_c$. Finally, although we defer the details to Section A.2, we can regard p as yielding an unbiased estimator, which means that there is no difference between the expected value of the estimator $\hat{Y}_c$ and the value of Y_c (the value being estimated).

A.1.3 Practical considerations

In practice, although there is currently little support for instruction-based sampling, most microprocessors and operating systems support thread-level event-based sampling. In particular, the performance monitoring unit (PMU) for most microprocessors of interest is powerful enough to measure a wide range of resources at the thread level and to generate interrupts. To use a typical microprocessor's PMU to collect an event-based sample, we program the PMU to monitor resource R and generate a per-thread sampling interrupt with period p.[2] When a sampling interrupt is generated, the PMU associates it with an instruction in the executing program. Thus, the PMU implements the mapping $\mathcal{M}$ relating $\mathcal{P}_R$ and $\mathcal{P}_{P|\mathcal{I}}$.

Unfortunately, this mapping $\mathcal{M}$ is often imprecise because of the difficulty of pinpointing the instruction that consumed the p^{th} resource unit in the context of superscalar, out-of-order, pipelined execution. This effect is called PMU *skid*. Our methodology effectively copes with skid by aggregating metrics at the loop and procedure level, where the effects of imprecision are minimal. For instance consider an out-of-order pipeline, the source of most of these troubles. As long the number of instructions in the pipeline's reorder buffer is small compared to the total number of instruction instances in the loop is small (the number of loop iterations multiplied by the static instruction count), loop-level attribution is very precise. We can be more precise if we have a distribution that models the PMU's skid. In this case we can compare the expected value of the PMU's skid with the total number of instruction instances in the loop.

Some PMU designs have attempted to address the problem of imprecise mappings. For instance, some PMUs support precise attribution, though with important caveats [135]. Others have used instruction-based sampling, where the PMU directly associates a sampled instruction j with its resource usage y_j [47, 54]. We welcome these improved designs, but currently cannot rely on their wide availability.

One potential problem of systematic sampling is that a correlation may exist between the sample period and the sample points of population $\mathcal{P}_P$. For example, when sampling cycles with period p, it may be the case that a loop has a trip count of p

[2]Note that it may not be possible to measure all resources at the thread level; notable examples of this in recent multicore processors are 'uncore' events that monitor shared chip-level resources.

cycles. In this case, the sample can no longer be considered a simple random sample. Fortunately, in practice this is a minor concern. The complexity of binary code (because of compiler optimizations), operating systems, and of architectures (because of superscalar, out-of-order, pipelined execution) makes it difficult to establish extensive periodic behavior. Moreover, randomizing the period's low order bits both makes correlations extremely unlikely and has negligible effect on the quality of the estimators $\hat{Y}$ and $\hat{Y}_c$.

A.2 Analyzing the Strategy

We would like to answer several questions about this strategy. For a given program context c, how accurate is the estimator $\hat{Y}_c$? How many samples does one need in context c to provide a certain confidence in the value of $\hat{Y}_c$? How does one select a good sample period? In answering these questions, we focus on event-based sampling because it is so dominant, though the results naturally extend to instruction-based sampling.

Our analysis is related to the method used to estimate totals over subpopulations when neither the resource metric total $Y = N_R$ nor the actual number of instruction instances N_c in context c is known [42, §2.13]; cf. [42, §8.12, §5A.14]. However, we have adapted several aspects of it to the particulars of using sampling to gather performance profiles.

We emphasize again that our intention here is to set the practice of sampling-based measurement in an appropriate theoretical context, though we comment on how this analysis can be appropriately extended to account for things like PMU skid.

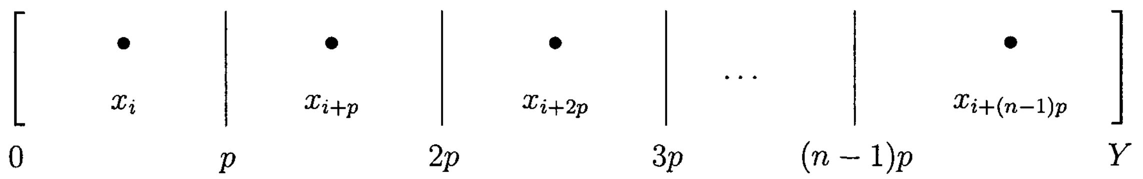


Figure A.1: A systematic sample drawn from resource population $\mathcal{P}_R$ using period p, where sample points are represented with bullets (•).

A.2.1 Error bounds for $\hat{Y}_c$

Recall that when taking a systematic sample of population $\mathcal{P}_R$ using period p, we pick a random starting point x_i (where $1 \leq i \leq p$) to obtain the simple random sample $x_i, x_{i+p}, x_{i+2p}, \ldots, x_{n_R}$, where $n_R = i+(n-1)p$ and n is the number of samples. While we cannot initialize a sample source exactly as the program begins execution, initialization occurs early enough during process initialization that bounding i by $1 \leq i \leq p$ is a good estimate. Assume $n \geq 1$. Figure A.1 represents such a sample of a population $\mathcal{P}_R$ for resource R. At the start of the execution, 0 units of resource R have been consumed; at the end, the total is $Y = N_R$. Consequently, we have

$$Y = i + (n-1)p + j = N_R \qquad \text{where } 1 \leq i \leq p \text{ and } 0 \leq j < p. \tag{A.5}$$

Assuming that samples can be handled instantaneously, the sample period p divides the entire execution into n regions, where the first $n-1$ regions are of size p and the last region is of size $i + j$.

