Application Performance Profiling on the Cray XD1 using HPCToolkit[*]

John Mellor-Crummey[1] Nathan Tallent[1] Michael Fagan[1] Jan E. Odegard[2]

[1] Department of Computer Science, MS 132
[2] Computer and Information Technology Institute, MS 39
Rice University
6100 Main Street, Houston, TX 77005-1892.
{johnmc,tallent,mfagan,odegard}@rice.edu

Abstract

HPCToolkit is an open-source suite of multi-platform tools for profile-based performance analysis of sequential and parallel applications. The toolkit consists of components for collecting performance measurements of fully-optimized executables without adding instrumentation, analyzing application binaries to understand the structure of optimized code, correlating measurements with program structure, and a user interface that supports top-down analysis of performance data. This paper provides an overview of HPCToolkit and demonstrates its utility for application performance analysis on a Cray XD1.

1 Introduction

Modern microprocessors such as AMD's Opteron achieve high performance by employing a diverse collection of strategies. Opterons employ a superscalar design with out-of-order instruction issue, pipelined functional units, short vector parallelism, a hierarchy of caches, a translation lookaside buffer for fast translation of virtual to physical addresses, non-blocking memory operations, and hardware support for prefetching data into cache. As a result, achieving top application performance requires tailoring applications to effectively exploit the capabilities of this bewildering array of features.

Nearly a decade ago, Rice University began developing a suite of performance tools now know as HPC-Toolkit. This effort initially began with the objective of building tools that would help guide our research on compiler technology. As our tools matured, it became clear that they would also be useful for application developers attempting to harness the power of parallel systems such those available from Cray today. Since HPCToolkit was developed in large part for our own use, our goals for its design were that it be simple to use and yet provide fine-grain detail about application performance bottlenecks. We have achieved both of these goals.

This paper provides an overview of HPCToolkit and its capabilities. HPCToolkit consists of components for collecting performance measurements of fully-optimized executables without adding instrumentation, analyzing application binaries to understand the structure of optimized code, correlating measurements with program structure, and a user interface that supports top-down analysis of performance data. Section 2 outlines the design principles that shaped HPCToolkit's development and provides an overview of some of HPCToolkit's key components. Section 3 describes HPCToolkit's components in more detail. Section 4 presents some screenshots of HPCToolkit's user interface to demonstrate the utility of our tools for analyzing the performance of complex scientific applications running on a Cray XD1. The paper concludes with a brief status report that outlines our ongoing efforts to enhance the tools.

[*]Project URL: `http://www.hipersoft.rice.edu/hpctoolkit`. This paper was presented at the Cray User Group meeting, May 7–10, 2007, Seattle, Washington.

2 Design Principles

Here we enumerate the design principles that form the basis for HPCToolkit's approach.

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.

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 [7] from the US Lattice Quantum Chromodynamics project [16] required seven hours of recompilation [12] (on a non-Cray machine).

Binary instrumentation, such as that performed by Dyninst [8] or Pin [9], 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` [5], a well-known call graph profiler, and the SPEC integer benchmarks showed that on average `gprof` dilates execution time by 82% [4]. 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.

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 to insert instrumentation at any one level, nor is it sufficient 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.

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.

Derived performance metrics are essential for effective analysis. Derived measures such 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 for ranking and sorting.

Performance analysis should be top down. It is unreasonable to require users to hunt through mountains of printouts or many screens full of data in multiple windows 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.

