A Methodology for Accurate, Effective and Scalable Performance Analysis of Application Programs

John Mellor-Crummey
Rice University
johnmc@cs.rice.edu

Nathan Tallent
Rice University
tallent@cs.rice.edu

***Abstract*—We describe a unique and comprehensive methodology for accurately measuring and effectively analyzing the performance of an application's execution. This methodology is 1) *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation); 2) *effective*, because it associates useful performance metrics (such as memory bandwidth) with important source code abstractions (such as loops); and 3) *scalable*, because it can be used to effectively analyze the performance of a single thread or a large parallel code. Our methodology has been implemented in HPCTOOLKIT, an open-source suite of multi-platform tools. One of HPCTOOLKIT's unique capabilities is collecting call path profiles of fully-optimized SPMD parallel programs and pinpointing scalability bottlenecks.**

I. INTRODUCTION

Modern programs frequently employ sophisticated modular designs that exploit object oriented abstractions and generics. These codes often contain deep loop nests spread across multiple routines. To deliver high performance for such codes, compilers inline routines and optimize loops. However, because the increasing architectural complexity of chips has made obtaining a significant percentage of peak performance extremely difficult, optimizing compilers usually deliver only a fraction of the possible improvement. Since only carefully tuned applications achieve a substantial fraction of peak performance, much of the burden of tuning falls to application developers. Consequently, good performance tools are essential for helping developers determine where they should invest their tuning effort.

We describe a unique and comprehensive methodology for analyzing the performance of an application's execution. This methodology is

- *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation);
- *effective*, because it associates useful performance metrics (such as memory bandwidth) with important source code abstractions (such as loops);
- and *scalable*, because it can be used to effectively analyze the performance of a single thread or a large parallel code.

Our methodology is derived from a set of mutually complementary principles that, while not novel in themselves, form a unique and coherent synthesis that is greater than the constituent parts. Moreover, we have developed several novel measurement and analysis techniques. This work has been implemented in HPCTOOLKIT [12], an open-source suite of multi-platform tools.

The rest of the paper is as follows. Section II enumerates the fundamental principles that form the basis of our methodology. Section III explains how these principles inform HPCTOOLKIT's performance measurement and, in particular, sampling-based call path profiling. Section IV shows how HPCTOOLKIT applies these principles to enable effective analysis of execution measurements. The case studies in Section V apply our tools and techniques actual codes. Section VI summarizes our contributions and discusses future directions.

II. A PERFORMANCE ANALYSIS METHODOLOGY

The following enumerates the fundamental principles that form the basis of our methodology and summarizes their design impact on HPCTOOLKIT.

A. Performance Analysis Principles

1) Language independence: Modern scientific programs often have a numerical core written in some modern dialect of Fortran, while using a combination of 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 manipulating source code written in different languages.

2) Avoid code instrumentation: Manual instrumentation is unacceptable for large applications. In addition to the effort it involves, adding instrumentation manually requires users to make *a priori* assumptions about where performance bottlenecks might be before they have any information.

Even using automatic tools to add source-level instrumentation can be problematic. For instance, using the Tau performance analysis tools to add source-level instrumentation to the Chroma code [9] from the US Lattice Quantum Chromodynamics project [14] required seven hours of recompilation [13].

Binary instrumentation, such as that performed by Dyninst [7] or Pin [10], addresses the aforementioned problems; however, instrumentation-based measurement itself can be problematic. Adding instrumentation to every procedure can substantially dilate a program's execution time. Experiments with `gprof` [6], a well-known call graph profiler, and the SPEC integer benchmarks showed that on average `gprof`

dilates execution time by 82% [5]. Adding instrumentation to loops presents even a greater risk of increasing overhead. Unless compensation techniques are used, instrumentation can also magnify the cost of small routines.

3) Avoid blind spots: Production applications frequently link against fully optimized and even partially stripped binaries — such as math and communication libraries — for which source code is not available. To avoid systematic error, performance measurements must be collected for such libraries, again implying that source code instrumentation is insufficient. However, fully optimized binaries create challenges for call path profiling and hierarchical aggregation of performance measurements (see Sections III-F and IV-A). To deftly handle such binaries, HPCTOOLKIT performs several types of binary analysis.