Although the assumption of instantaneous handling of a sample may sound unrealistic, it is often reasonable in practice. This may be seen in two ways. First, by using reasonable sampling periods such as hundreds to thousands of samples/second, the overhead of profiling is extremely low (a few percent). In comparison, the Digital Continuous Profiling Infrastructure, which collected (system-wide) *flat* profiles, sampled at a rate of 5200 samples/second for an overhead of 0.5-3.0% [10]. Second, it is possible to self-correct for most of the resources consumed while processing a sample by resetting the sample source just before the sample handler returns control to the application thread.[3]

From Equation A.3, we know that the total estimator for Y is $\hat{Y} = pn$. Rearranging Equation A.5, we obtain:

$$Y = pn + (i + j - p) = \hat{Y} + (i + j - p) \tag{A.6}$$

Clearly, if n is large, then $\hat{Y} \gg (i + j - p)$ and the estimator $\hat{Y}$ is very good.

To derive error bounds for estimator $\hat{Y}$, we use Equation A.6 to compute the minimum and maximum values of Y with respect to $\hat{Y}$ and period p:

$$Y_{\mathsf{min}} = pn - (p-1) = \hat{Y} - (p-1) \qquad i = 1 \text{ and } j = 0 \tag{A.7}$$

$$Y_{\mathsf{max}} = pn + (p-1) = \hat{Y} + (p-1) \qquad i = p \text{ and } j = p-1 \tag{A.8}$$

Joining Equations A.7 and A.8 yields the following bounds for Y in terms of $\hat{Y}$ and p:

$$\hat{Y} - (p-1) \le Y \le \hat{Y} + (p-1) \tag{A.9}$$

This result says that given the total estimator $\hat{Y}$ and period p, we can compute an upper and lower bound for the actual total Y. An alternative way to derive

[3]Of course, processing a sample will have some side effects, such as a certain amount of cache pollution. We have attempted to minimize these effects in HPCToolkit.

Equation A.9 is to observe that by the pigeonhole principle, there cannot be more than $p-1$ resource units before the first sample or after the last sample — or there would have been another sample.

To bound the error of estimator $\hat{Y}_c$ for any context c, we make an argument analogous to the derivation of Equation A.9. Recall from Equation A.4 that $\hat{Y}_c = pn_c$, where n_c is the number of samples in c. Then, we have:

$$\hat{Y}_c - (p-1) \leq Y_c \leq \hat{Y}_c + (p-1) \tag{A.10}$$

A.2.2 Accuracy of $\hat{Y}_c$

We would like to know how many samples within a context c are necessary to produce an accurate estimator $\hat{Y}_c$ for the true value Y_c. Equation A.10 implies that given $\hat{Y}_c$ and period p, Y_c is somewhere within $\hat{Y}_c \pm (p-1)$. Thus, we want to know when $\hat{Y}_c$ is large relative to $(p-1)$. To estimate the accuracy of $\hat{Y}_c$ in terms of samples, we express one side of the magnitude of $\hat{Y}_c$'s potential error as a percentage:

$$\frac{(p-1)}{\hat{Y}_c}(100\%) < \frac{p}{pn_c}(100\%) = \frac{100}{n_c}\% \tag{A.11}$$

Equation A.11 implies that given n_c samples within context c, estimator $\hat{Y}_c$ has an accuracy of $\pm\frac{100}{n_c}\%$. In other words, 20 samples within context c yields an error bound of $\pm 5\%$; similarly, 10 samples yields a bound of $\pm 10\%$.

We might conclude from the above that we should be worried if we do not have more than, say, 10 samples in any given context c. Indeed, if our goal is a specified accuracy for the context's estimator $\hat{Y}_c$, we would probably have reason for concern. However, for the purpose of performance analysis, as long as there are sufficient samples in *important* contexts, often we *do not care* if the number of samples in other contexts is low. To see this, observe that our main concern in performance analysis is to pinpoint bottlenecks. This means that it is only necessary to obtain reliable estimates for *important* contexts. Often, many contexts in an application are unimportant and there is no problem if they receive only a handful of samples. Another way to state this observation is that, in contrast with instrumentation, sampling naturally elides unimportant data. Moreover, our stress on top-down analysis naturally highlights the important contexts with very accurate metric values.

To finish our our analysis of the accuracy of $\hat{Y}_c$, we consider whether $\hat{Y}_c$ is an unbiased estimator for context c. Recall that when a sample point is generated by resource R and associated with instruction instance j, we assign p units of resource R to y_j. A method of estimation is unbiased if the average value of the estimate, taken over all possible samples of a given size n_c is exactly equal to the true population value [42, p. 22]. Assume $n_c \geq 1$. By extension of Equation A.5, context c has resource metric total $Y_c = i + (n_c - 1)p + j$. Figure A.1 illustrated how the sample period p divides the execution of context c into n_c regions. Assume that $Y_c = pn_c$, meaning that $i + j = p$. By systematic sampling, there are p possible samples of size n_c within c. Because for each sample we have $Y_c = pn_c = \hat{Y}_c$, clearly p is an unbiased estimator.

In general, however, $Y_c \neq pn_c$. By the pigeonhole principle, the first $n_c - 1$ sample points must fall into the first $n_c - 1$ regions of Figure A.1. The last region is of size $i + j$ and ranges from 1 to $2p - 1$ units. Therefore, depending on the location of the first sample, the last region could hold 0, 1 or 2 sample points. Consequently, the number of sample points varies from n_c by ± 1. By Equation A.10, the estimator $\hat{Y}_c$ of each possible sample is bounded by $Y_c \pm (p - 1)$. The average of $\hat{Y}_c$ over all p samples within c is

$$\frac{1}{p}\sum_{i=1}^{p}\hat{Y}_{c,i} = \sum_{i=1}^{p}\frac{pn_{c,i}}{p} = \sum_{i=1}^{p} n_{c,i} \approx Y_c$$

Since some samples are underestimates and some are overestimates, the approximation is very close in practice and we can consider p to be an unbiased estimator.