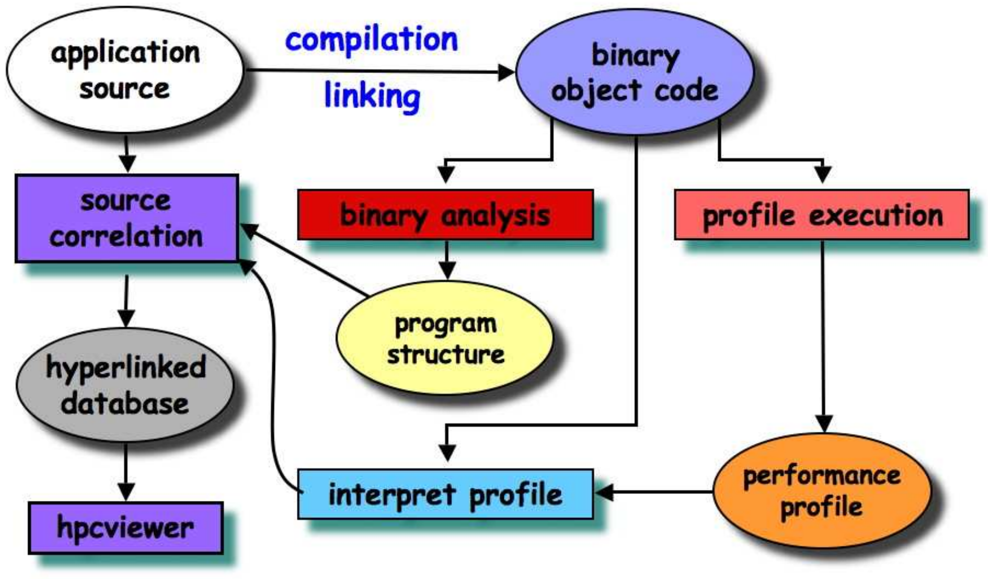


Figure 1: Overview of HPCToolkit tools workflow.

Hierarchical aggregation is important in the face of approximate attribution. In modern multi-issue microprocessors 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.

With instruction-level parallelism, aggregate properties are vital. Even if profiling instrumentation could provide perfect attribution of costs to executable instructions and if 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.

3 HPCToolkit

HPCToolkit supports 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.

Figure 1 provides an overview of HPCToolkit's components and the workflow of using them. First, one compiles and links one's application. 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. Some compilers always include line map information; for others, one must add a variant of a `-g` flag to the compiler's command line in addition to optimization flags. Second, one launches an application with either of HPCToolkit's measurement tools. These tools use statistical sampling to collect a performance profile. Third, one invokes HPCToolkit's tool for analyzing the application binary to recover information about files, functions, loops, and inlined code. Fourth, one uses another HPCToolkit component to combine 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. In the following sections, we describe 1) our measurement approach; 2) gathering program structure with binary analysis; and 3) our interactive performance data viewer. In section 4, we identify some performance issues in two leading scientific applications on the Cray XD1 using the capabilities of HPCToolkit.

3.1 Performance Measurement

Performance measurement using HPCToolkit avoids instrumentation because of the overhead and distortion that it typically adds. For this reason, HPCToolkit employs *statistical sampling*.

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 other 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.

Sampling triggers. Different kinds of event triggers lead to different measurements of program performance. Event triggers can be either asynchronous and 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.

Measuring dynamically-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.

Flat profiling. HPCToolkit supplies a lightweight profiler called `hpcrun` 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 yields the best results when a program's call graph is a tree.

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 performance profile sample is the set of procedure frames active on the call stack at the time the sample is taken. We refer to the state of the call stack as the *calling* *context* of the sample, and 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 `csprof`.

When synchronous or asynchronous events occur, `csprof` records the *full calling context* for each event. A calling context collected by `csprof` 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 samples were to become a concern, we could collapse calling contexts for recursive calls [1] or record only a suffix of full contexts.

Rather than storing the call path independently for each sample, we represent all of the call paths observed by samples 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.

Maintaining control over applications. For HPCToolkit to maintain control over an application, certain calls to standard C library functions must be intercepted. For instance, HPCToolkit 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 HPCToolkit. To intercept such function calls in dynamically-linked executables, the profiler uses library preloading to interpose its own wrapped versions of library routines.

Handling dynamic loading. Modern operating systems such as Linux also enable programs to load and unload shared libraries at run time, a process known as *dynamic loading*. Dynamic loading presents the possibility that a several different functions may be mapped to any particular address over the execution of a program. As `hpcrun` and `csprof` only collect a sequence of one or more program counter values when a sample is taken, some provision must be made for mapping these program counters to the functions that contained them during post-mortem analysis. 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.

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` and `csprof` intercept thread creation and destruction to initialize and finalize profile state.

3.2 Correlating Measurements with Source Code Structure

Modern scientific codes frequently employ sophisticated object-oriented design. In these codes, deep loop nests are often spread across multiple routines. To achieve high performance, such codes rely on compilers to inline routines and optimize loops. Consequently, to effectively interpret performance, transformed loops must be understood in the calling context of transformed routines.