4) Context is essential for understanding layered and object-oriented software: In modern, modular programs, it is important to attribute the costs incurred by each procedure to the different contexts in which the procedure is called. The cost incurred for calls to communication primitives (*e.g.*, MPI_Wait) or code that results from instantiating C++ templates for data structures can vary widely depending upon their calling context. Because there are often layered implementations within applications and libraries, it is insufficient either to insert instrumentation at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCTOOLKIT supports call path profiling to attribute costs to the full calling contexts in which they are incurred.

5) Any one performance measure produces a myopic view: Measuring time or only one species of system 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, measures such as cache miss count indicate problems only if both the *miss rate* is high and the latency of the misses is not hidden. HPCTOOLKIT supports collection, correlation and presentation of multiple metrics.

6) Derived performance metrics are essential for effective analysis: Typical metrics such as elapsed time are useful for identifying algorithms that do not scale to common problem sizes. However, effectively tuning a program with appropriate complexity bounds requires a measure of not where resources are consumed, but where they are consumed *inefficiently*. For this purpose, derived measures such as the differences between peak and actual performance are far more useful than raw data such as counts of floating point operations. For maximum effectiveness, a tool should compute user-defined derived metrics automatically so that they can be used as keys for ranking and sorting.

7) Performance analysis should be top down: It is unreasonable to require users to hunt through mountains of printouts or wade through multiple windows full of data to identify important problems. To make analysis of large programs tractable, performance tools should organize performance 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 user interface supports hierarchical presentation of performance data according to both static and dynamic contexts, along with ranking and sorting based on multiple metrics.

8) Hierarchical aggregation is important in the face of approximate attribution: In modern multi-issue microprocessor cores with multiple functional units, out-of-order execution, and non-blocking caches, the amount of instruction level parallelism is such that it is very difficult or expensive to associate particular events with specific instructions. On such systems, line level (or finer) information can be misleading. However, even in the presence of fine-grain attribution problems, aggregate information for loops or procedures can be very accurate. HPCTOOLKIT's hierarchical presentation of measurement data deftly addresses this issue; loop level information available with HPCTOOLKIT is particularly useful.

9) With instruction-level parallelism, aggregate properties are vital: Even if profiling instrumentation could provide perfect attribution of costs to executable instructions and compilers could provide perfect mapping from executable instructions to source code, a program's performance on machines with extensive instruction-level parallelism is less a function of the properties of individual source lines, and more a function of the data dependences and balance among the statements in larger program units such as loops or loop nests [2]. For example, the balance of floating point operations to memory references within one source line is irrelevant to performance as long as the innermost loop containing that statement has an appropriate balance between the two types of operations, a good instruction schedule that keeps the pipelines full, and memory operations that can be scheduled to hide most of the cache miss latency.

10) Measurement and analysis must be scalable: Scientific applications are increasingly executed on clusters with SMP nodes of multicore chips, creating several layers parallelism. It is critical that measurement techniques must be scalable to 10s and even 100s of thousands of threads.

B. Developing a Methodology

From these principles we have devised a general methodology consisting of the workflow depicted in Figure 1; this workflow is implemented in HPCTOOLKIT. The workflow is organized around four principal capabilities:

1) *measurement* of performance metrics while an application executes,
2) *analysis* of application binaries to recover program structure,
3) *correlation* of dynamic performance metrics with source code structure, and
4) *presentation* of performance metrics and associated source code.

First, one compiles and links one's application for a production run, using *full* optimization. Second, one launches an application with HPCTOOLKIT's measurement tool, which uses statistical sampling to collect a performance profile.

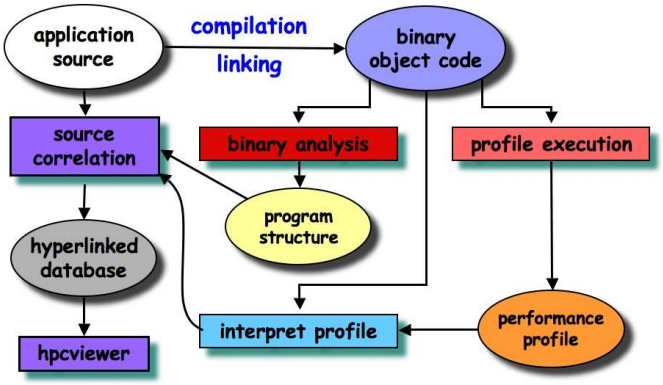

Fig. 1. Overview of HPCTOOLKIT workflow.