When using a PMU with a high skid factor, the analysis becomes more complicated. For example, because of skid, a sample in small procedure (context) could be attributed to either a callee or caller of that procedure. However, we noted earlier that the effects of skid are greatly diminished for code that appears within a long-running loop.

To more fully account for skid, we could perform the following two-part process. First, we obtain a distribution of a PMU's skid, possibly by using microbenchmarks. Then, using this distribution, we could describe program characteristics that allow us to make precise statements about accuracy. For instance, if the total instruction instances of a context c are large relative to the expected value of the PMU's skid, then the results of the above analysis should apply.

A.2.3 Choosing sampling periods

Finally, we consider the question of choosing good sampling periods. For most programs, a sampling frequency of hundreds to thousands of samples/second yields high accuracy and low overheads. In extreme cases, such as for very long- or short-running applications, it may be desirable to customize the sampling frequency. With time-based events, one can easily derive a sampling period by estimating program run time and the desired total number of sample points.

A powerful use of a PMU is to determine an application's rate-limiting resource by sampling on events that are not related to time. The simple approach above is insufficient for non-time events. To compute good periods for event-based sampling of a non-time event, we modify the approach just outlined. There are three steps.

Assume we wish to sample on an event that monitors a specific resource. The first step is to determine both a saturation request rate and maximum request rate for that resource. The saturation request rate is the request rate that creates contention for usage of that resource. The maximum request rate is the maximum rate that the resource can be requested by a single program thread. (Typically, the saturation request rate is less than maximum request rate, though this is not necessary.) This information can be computed with knowledge of a platform's architecture and ABI. For example, consider an architecture where L3 misses access main memory. Given information on the bandwidth between L3 and main memory and L3 line size, one can estimate the L3 cache miss rate that saturates the memory bus. A result of this analysis might be that r_{sat} L3 miss events per cycle results in memory bus saturation. To determine the maximum request rate for the resource, one can use information such as maximum number of operands per instruction, number of hardware contexts, and the issue width for each context. The result is a maximum request rate of r_{max} events per cycle.

The next step is to obtain an initial sampling period by converting the saturation request rate into a sampling period using a target sampling frequency. Suppose we wish to sample at 1000 samples/second on a processor core running at 1 GHz. This translates into a target frequency of 1 sample for every 1M cycles. To convert the saturation rate of r_{sat} events/cycle to a sampling period, we scale the rate by 1M cycles to obtain a period of 1M $\times$ r_{sat} events. Thus, a program execution that uses the given resource exactly at the saturation threshold generates sampling signals at the target frequency of 1000 samples/second. On the other hand, an execution that consumes the resource far below the saturation point generates samples at a much lower frequency, which is not a problem.

The final step is only relevant if applications typically exceed the saturation request rate by large amounts. For instance, suppose we have computed a period for an L3-cache-miss event, where L3 misses per second multiplied by L3 line size is directly related to memory bandwidth. When the hardware's memory bandwidth is exceeded, the application will generate L3 misses at a rate between the saturation and maximum request rates. When this happens, the period derived from the saturation request rate may result in sampling frequencies that are much higher than the target frequency. Such excessive sampling frequencies are undesirable because we do not want a performance tool to significantly contribute to overhead even if an application contains a severe bottleneck. Clearly, the relative magnitudes of the maximum and saturation request rates indicates the degree to which this could be an issue. To resolve this problem, experimentation is needed to choose a sampling period such that resource saturation is reliably detected without an excessively high sampling frequency.

Once good periods are chosen, it is easy to analyze an application's performance with respect to the resource in question. To do this, we sample both the resource event and a time-relative metric such as processor cycles. Then, we create a derived metric that converts events back into resource usage rates. If relatively few samples occur in any given context, the usage rate will be low and we can safely conclude the resource is not a rate limiter. Conversely, if the saturation rate is frequently exceeded, that resource contributes to a program bottleneck.

Appendix B

Efficiently Representing Logical CCTs

This appendix complements Chapter 4 by discussing the details of how to efficiently represent logical calling context trees.

Recall that Section 4.3.2 defined a logical calling context tree (L-CCT) as a tree of bichords. Accordingly, two distinct call paths in the tree may be partially shared if and only if they they share a common prefix of bichords. (All paths share a common root.) One issue that arises during a straight-forward implementation of L-CCTs is that common notes between multiple bichords are unnecessarily duplicated. We illustrate this problem with an example.