Figure 2 shows an example of a simple C++ program designed to test the effectiveness of compiler optimizations in hiding a simple object-oriented abstraction. In this example, the `Mp` class is derived from the Standard Template Library's (STL) `map` class template and given a virtual member function `Mp::add` to wrap the insertion of elements. Far from being purely academic, the code represents a common and useful way of building C++ classes and of using STL and STL-influenced containers. In particular, STL's `map` is such a useful abstraction, that a developer might easily use it in place of a hash table, even though it is

```
    class Mp :  public std::map<int, double> {
    public:
      Mp() { }
      virtual ~Mp() { }
      virtual void add(int i, double d) { insert(std::make_pair(i, d)); }
    };

    int main() {
      Mp m;
L1    for (int i = 1; i < 1000; ++i) {
S1      m.add(i, (double)i);
      }

L2    for (int i = 1; i < 10000000; ++i) { // Ten million
S2      m.add(i + 1000,     (double)i);
S3      m.add(i + 10000000, (double)i);
      }
    }
```

Figure 2: Source code for testing C++'s STL `map`.

implemented using balanced trees and therefore does not provide the amortized time bound guarantees that are typical of a hash table. (This example uses `int` keys rather than strings to represent the more reasonable decision of using `map` with a pointer-valued key.) When executed, loops L_1 and L_2 insert a combined total of approximately 20 million items into an instance of `Mp`.

Even though Figure 2 contains only a few lines of code, optimizing compilers often make significant transformations to such code improve performance. GCC 4.1 on a Cray XD1 inlines both `add` and `map<>::insert` into the call sites at S_1, S_2 and S_3. It inlines both the `Mp` destructor and the corresponding `map<>` destructor into `main`. Several template instantiations internal to the STL implementation are inlined into other template instantiations. Other compilers perform different transformations. For instance, the Intel 9.1 (Itanium) compiler does *not* inline `add` in the call site at S_3, though it does at S_1 and S_2. Moreover, it fuses loops L_1 and L_2, apparently, noting that each item inserted into the map has a unique key. (We discovered all of these facts using HPCToolkit.) Consequently, to meaningfully understand the performance of such a program, it is necessary to correlate performance data with the *optimized* binary.

3.2.1 Correlation for Optimized Programs

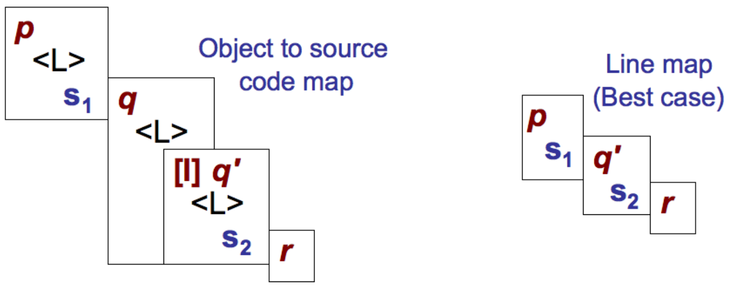
To combine performance data with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure. Since the most important elements of the source code structure are procedures and loop nests — procedures embody the actual executable code while loops often consume the bulk of the executable time — we focus our efforts recovering them. An example of what this mapping might look like is shown in Figure 3. In this example, the object to source code structure map is represented as a tree of scopes, where a load module (the binary) contains source files; files contain procedures; procedures contain loops; procedures and loops contain statements; and where scopes such as procedures, loops and statements can be annotated with object code address interval sets. This object to source code mapping should be contrasted with the binary's line map, which maps an object address to its corresponding source file, procedure name and line number to enable stepping by a debugger. In particular, the line map is fundamentally line based.

As an example of how HPCToolkit's correlation tool can use the object to source code structure mapping, consider Figure 4, which shows two possible representations for the call path fragment $\ldots s_1 \rightarrow s_2 \ldots$ where s_1 and s_2 are call sites. Assume that s_1 represents a call from procedures $p \rightarrow q$ and s_2 a call from procedures $q' \rightarrow r$. Further assume that both s_1 and s_2 are located within loops and that q's source code contains a call site $s' : q \rightarrow q'$ which has been replaced by an inlined version of q'. In other words, in contrast to what was actually executed, source code analysis would expect the call path fragment to be

```
<LM n=".../hmc" base_addr="0x4000000000000000">                    load module
  <F n=".../hmc.cc">                                                source file
    <P n="doHMC" l="257-449" addr="[0x1eac0-0x21720)">             procedure
      ...
      <S l="309-309" addr="[0x1f1b6-0x1f1c6)..."/>                  statement
        <L l="311-435" addr="[0x1f460-...)">                        loop
          ...
          <S l="313-313" addr="[0x1f250-0x1f256)..."/>
          ...
        </L>
      ...
    </P>
  </F>
...
</LM>
```

An object to source code structure mapping represented as a static scope tree expressed in XML. Static scopes include a load module (LM), file (F), procedure (P), loop (L) and statement (S). Procedures, loops and statements are annotated with corresponding object address interval sets.

Figure 3: An object to source code structure mapping.



Two possible representations for the call path fragment $\ldots s_1 \rightarrow s_2 \ldots$, where s_1 and s_2 are call sites and where s_1 represents a call from p to q and s_2 a call from q' to r.