Third, one invokes HPCTOOLKIT's tool for analyzing the application binary to recover information about files, functions, loops, and inlined code.[1] Fourth, one combines information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with an interactive viewer.

At this level of detail, much of the workflow approximates other peformance analysis systems, with the most unusual step being binary analysis. However, the following sections will highlight how the methodological principles enumerated above suggest several novel approaches to both accurate measurement (Section III) and effective analysis (items 2-4 and Section IV).

III. ACCURATE MEASUREMENT

This section highlights the ways in which we have applied the methodological principles from Section II to measurement. Without accurate performance measurements for the fully optimized application, analysis is beside the point. 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 systematically dilating execution. In particular, HPCTOOLKIT avoids instrumentation and favors *statistical sampling*.

A. Statistical sampling

Statistical sampling uses a recurring event trigger to send signals to the program being profiled. When the event trigger occurs, a profiling signal is sent to the program. The signal handler then records the program counter (PC) and possibly the context. The recurring nature of the event trigger means that the program counter is sampled many times, resulting in a histogram of program counter/context. As long as the number of samples collected during execution is sufficiently large, their distribution is expected to approximate the true distribution of the costs that the event triggers are intended to measure.

B. Sampling triggers

Different kinds of event triggers lead to different measurements of program performance. Event triggers can be either asynchronous or synchronous. Asynchronous triggers are not initiated by direct program action, but may arise from interrupts triggered by the UNIX interval timer or hardware performance counter events. Hardware performance counter events 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 record bytes allocated, written, or communicated, respectively.

C. Measuring dynamically and statically linked executables

To enable measurement of unmodified, dynamically-linked, optimized application binaries, HPCTOOLKIT uses the library preloading feature of modern dynamic loaders. HPCTOOLKIT instructs the dynamic loader to preload a profiling library before launching an application using the `LD_PRELOAD` environment variable (or equivalent). For asynchronous triggers, the library's initialization routine allocates and initializes profiler state, configures the signal handlers and asynchronous event triggers (timers and/or hardware performance counters), and then initiates profiling. The library's finalization routine halts profiling and writes the profile state to disk for post-mortem analysis. Synchronous triggers do not need signal handlers or asynchronous event triggers; instead, dynamic preloading overrides the library routines of interest and logs information as appropriate when the routine is called in addition to performing the requested operation. Because static linking is mandatory on some lightweight operating systems such as Catamount and Compute Node Linux for the Cray XT, we have developed a linking script that mimics symbol overriding.

D. Flat profiling

HPCTOOLKIT supplies a lightweight profiler that simply collects program counter histograms without any information about calling context. This kind of simple profiling is referred to as *flat profiling*. Even such lightweight profiling can supply valuable information about a program's performance. Flat profiling suffices when a program's call graph is a tree.

E. Call path profiling

Although flat profiles are often effective, experience has shown that comprehensive performance analysis of modern modular software requires information about the context in which costs are incurred. One important kind of context for any sample event is the set of procedure frames active on the call stack at the time the event trigger fires. We refer to the state of the call stack as the *calling context* of the event, and

[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. Since compilers often provide line map information for fully-optimized code, this requirement need not require a special build process.

we refer to the process of augmenting simple PC histograms with calling context as *call path profiling*. The HPCTOOLKIT component that collects call path profiles is called `hpcrun`.

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 pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program counter location at which the event occurred. The rest of the list contains the return address for each of the active procedure frames. We retain stack pointers as well to distinguish between recursive invocations. We have not observed excessive space requirements when retaining entire call paths; if the storage of call paths were to become a concern, we could collapse calling contexts for recursive calls [1] or record only a suffix of full paths.

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) [1]. In a calling context tree, the path from the root of the tree to a node corresponds to a distinct call path observed during execution; a count at each node in the tree indicates the number of times that the path to that node was sampled.

F. 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 (space is reserved for the caller's registers and local variables; the frame is extended with calls to `alloca`; arguments to called procedures are pushed) and shrink (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 routine's machine instructions and computes how to unwind a stack frame for the procedure. 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 routine's stack frame. Each recipe describes (1) where to find the current frame's return address, (2) how to recover the value of the stack pointer for the caller's frame, and (3) how to recover the value that the base pointer register had in the caller's frame. Once 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 starts and ends. When working with applications, one often encounters partially-stripped binaries that are missing information about function boundaries. To address this problem, we developed a binary analyzer that identifies potential 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.