Suppose over the course of several samples, we obtain several logical unwinds of the forms below (where inner frames are on the left and a sample point, if relevant, is underlined):

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.1}$$

$$\langle (\underline{p'_{i,\mathsf{b}}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.2}$$

$$\langle (\underline{p'_{i,\mathsf{c}}}, p_{i,\mathsf{b}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.3}$$

$$\ldots, \langle (p_{i,\mathsf{c}}, p_{i,\mathsf{b}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.4}$$

$$\langle (\underline{p'_{i,\mathsf{a}}}), (l_{i,1}) \rangle, \ldots \tag{B.5}$$

$$\ldots, \langle (p_{i,\mathsf{e}}, p_{i,\mathsf{f}}, p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.6}$$

$$\ldots, \langle (p_{i,\mathsf{c}}), (l_{i,1}) \rangle, \langle (p_{i,\mathsf{b}}), (l_{i,1}) \rangle, \langle (p_{i,\mathsf{a}}), (l_{i,1}) \rangle, \ldots \tag{B.7}$$

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{j,1}) \rangle, \ldots \tag{B.8}$$

$$\ldots \langle (p_{i,\mathsf{a}}), (l_{i,2}, l_{i,1}) \rangle, \ldots \tag{B.9}$$

Unwinds (B.1)–(B.6), with bichords of association $\mathbf{M} \leftrightarrow 1$ and $1 \leftrightarrow 1$, could represent an interpreter implementing a high-level logical operation, signified by l-note $l_{i,1}$. Although none of these bichords are equal, all share $l_{i,1}$; and all but (B.5) share $p_{i,\mathsf{a}}$. However, a L-CCT treats each bichord as an atomic unit, thereby requiring that any common notes be duplicated when the corresponding call paths are inserted into the L-CCT. (Even the bichords in Unwinds (B.3) and (B.4) must be distinct because the former contains a sample and should therefore be a leaf node.) In general, if the M-portion of these bichords is long, samples occur in most of the unique prefixes. An analogous situation occurs in our Cilk profiler, where the root bichord of (almost) all call paths has association $1 \leftrightarrow \mathbf{M}$. As a result, several seemingly unnecessary p-notes exist with the L-CCT. For compact representation of an L-CCT, it is desirable to know when it is both possible and profitable to share the notes of two bichords.

B.1 Terminology

Observe that some associations are naturally related. For example, $1 \leftrightarrow 0$ is the natural 'base case' of $\mathbf{M} \leftrightarrow 0$. Similarly, $1 \leftrightarrow 1$ is the natural 'base case' of both $1 \leftrightarrow \mathbf{M}$ and $\mathbf{M} \leftrightarrow 1$. We therefore define three *association classes*, which group related associations:

- $\mathcal{A} \leftrightarrow 0 = \{1 \leftrightarrow 0, \mathbf{M} \leftrightarrow 0\}$
- $\mathcal{A} \leftrightarrow 1 = \{1 \leftrightarrow 1, \mathbf{M} \leftrightarrow 1\}$
- $1 \leftrightarrow \mathcal{A} = \{1 \leftrightarrow 1, 1 \leftrightarrow \mathbf{M}\}$

In this notation, '$\mathcal{A}$' acts like a variable that can take a different value for each member in a set of associations.

Let the functions ip and lip return the physical and logical instruction pointers given a p-note or l-note, respectively. The functions assoc and assoc-class return the association and association-class of a bichord, respectively. For convenience, we also define assoc-class= to test whether two bichords have identical association classes, respectively.

B.2 Sharing Within Bichords

We first consider the limits of sharing within bichords. Sharing between any two bichords may either be full or partial. If two paths partially share a bichord, they may still be able to partially share another bichord (cf. Unwinds (B.4) and (B.7)). However, partially sharing either bichord requires that the paths diverge in some fashion (otherwise they would be equal). Additional sharing requires that paths merge again, turning the tree into a graph and creating ambiguous calling contexts. Therefore, two bichords may be partially shared only if they are both roots of their respective call paths or their respective call path predecessors are fully shared. After partial sharing, paths must diverge.

The next task is to clearly define when partial sharing may occur between two bichords $B_x = \langle P_x, L_x \rangle$ and $B_y = \langle P_y, L_y \rangle$. We divide the analysis into two cases.

Case 1. $P_x = P_y$ or $L_x = L_y$. Without loss of generality assume the latter.

- assoc-class=(B_x, B_y): Compare Unwinds (B.1)–(B.6). Although these bichords represent at least three fully distinct contexts and two different associations, they have identical association classes. Each p-chord (except (B.5)) has a common prefix beginning with p-note $p_{i,\mathsf{a}}$. In general, several other types of non-prefix sharing are possible (e.g., suffixes). However, prefix sharing naturally corresponds to tree structure whereas non-prefix sharing effectively requires that a path diverges, skips one or more p-notes, and then re-merges.

Therefore we formulate the prefix condition for partially sharing two bichords B_x and B_y:

- $((P_x \sqsubset P_y) \vee (P_y \sqsubset P_x))$ and $L_x = L_y$
- $P_x = P_y$ and $((L_x \sqsubset L_y) \vee (L_y \sqsubset L_x))$ (by symmetry)

where $=$ and $\sqsubset$ ('strict prefix') are defined with respect to the sequence of notes that form a chord.

The one issue is that B_x and B_y may have different associations; prefix sharing is not effective if associations must be duplicated. However, because we know the bichord's association classes are identical, we know that if their associations are different, one association must be the 'base case' of the other. For example, Unwinds (B.1) and (B.2) have associations $1 \leftrightarrow 1$ and $\mathbf{M} \leftrightarrow 1$, respectively. We show below how to implement an implicit 'base-case flag' that preserves this information.

It turns out that the prefix condition can be relaxed slightly. Consider Unwinds (B.2) and (B.3), which may share p-note $p_{i,\mathsf{a}}$ by the above condition. Observe that $p'_{i,\mathsf{b}}$ represents a sample point while $p_{i,\mathsf{b}}$ represents a call site. Although in general $\mathsf{ip}(p'_{i,\mathsf{b}}) \neq \mathsf{ip}(p_{i,\mathsf{b}})$, a sample can be taken at a call site (technically, a return address), meaning that it is possible that $\mathsf{ip}(p'_{i,\mathsf{b}}) = \mathsf{ip}(p_{i,\mathsf{b}})$. We show below how to implement an implicit 'sample-point flag' that enables us to extend the prefix condition to allow sharing in this case. The flag indicates that the note both is and is not a sample point.