Figure 4: Combining call path profiles with static program structure.

$\ldots s_1 \rightarrow s' \rightarrow s_2 \ldots$. Figure 4 shows s_1 and s_2 surrounded by two versions of their static program structure. The left hand side shows how HPCToolkit presents this performance data fragment while the right hand side shows the best possible presentation using only line map information. HPCToolkit identifies that both s_1 and s_2 are located within loops (indicated by `<L>`). Moreover, even though the optimized code no longer contains s', HPCToolkit infers that the dynamic call path is missing a call through procedure q and nests s_2 within both its original procedure q' and its new host procedure q.[1] In contrast, by using the line map a tool can at best identify s_1's containing procedure (p) and s_2's original procedure (q'). It should be noted that this best case is actually serendipitous, because the line map's information for q is (in general) fundamentally ambiguous as to whether s_2 is located within q' or q. Moreover, the line map is insufficient to accurately determine the source line bounds of p, q' and r. In contrast, HPCToolkit's object to source code mapping typically computes the source line bounds of p, q and r exactly, though it only approximates q''s; usually the source line bounds of the loops within p, q' and r are recovered exactly. The most important benefit of the object to source code structure mapping is that reconstructed procedures, loops and inlined frames can be treated as 'first-class' entities for the purpose of assigning cost metrics.

[1]Currently HPCToolkit can only identifies one level of inlining. Nested inlining is marked as inlined code, but flattened with respect to the host procedure or enclosing loop.

3.2.2 `bloop`: Recovering Program Structure from Optimized Binaries

To recover an object to source code structure mapping from binaries for optimized applications, we built the `bloop` [10, 12] binary analysis tool. A key goal of `bloop` is to discover information about inlined procedures and loops. For this purpose, `bloop` analyzes object code instructions and their associated line map information along with DWARF [3, 13] debugging information generated Linux compilers. DWARF allows compilers to generate information describing procedures and data to improve the functionality of a debugger. Since most compilers do not generate elaborate DWARF, `bloop` bases its algorithms on a 'lowest-common-denominator' subset of DWARF.

To construct the object to program structure mapping, `bloop` first reconstructs an 'outline' of all the procedures within the binary, locating them within their source file and within any enclosing procedure (in the case of procedure nesting). By making inferences from typical DWARF information, `bloop` is able to recover accurate bounds for procedures which enables it to identify inlined code.

Having an outline of the procedure hierarchy, `bloop` recovers the loop nesting structure for each object code procedure. This task can be broadly divided into two components: 1) analyzing object code to find loops and 2) inferring a source code representation from them. To find loop nests within the object code, `bloop` first decodes the instructions in a procedure to compute the control flow graph (CFG) and then use Havlak's algorithm [6] to recover the tree of loop nests. Given this tree of object code loops, `bloop` then recovers a source code representation for them. This is a challenging problem because with fundamentally line-based information (from the line map) — DWARF has no way to represent information about loops — `bloop` must distinguish between 1) loops that contain inlined code, 2) loops may themselves be inlined and 3) loops that may be inlined *and* contain inlined code. Surprisingly, a small set of relatively simple heuristics allows `bloop` to distinguish between these three cases and identify accurate loop bounds in the vast majority of cases, even in the presence of complex loop transformations such as software pipelining.