G. Maintaining control over applications

For `hpcrun` to maintain control over an application, certain calls to standard C library functions must be intercepted. For instance, `hpcrun` must be aware of when threads are created or destroyed, or when new dynamic libraries are loaded with `dlopen`. When such library calls occur, certain actions must be performed by `hpcrun`. To intercept such function calls in dynamically-linked executables, the profiler uses library preloading to interpose its own wrapped versions of library routines.

H. 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. As `hpcrun` only collects a sequence of one or more program counter values when a sample is taken, post-mortem analysis must map these instruction addresses to the functions that contained them. For this reason, the profiler identifies each sample recorded with the set of shared libraries loaded at the time. We call the list of shared objects loaded at a particular time an *epoch*; every sample collected is associated with a particular epoch. When a shared object is loaded at run time, a new epoch is constructed.

While the loading of shared objects requires the creation of new epochs, new epochs can also be created for other reasons. For instance, a program that is aware of the profiler's existence could ask the profiler to collect new epochs at phase changes during execution: an epoch associated with initialization, an epoch associated with each distinct computation phase, and so forth. This mechanism enables the performance analyst to divide an application profile into distinct phases and also provides a method of temporally ordering the collected samples at a coarse level, delivering some of the benefits of tracing, but without the space overhead.

I. Handling threads

When multiple threads are involved in a program, each thread maintains its own calling context tree. To initiate profiling for a thread, `hpcrun` intercepts thread creation and destruction to initialize and finalize profile state.

J. Evaluation

HPCTOOLKIT is unique in its ability to collect accurate call-path profiles of fully optimized code for minimal overhead, even with binary analysis to compute unwind information. Experiments show that `hpcrun` collects call path profiles for the SPEC CPU2006 benchmarks with an average overhead of less than 1% (200 samples/second) and an unwind success rate of 100% (*i.e.*, our binary analyzer computed an accurate unwind recipe for every sample event). These results hold over both the Intel 10.0 compiler and the PathScale 3.1 compiler on

a two-way 2 GHz Opteron (246) processor (1 MB L2) SMP system with 8 GB memory running Fedora Linux (Core 7). In contrast to HPCToolkit, the widely used VTune [8] employs binary instrumentation and requires a dynamically linked application for call path profiling. VTune's instrumentation has blind spots for functions in stripped executables or libraries and imposes enough overhead that Intel explicitly discourages program-wide call path profiling.

IV. Effective Performance Analysis

Given accurate measurements, performance metrics must be correlated to source code structure and presented in a compelling way. The following describes HPCTOOLKIT's approach.

A. Correlating Performance Metrics with Optimized Code

To enable effective analysis, measurements of fully optimized programs must be correlated with important source code abstractions. Since measurements are made with reference to the binary — what actually executes — for analysis, it is necessary to map measurements back to the program source. To perform this translation, *i.e.*, to associate sample-based performance measurements with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure.[2] HPCTOOLKIT's `hpcstruct` constructs this mapping using binary analysis; we call this process "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, `hpcstruct` combines line map information with interval analysis in a way that enables it to identify transformations to procedures such as inlining and account for transformations to loops [13].[3]

Several benefits naturally accrue from this approach. First, HPCTOOLKIT can expose the structure of and assign metrics to what is actually executed, *even if source code is unavailable*. For example, `hpcstruct`'s program structure naturally reveals transformations such as loop fusion and scalarization 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. `hpcrun`'s function discovery heuristics exposes distinct logical procedures within stripped binaries.

B. Computed Metrics

Identifying performance problems and opportunities for tuning may require synthetic performance metrics. To quickly tune an algorithm that is not effectively using hardware resources, one should compute a metric that reflects *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 and FLOP count divided by a target FLOPs-per-cycle value, and displaying this measure at loop and procedure level. Our experiences with using multiple computed metrics such as miss ratios, instruction balance, and "lost cycles" underscore the power of this approach.

One novel application of HPCTOOLKIT's call path profiles is to use them to pinpoint and quantify scalability bottlenecks in SPMD parallel programs [3]. Combining call path profiles with program structure information, HPCTOOLKIT can use this metric to pinpoint poorly scaling code in context.