- assoc-class$\neq$(B_x, B_y): An enumeration of the possibilities for B_y for each of the five possible associations for B_x shows that this case is impossible (by the assumption $L_x = L_y$).

Case 2. $P_x \neq P_y$ and $L_x \neq L_y$.

- assoc-class=(B_x, B_y): Note that neither association may be in association class $\mathcal{A} \leftrightarrow 0$; otherwise $L_x = L_y$.

  We now consider the two other association classes and focus, without loss of generality, on $\mathcal{A} \leftrightarrow 1$. There are three cases. First, both bichords may have association $1 \leftrightarrow 1$. Second, one bichord has association $1 \leftrightarrow 1$ and the other $\mathbf{M} \leftrightarrow 1$. Third, both bichords have association $\mathbf{M} \leftrightarrow 1$.

  In the first case, no sharing is possible (since neither chord is equal). In the second and third cases, prefix sharing among p-notes may be possible. However, l-notes must be duplicated to maintain distinct logical calling contexts (cf. Unwinds (B.2) and (B.8)). Therefore, partial sharing is not profitable.

- assoc-class$\neq$(B_x, B_y): Since association classes are fully distinct, partial sharing is not possible without duplicating association information (cf. Unwinds (B.2) and (B.9)).

B.3 Implementation

We now translate the above conclusions into a practical implementation for the L-CCT.

We maintain the two-level distinction between bichords and notes implicitly. A bichord is represented by a list of `Node`-structures. Each `Node` contains an association (assoc) and a physical and logical instruction pointer (ip and lip, respectively). Given a bichord $\langle P_x, L_x \rangle$, we need n `Node`s $X_1, \ldots, X_n$ where $n = \mathsf{max}(|P_x|, |L_x|)$ and where X_1 represents the outermost portion of the bichord. Let the function note-id return the index of a `Node`-structure within a bichord: $\mathsf{note\text{-}id}(X_j) = j$.[1] Note that $\mathsf{ip}(X_j) = \mathsf{NIL}$ if $|P_x| < j \leq n$; similarly for $\mathsf{lip}(X_k)$.

Given this representation, a logical call path is simply a list of `Node`-structures $X_1, \ldots, X_n$. A bichord begins at every X_i where $\mathsf{note\text{-}id}(X_i) = 1$. A L-CCT is a tree of `Node`-structures. Each `Node` in the L-CCT may have a vector of metric values. A non-zero metric count naturally implements the 'sample-point flag' mentioned above. To implement the 'base-case flag', we simply ensure that when a $1 \leftrightarrow 1$ bichord shares the root of, say, an $\mathbf{M} \leftrightarrow 1$ bichord, the root `Node` has association $1 \leftrightarrow 1$. Thus, the bichords in Unwinds (B.1) and (B.2) would be represented as two `Node`s $\ldots X_1, X_2 \ldots$ where $\mathsf{assoc}(X_1) = 1 \leftrightarrow 1$, $\mathsf{assoc}(X_2) = \mathbf{M} \leftrightarrow 1$; where X_2 has a non-zero metric value; and where X_1 is an interior node.

The final item is to describe an efficient way to insert a logical call path into the L-CCT in a way that corresponds to the full and partial sharing of bichords described above. To ensure the L-CCT is rooted, we prefix a synthetic root node to the beginning of every call path, implying that every call path has a length of at least two. Inserting a path into the L-CCT therefore turns into the following problem: Given the call path fragment $f' \rightarrow g'$ (as `Node`-structures) and given a node f in the L-CCT such that $f' = f$, is it the case that $\exists g$ such that g is a child of f and $\mathsf{sharable?}(g, g')$ holds? If the answer is yes, g may be shared and insertion proceeds to the children of g and g'. Otherwise, a new path for g is spliced into the tree.

To define sharable?, we first consider a physical calling context tree where `Node`-structures only contain a physical instruction pointer (ip). In this case we simply

[1]In implementation, assoc and note-id may be combined into one bit-field, since the former only needs 3 bits; we use 8 and pre-compute association classes.

have:

$$\mathsf{sharable?}(f, f') : \mathsf{ip}{=}(f, f')$$

To extend this definition to a L-CCT, we observe that both ips and lips should be equal if bichords are equal or if one is a prefix of the other. To properly compute a prefix, bichords must be demarcated and aligned which we can ensure by also testing note-id(). Consulting note-id() also forces path divergence after partial sharing. Finally, we need to ensure that sharing is only permitted when at least one of cases from above, $P_x = P_y$ and $L_x = L_y$, holds. We can check this by additionally examining assoc-class. This results in the following simple test:

$$\begin{aligned} \mathsf{sharable?}(f, f') : \mathsf{ip}{=}(f, f') \wedge \mathsf{lip}{=}(f, f') \wedge \\ \mathsf{assoc\text{-}class}{=}(f, f') \wedge \mathsf{note\text{-}id}{=}(f, f') \end{aligned}$$

Appendix C

Definitions of Atomic Primitives

The swap primitive takes a memory location m and a new value newval for m. It atomically performs the following operation, written as C pseudo-code:

```
type swap(void* m, type newval)
{
  type myold = *m;
  *m = newval;
  return myold;
}
```

The CAS (compare-and-swap) primitive takes a memory location m and an old and new value for m, oldval and newval, respectively. It atomically performs the following operation, written as C pseudo-code:

```
type CAS(void* m, type oldval, type newval)
{
  type myold = *m;
  if (myold == oldval) *m = newval;
  return myold;
}
```

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] L. Adhianto, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Performance measurement and analysis for supercomputers with node-level parallelism. In *Workshop on Node Level Parallelism for Large Scale Supercomputers, in conjuction with ACM/IEEE Conference on Supercomputing*, November 2008.