C. Presentation and Visualization

HPCTOOLKIT's visualization tool, `hpcviewer`, presents performance metrics correlated to program structure (Section IV-A) and mapped to source code (if available). We highlight two features of `hpcviewer` here and postpone additional details for the case studies of Section V.

The first important feature of `hpcviewer` is the ability to manipulate and rank order static program structure. Program structure, normally a hierarchy (a tree), can be too rigid for effective analysis. Consequently, `hpcviewer` allows the user to 'relax' parent-child hierarchical relationships to, *e.g.*, sort all the inner loops in a program by memory bandwidth, comparing them as peers. This allows the analyst to quickly identify the top memory bound loops without regard for which source file or even library from which they derive.

The second important feature of `hpcviewer` is the ability to manipulate dynamic calling structure. In particular, `hpcviewer` supports three distinct views of an application's call path structure.

- The *top-down* view of the calling context tree associates metrics with their calling context (see Figure 2). The root of the tree is the program entry point; the path from any node to the root represents its calling context. For any given context, `hpcviewer` computes both *exclusive* and *inclusive* metric values for that context. Inclusive metrics include values associated with that context plus all of its callees; exclusive metrics subtract the values of callees from that total.
- If the top-down view looks 'down' the call chain, the *bottom-up* view (see Figure 3) looks 'up' to a procedure's callers. This view enables one to understand how the costs of a modular component, such as a set used in several different contexts, may be distributed. In this example, the bottom-up view would show cost metrics attributed to the set and apportion that total cost among its users (callers).
- The *flat* view (see Figure 4) 'flattens' the calling context of each sample point and then combines context-less samples from the same procedure to generate exclusive metrics. A unique aspect of this view is that because it is

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

[3]Without line map information, `hpcstruct` can still identify procedures and loops, but is not able to account for inlining or loop transformations.

computed from the calling context tree, it also computes *inclusive* metrics for call sites within each procedure.

D. Evaluation

HPCTOOLKIT uniquely combines dynamic call path information with static program structure and provides the bottom-up and flat views to complement the standard top-down view. Its novel method of binary analysis exposes important compiler optimizations such as inlining or without incurring run time overhead or relying on source code. In contrast, VTune does not expose program structure by identifying inlined frames.

V. CASE STUDIES

To demonstrate HPCTOOLKIT's capabilities for analyzing the performance of modular applications, we present several screen shots of the `hpcviewer` browser displaying performance data collected for two modern scientific codes under development with funding from the Department of Energy's Office of Science.

A. Chroma's `hmc`

We first demonstrate the detailed attribution of performance data for `hmc`, a C++ application for lattice quantum chromodynamics built upon the Chroma library [9]. Chroma employs a highly modular design that makes extensive use of C++ expression templates. Because of its use of expression templates, at compile time complex templates are instantiated, customized for the many different contexts in which they are used, and sometimes inlined. Consequently, `hmc` can take hours to compile and yields very large executables (approximately 110MB) on a dual-socket dual-core 2.2 GHz Opteron (275) system running Linux and with 4 GB memory per socket). We use the GCC 4.1.1 compiler with the `-O3` option.

Figure 2 shows a calling context tree view of a call path profile of `hmc`. 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. One can see procedure activations along call paths interspersed with loops within the procedures. The selected line in the navigation pane and the source pane (top sub-pane) shows the procedure `globalSumArray`, which has been inlined into its caller shown returning type `SystemSolverResults_t`[4]. Our use of binary analysis to recover information about loops and inlined code is unique.

In the calling context tree view of `hmc` shown in Figure 2, two columns of metric data are shown: inclusive and exclusive time for timer-based sampling, denoted as "samples (I)" and "samples (E)." The inclusive times show both the absolute time spent in each context shown in the navigation pane. Further detail about the costs associated with any node in the tree can be obtained by opening the node to look at the costs attributed

[4]The full name of the caller derives from an expression template and is too long to reproduce here.

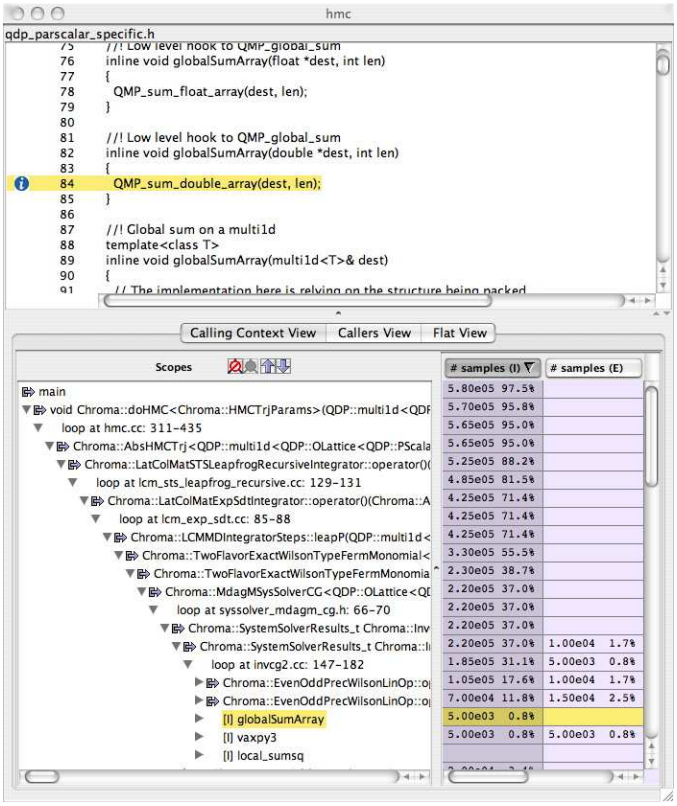

Fig. 2. A *calling context* view for `hmc`.

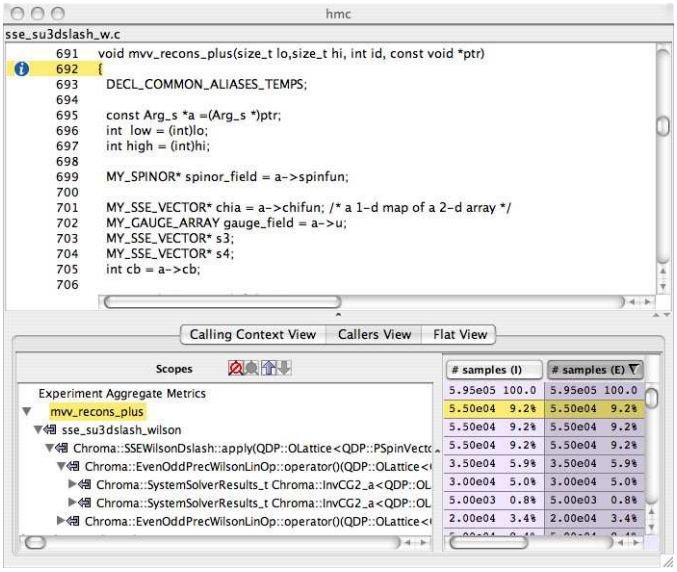

Fig. 3. A *caller's view* for `hmc`.

to calls, loops or inlined code within. The inclusive time metric in the calling context tree view for the call to `SystemSolve` shows that 37% of the time was spent in that routine. The exclusive time metric shows that of that 37%, 1.7% is spent in the routine itself, the rest in routines it calls. Immediately below, we can see that 0.8% of the total execution time is spent directly within the loop at lines 147–182 of `invcg2.cc`.

This fine level of details shows the power of combining static information from the application binary with dynamic call path measurements.

Figure 3 shows a bottom up caller's view of a call path profile of `hmc`. This figure highlights the most costly routine in the execution `mvv_recons_plus`. The caller's view shows that the all of the cost attributed to this routine were incurred in calls from `sse_su3dslash_wilson`, which was called from an expression template for `SSEWilsonDslash::apply`. This template instantiation is invoked from several distinct instantiations of the expression template for `EvenOddPrecWilsonLinOp::operator()`. We can see that of the 9.2% total time spent in `mvv_recons_plus`, 5.9% of the total time was incurred on behalf of the first instantiation of the `EvenOddPrecWilsonLinOp::operator()`. In the navigation pane, selecting the icon next to the name of the caller navigates the source display to the call site of the callee; selecting the name of the caller navigates to the start of the caller's routine. This example shows the power of our tools for attributing costs incurred in a routine to the multiple contexts in which it was called.