[3] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proc. of the 1995 ACM/IEEE Conference on Supercomputing*, page 50, New York, NY, USA, 1995. ACM.

[4] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of IBM BlueGene/P. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[5] S. R. Alam, R. F. Barrett, M. R. Fahey, J. A. Kuehn, O. B. Messer, R. T. Mills, P. C. Roth, J. S. Vetter, and P. H. Worley. An evaluation of the Oak Ridge National Laboratory Cray XT3. *International Journal of High Performance Computing Applications*, 22(1):52–80, 2008.

[6] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley. Cray XT4: An early evaluation for petascale scientific simulation. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[7] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization made easier with Intel Performance-Tuning Utility. *Intel Technology Journal*, 11(4):275–286, November 2007.

[8] AMD. CodeAnalyst performance analyzer. `http://developer.amd.com/cpu/codeanalyst`, January 2010.

[9] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[10] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[11] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 1(1):6–16, 1990.

[12] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[13] Apple Computer. Shark. `http://developer.apple.com/tools/sharkoptimize.html`, November 2004.

[14] J. H. Applegate, M. R. Douglas, Y. Gürsel, P. Hunter, C. L. Seitz, and G. J. Sussman. Detecting application load imbalance on high end massively parallel systems. *Lecture Notes in Physics*, 267/1986:86–95, 1986.

[15] Argonne Leadership Computing Facility. INCITE Getting Started Workshop. `http://workshops.alcf.anl.gov/gs10/agenda/`, January 2010.

[16] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. *Proc. of the 8th International Symposium on High Performance Distributed Computing*, pages 115–124, 1999.

[17] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, New York, NY, USA, 2001. ACM.

[18] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proc. of the 19th International Conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[19] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[20] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. *Lecture Notes in Computer Science*, 3769/2005:465–476, 2005.

[21] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Language Systems*, 16(4):1319–1360, 1994.

[22] C. Bernard, T. Burch, C. DeTar, S. Gottlieb, E. Gregory, U. Heller, J. Osborn, R. Sugar, and D. Toussaint. QCD thermodynamics with three flavors of improved staggered quarks. *Phys. Rev.*, D71:034504, 2005.

[23] W. Binder. Portable and accurate sampling profiling for Java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[24] J. D. Bratt. *Abraham Kuyper: A Centennial Reader.* Eerdmans, Grand Rapids, 1998.

[25] D. Breazeal. A new direction for PGI performance profiling. *PGI Insider*, August 2009. `http://www.pgroup.com/lit/articles/insider/v1n2a2.htm`.

[26] C. P. Breshears. Using Intel Thread Profiler for Win32 threads: Philosophy and theory. `http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory`, August 2007.

[27] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *Proc. of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 1992. ACM.

[28] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[29] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[30] Bull SAS. Bullx Cluster Suite Product Specifications. `http://www.bull.com/hpc/download/S-bullxCSUITE-en1.pdf`, July 2009.

[31] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. PerfExpert: An automated HPC performance measurement and analysis tool with optimization recommendations. *(Under review)*, January 2010.

[32] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[33] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004. USENIX Association.

[34] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proc. of the Intl. Workshop on OpenMP Appl. and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.

[35] Center for Scalable Application Development Software. Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies. `http://cscads.rice.edu/workshops/summer09/leadership-computing`, July 2009.

[36] S. Cepeda. Performance analysis and Intel Parallel Amplifier. `http://www.ddj.com/architect/217700473`, May 27, 2009.

[37] K. Chand, B. Fix, T. Dahlgren, L. F. Diachin, X. Li, C. OllivierGooch, E. S. Seol, M. S. Shephard, T. Tautges, and H. Trease. The ITAPS iMesh interface. `http://www.tstt-scidac.org/software/documentation/iMesh_userguide.pdf`, June 2007.

[38] M. Charney. XED2 user guide. `http://www.pintool.org/docs/24110/Xed/html`, January 2009.

[39] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu. MPI performance analysis tools on Blue Gene/L. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, page 123, New York, NY, USA, 2006. ACM.

[40] M. Chung. Monitoring and managing Java SE 6 platform applications. `http://java.sun.com/developer/technicalArticles/J2SE/monitoring`, August 2006.

[41] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *Proc. of the 21st International Conference on Supercomputing*, pages 13–22, New York, NY, USA, 2007. ACM.

[42] W. G. Cochran. *Sampling Techniques.* John Wiley & Sons, Inc., New York, third edition edition, 1977.

[43] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 80–89, Washington, DC, USA, 1996. IEEE Computer Society.

[44] D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin 2000 and Onyx2 performance tuning and optimization guide. Technical Report 007-3430-003, Silicon Graphics, Inc., 2001.

[45] M. E. Crovella and T. J. LeBlanc. Parallel performance using lost cycles analysis. In *Proc. of the 1994 ACM/IEEE Conference on Supercomputing*, pages 600–609, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[46] DARPA High Productivity Computing Program. Scalable Synthetic Compact Application benchmarks. `http://www.highproductivity.org/SSCABmks.htm`, 2007.

[47] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[48] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[49] L. DeRose, J. Ted Hoover, and J. K. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. *Proc. of the International Parallel and Distributed Processing Symposium*, April 2001.