B. S3D

The second application we discuss is `S3D`, a Fortran code being developed at Sandia National Laboratory to support high fidelity simulation of turbulent reacting flows [11]. It is primarily written in Fortran 90, with some supporting routines written in Fortran 77. `S3D` is the focus of analysis and optimization efforts to prepare it for large-scale simulation runs on the Cray XT3/XT4 at Oak Ridge National Laboratory. For this study, we compiled `S3D` on our Cray XD1 using the Portland Group's `pgf90` compiler, version 6.1.2, using the `-fast` option.

Figure 4 shows part of a loop-level flat view of a timer-based call path profile of a single-processor execution of `S3D`. This 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. The top line in the flat view shows that 13.6% of execution time is spent in an unknown file in `S3D`. Unflattening one level would show that this cost represents time spent in the PGI's implementation of the Fortran `exp` operation. Here, we focus on the third loop on lines 209-210 of file `rhsf.90`. We notice that this loop contains a loop at line 210 that does not appear explicitly in the code. We can see that this loop consumes 5.4% of the total execution time, more than the 5.2% of the time spent in `computescalargradient`! This loop, discovered by `hpcstruct`, represents the time spent repeatedly copying a non-contiguous 4-dimensional slice of array `grad_Ys` into a contiguous array temporary before passing it to `computescalargradient`. In other portions of the code, `hpcstruct` recovered several loops representing the scalarization of Fortran 90 vector statements. The ability

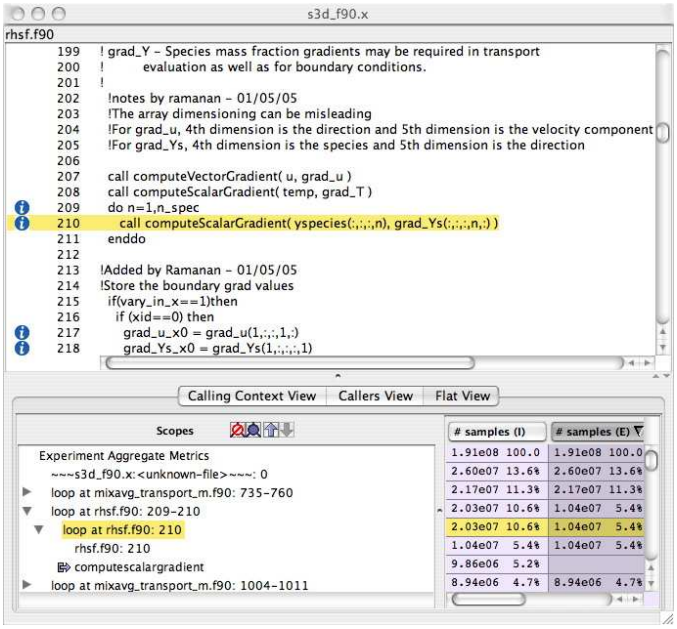

Fig. 4. `hpcviewer`'s *flat view* exposing loops for `S3D`.

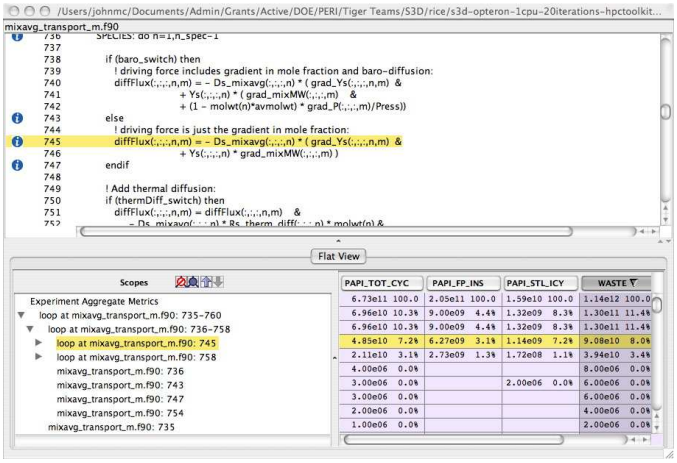

Fig. 5. A loop-level view of a flat hardware performance counter profile for `S3D`.

to explicitly discover and attribute costs to such compiler-generated loops is a unique strength of our tools.