[50] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the International Symposium on Code Generation and Optimization*, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[51] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2009. ACM.

[52] A. Dubey, L. B. Reid, and R. Fisher. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta*, 132:014046, 2008.

[53] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. *SIGARCH Comput. Archit. News*, 28(5):202–211, 2000.

[54] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proc. of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 26–30, New York, NY, USA, 2008. ACM.

[55] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. *SIGPLAN Not.*, 41(11):175–184, 2006.

[56] R. Fowler, L. Adhianto, B. de Supinski, M. Fagan, T. Gamblin, M. Krentel, J. Mellor-Crummey, M. Schulz, and N. Tallent. Frontiers of performance analysis on leadership class systems. *Journal of Physics: Conference Series*, 2009.

[57] Free Standards Group. DWARF debugging information format, version 3. `http://dwarf.freestandards.org`, December 2005.

[58] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[59] W. Frings, F. Wolf, and V. Petkov. Scalable massively parallel i/o to task-local files. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2009. ACM.

[60] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM.

[61] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *Proc. of the GCC Developers' Summit, 2006*, pages 21–36, 2006.

[62] K. Fürlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *Proc. of the First and Second International Workshops on OpenMP*, pages 15–23, Eugene, Oregon, USA, May 2005. LNCS 4315.

[63] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[64] T. Gamblin, R. Fowler, and D. A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *Proc. of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2008.

[65] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. of the 1982 ACM SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[66] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, August 1983.

[67] R. J. Hall. Call path profiling. In *Proc. of the 14th International Conference on Software Engineering*, pages 296–306, New York, NY, USA, 1992. ACM.

[68] G. J. Hansen, C. A. Linthicum, and G. Brooks. Experience with a performance analyzer for multithreaded applications. In *Proc. of the 1990 ACM/IEEE Conference on Supercomputing*, pages 124–131, Washington, DC, USA, 1990. IEEE Computer Society.

[69] R. J. Harrison, G. I. Fann, T. Yanai, and G. Beylkin. Multiresolution quantum chemistry in multiwavelet bases. *Lecture Notes in Computer Science*, 2660/2003:103–110, 2003.

[70] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[71] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001.

[72] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A performance comparison through benchmarking and modeling of three leading supercomputers: Blue Gene/L, Red Storm, and Purple. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, page 74, New York, NY, USA, 2006. ACM.

[73] IBM. IBM lock analyzer for Java. `http://www.alphaworks.ibm.com/tech/jla`, September 2007.

[74] IBM. IBM system Blue Gene solution: Performance analysis tools. Redpaper REDP-4256-01, November 2008.

[75] Intel Corporation. Intel Performance Tuning Utility. http://software.intel.com/en-us/articles/intel-performance-tuning-utility, April 2009.

[76] Intel Corporation. Intel Thread Profiler. http://www.intel.com/software/products/tpwin, January 2010.

[77] Intel Corporation. Intel VTune performance analyzer. http://www.intel.com/software/products/vtune, January 2010.

[78] M. Itzkowitz and Y. Maruyama. HPC Profiling with the Sun Studio Performance Tools. http://developers.sun.com/sunstudio/documentation/techart/hpc_profiling.pdf, October 2009.

[79] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. http://www.compunity.org/futures/omp-api.html, October 2007.

[80] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, March 2001.

[81] R. Kufrin. PerfSuite: An accessible, open source performance analysis environment for Linux. In *Proc. of the 6th International Conference on Linux Clusters*, April 2005.

[82] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008.

[83] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: Towards debugging millions

of cores. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[84] D. Levinthal. Cycle accounting analysis on Intel Core2 processors. `http://assets.devx.com/goparallel/18027.pdf`.

[85] J. Levon *et al.* OProfile. `http://oprofile.sourceforge.net`, November 2009.

[86] P. Lichtner *et al.* PFLOTRAN project web site. `https://software.lanl.gov/pflotran`, 2009.

[87] J. C. Linford, M.-A. Hermanns, M. Geimer, D. Boehme, and F. Wolf. Detecting load imbalance in massively parallel applications. Technical Report FZJ-JSC-IB-2008-09, Forschungszentrum Jülich, December 2008.

[88] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[89] P. MacNeice, K. M. Olson, C. Mobarry, R. deFainchtein, and C. Packer. PARAMESH : A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126:330–354, 2000.

[90] A. D. Malony, S. Shende, A. Morris, S. Biersdorff, W. Spear, K. Huck, and A. Nataraj. Evolution of a parallel performance system. In *Tools for High Performance Computing*, pages 169–190. Springer Berlin Heidelberg, 2008.

[91] M. E. Maxwell, P. J. Teller, and L. Salay. Accuracy of performance monitoring hardware. In *Proc. of LACSI Symposium*, 2002.

[92] P. E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1999.

[93] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.

[94] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[95] J. Mellor-Crummey and N. R. Tallent. A methodology for accurate, effective and scalable performance analysis of application programs. In *Workshop on Tools, Infrastructures and Methodologies for the Evaluation of Research Systems, in conjuction with the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 4–11, February 2008.

[96] J. Mellor-Crummey, N. R. Tallent, M. Fagan, and J. Odegard. Application performance profiling on the Cray XD1 using HPCToolkit. In *Proc. of the Cray User's Group*, May 2007.

[97] C. L. Mendes and D. A. Reed. Monitoring Large Systems Via Statistical Sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, 2004.

[98] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1999. `http://www.mpi-forum.org/docs/mpi-11.ps`.

[99] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[100] R. T. Mills, C. Lu, P. C. Lichtner, and G. E. Hammond. Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN. *Journal of Physics Conference Series*, 78(012051), 2007.