In the views presented thus far about `S3D`, one can see how much time is spent in certain contexts, but with only time costs, it is impossible to tell if the computation is efficient or not. Only examining multiple metrics can show if the computation is efficient. Figure 5 shows a view of loops in the `S3D` application correlated with various measured metrics and sorted by a user-defined waste metric. Our waste metric represents the total number of floating point issue slots that were unused in each context. We compute waste for each context as $2 \times \text{cycles} - \text{FLOPS}$, which corresponds to the maximum number of FLOPS that could have been performed in the context (based on the number of cycles spent there and the maximum rate floating point operations could have been

executed) minus the actual number of floating point operations performed. Sorting by this waste metric shows us where we have underutilized the floating point unit the most. In contrast, computing ratios of FLOPS per cycle for various contexts (*e.g.*, loops, routine, and program) can show how efficiently the application is performing; however, these ratios do not tell us whether the time spent in a particular computation is *significant* with respect to the overall performance of the execution. Our waste metric does; sorting by high waste scores pinpoints opportunities for tuning.

VI. Conclusions and Future Directions

We have described a unique methodology for analyzing the performance of an application's execution, including several novel measurement and analysis techniques. This methodology is unique in two ways.

First, our techniques are coherent and mutually complementary. To achieve coherence, we have focused on both accurate measurement and effective analysis. If measurement includes systematic error, insightful presentation is beside the point. Alternatively, poor presentation of excellent data obscures and hinders insight. We have complemented sampling-based profiling with binary analysis to enable accurate measurement and recover source code structure.

Second, it is comprehensive and capable of identifying performance issues in real applications. `hpcrun` samples the whole calling context of an unmodified fully optimized binary 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). In sum, we can measure *what actually executes* and present it in an effective way that exposes details, but within the context of larger abstractions.

We are actively working on several new directions. One area of critical importance with the rise of multicore architectures is understanding the performance of multithreaded applications. While HPCToolkit currently measures and analyzes multiple threads of execution, each thread remains distinct from every other thread. That is, it takes a systems view of threads, where each thread is 'physically' a sibling of every other thread. However, application programmers often have a different *logical* model of threads where a thread executes within the logical context of another thread, such as in fork-join parallelism. We are nearing completion of an extension that reconstructs this logical model by locating a thread's execution within the context in which it was created. Similarly, parallel languages often have a logical model of threaded execution that is only indirectly preserved by the run time system. One example of this is Cilk [4], which executes asynchronous calls using work stealing. While the application programmer views the application in terms of one large dependency DAG (directed acyclic graph), Cilk's run time scheduler partitions the DAG into unrelated pieces for execution by a fixed-sized pool of threads. A standard profile of a Cilk application reveals the work assigned to each thread over the course of the computation, effectively scatting a single logical calling context randomly over the thread space. We are working on an efficient way to preserve enough information during measurement to reconstruct the mapping between the logical execution DAG and the actual parallel decomposition.

In the future, we plan to extend our methodology in two important ways. First, we plan to develop support for introspective analysis to cooperate with dynamic optimization. Second, we plan to design ways to move from descriptive to prescriptive performance statistics. For example, instead of computing a metric to expose under-performing loops and burdening the analyst with determining the appropriate corrective action, we envision a tool that recommends a series of specific transformations (*e.g.*, unroll-and-jam loop i by 3) that appropriately closes the loop's performance gap.

Acknowledgments

We acknowledge the contributions of our past and present colleagues Robert Fowler, Nathan Froyd, Michael Fagan, and Mark Krentel. Without their efforts, HPCToolkit would not exist in its present form.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[2] K. W. Cameron, Y. Luo, and J. Scharmeier. Instruction-levle microprocessor modeling of scientific applications. In *ISHPC 1999*, pages 29 – 40, Japan, May 1999.

[3] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of spmd codes using expectations. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 13–22, New York, NY, USA, 2007. ACM.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[5] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *GCC Summit '06: Proceedings of the GCC Developers' Summit, 2006*, pages 21–36, 2006.

[6] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.

[7] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference (SHPCC)*, pages 841–850, 1994.

[8] Intel Corporation. Intel VTune performance analyzers. http://www.intel.com/software/products/vtune/.

[9] Jefferson Lab. The Chroma library for lattice field theory. http://usqcd.jlab.org/usqcd-docs/chroma.

[10] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[11] D. Monroe. ENERGY Science with DIGITAL Combustors. http://www.scidacreview.org/0602/html/combustion.html.

[12] Rice University. HPCToolkit performance tools. http://www.hipersoft.rice.edu/hpctoolkit, 2007.

[13] N. 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.

[14] USQCD. U.S. lattice quantum chromodynamics. http://www.usqcd.org.