[101] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proc. of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.

[102] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proc. of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.

[103] M. Monchiero, R. Canal, and A. Gonzalez. Power/performance/thermal design-space exploration for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):666–681, May 2008.

[104] D. Monroe. ENERGY Science with DIGITAL Combustors. `http://www.scidacreview.org/0602/html/combustion.html`, 2006.

[105] D. Mosberger-Tang. libunwind. `http://www.nongnu.org/libunwind`, May 2009.

[106] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proc. of the 4th International Conference on Computing Frontiers*, pages 143–152, New York, NY, USA, 2007. ACM.

[107] P. J. Mucci. PapiEx: Execute arbitrary application and measure hardware performance counters with PAPI. `http://icl.cs.utk.edu/~mucci/papiex`, July 2008.

[108] P. J. Mucci and T. Mohan. An open source performance tools software suite for scientific computing. *Concurrency and Computation: Practice and Experience*, 22(2):206–216, 2009.

[109] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[110] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture (Version 1.1). `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[111] L. Oliker, A. Canning, J. Carter, C. Iancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific application performance on candidate petascale platforms. *International Parallel and Distributed Processing Symposium*, 0:69, 2007.

[112] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. *Lecture Notes in Computer Science*, 4382/2007:235–250, 2007.

[113] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[114] PETSc Team. KSPIBCGS solver. In *PETSc: Portable, Extensible Toolkit for Scientific Computation.* `http://www.mcs.anl.gov/petsc/petsc-as/`

snapshots/petsc-current/docs/manualpages/KSP/KSPIBCGS.html, December 2008.

[115] G. F. Pfister and V. A. Norton. Hot-spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[116] K. B. Pierce. Forward walking through binary code to determine offsets for stack walking. United States Patent 7,178,132 B2, February 2007.

[117] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proc. of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[118] J. Reinders. *Intel Threading Building Blocks.* O'Reilly, Sebastopol, CA, 2007.

[119] Rice University. HPCToolkit performance tools. http://hpctoolkit.org.

[120] Rice University. Rice HPC Summer Institute. http://k2i.rice.edu/events/HPC2009Institute, May 2009.

[121] Rice University. PACE: Platform-Aware Compilation Environment. http://dev.pace.rice.edu, January 2010.

[122] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *Proc. of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)*, pages 798–804, 2008.

[123] S. Sandmann. Sysprof. http://www.daimi.au.dk/~sandmann/sysprof, January 2010.

[124] S. S. Sastry, R. Bodík, and J. E. Smith. Rapid profiling via stratified sampling. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, pages 278–289, New York, NY, USA, 2001. ACM.

[125] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[126] M. Schulz and B. R. de Supinski. P^{N}MPI tools: A whole lot greater than the sum of their parts. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[127] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.*, 16(2-3):105–121, 2008.

[128] S. Shende, A. Malony, and A. Morris. *Optimization of Instrumentation in Parallel Performance Evaluation Tools*, volume 4699 of *LNCS*, pages 440–449. Springer, 2008.

[129] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[130] Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. Technical Report 007-3311-011, SGI, 2003.

[131] D. Skinner *et al.* IPM: Integrated performance monitoring. `http://ipm-hpc.sourceforge.net/`, Octobert 2009.

[132] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[133] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. of the 2004 International Conference on Parallel Processing*, pages 63–72, Washington, DC, USA, 2004. IEEE Computer Society.

[134] SPEC Corporation. SPEC CPU2006 benchmark suite. `http://www.spec.org/cpu2006`, August 2008.

[135] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[136] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`, June 2007.

[137] H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. B. III, and A. D. George. GASP! A standardized performance analysis tool interface for global address space programming models. Technical Report LBNL-61659, Lawrence Berkeley National Laboratory, 2006.

[138] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPC-Toolkit: Performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088 (5pp), 2008.

[139] N. R. Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. M.S. thesis, Department of Computer Science, Rice University, May 2007.

[140] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[141] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[142] N. R. Tallent and J. M. Mellor-Crummey. Identifying performance bottlenecks in work-stealing computations. *Computer*, 42(12):44–50, 2009.

[143] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2009. ACM.

[144] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–280, New York, NY, USA, 2010. ACM.

[145] T. J. Tautges. MOAB-SD: Integrated structured and unstructured mesh representation. *Eng. Comput. (Lond.)*, 20(3):286–293, 2004.

[146] Texas Advanced Computing Center. TACC Summer Supercomputing Institute (2009). `http://www.tacc.utexas.edu/summerinstitute/`, August 2009.

[147] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 1999.

[148] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 240–250, New York, NY, USA, 2002. ACM.

[149] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.

[150] O. Waddell and J. M. Ashley. Visualizing the performance of higher-order programs. In *Proc. of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82. ACM, 1998.

[151] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing*, pages 157–167. Springer Berlin Heidelberg, 2008.

[152] N. J. Wright, W. Pfeiffer, and A. Snavely. Characterizing parallel scaling of scientific applications using IPM. In *Proc. of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[153] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of the 2000 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.

[154] B. J. N. Wylie, M. Geimer, and F. Wolf. Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Sci. Program.*, 16(2-3):167–181, 2008.

[155] L. Yang and R. Brent. The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures. In *Proc. of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 324–328, 2002.

[156] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for Java just-in-time compilers. *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 0:148, 2003.

[157] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS r10000 performance counters. In *Proc. of the 1996 ACM/IEEE Conference on Supercomputing*, page 16. ACM, 1996.

[158] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–271, New York, NY, USA, 2006. ACM.