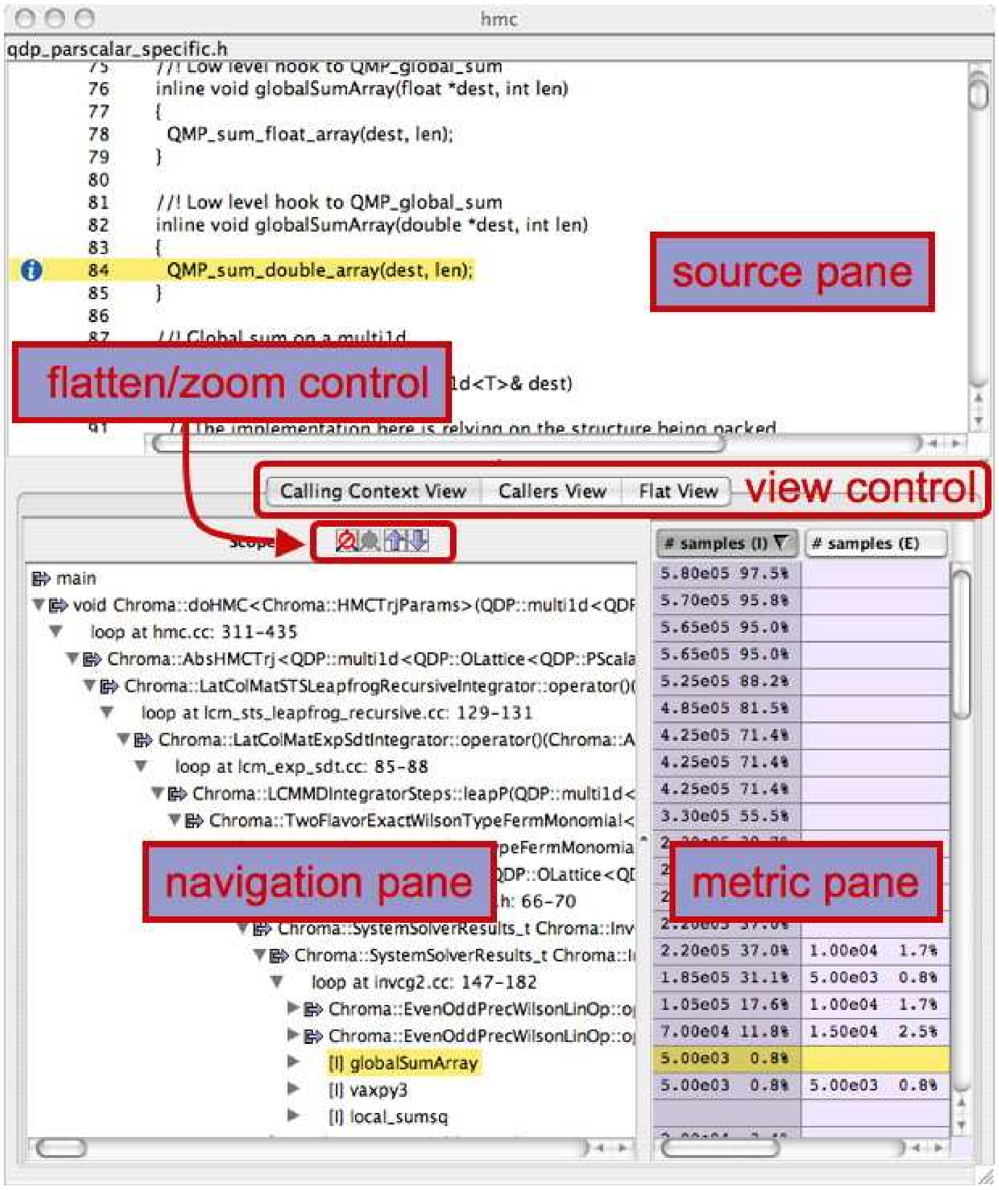
Although `bloop` has recovered a source code representation for loops, at this point, it has not yet accounted for loop invariant code motion and loop transformations such as software pipelining. Because of this, the same line instance may be found both outside of a loop and within it (*e.g.*, partial loop invariant code motion) or there may be duplicate nests that appear to be siblings (*e.g.*, iteration space splitting). To account for compiler loop transformations, we have developed normalization passes based on the observation that a particular source line (statement) appears uniquely in the program's source code. The combination of `bloop`'s heuristics and normalizations enable it to recover very accurate loop nests in practice. However, because it does base loop recovery on heuristics, it is important to note that the effects of an erroneous inference are limited to at most *one* procedure.

Although `bloop` often recovers very accurate program structure even in the presence of complex inlining and loop transformations, it is dependent on accurate debugging information. One implication of this is because compilers typically do not record or provide any information about macro expansion, `bloop` is unable to identify a macro function as 'inlined.' Another implication is that since most debugging information is line based, `bloop` can only distinguish between program constructs that are on distinct source lines. Finally, because compilers do not take advantage of DWARF's ability to record inlining decisions (except GCC), identifying alien code and recovering nested inlining is much more difficult than it could be. Currently, `bloop` does not attempt to recover the nesting structure of nested inlining.

The primary benefit of using a binary analyzer such as `bloop` to discover program structure is that it allows HPCToolkit to expose the structure of what is actually executed. `bloop` identifies transformations to procedures such as inlining and accounts for transformations to loops. One particularly noteworthy result is that `bloop`'s program structure naturally reveals such transformations as loop fusion and the generation of scalarization loops to implement Fortran 90 array notation.

3.3 Computed Metrics

Identifying performance problems and opportunities for tuning may require synthetic performance metrics. For instance, when attempting to tune the performance of a floating-point intensive scientific code, it is often less useful to know where the majority of the floating-point operations are than where floating-point performance is low. Instead, 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



```
//! Low level hook to QMP_global_sum
inline void globalSumArray(float *dest, int len)
{
  QMP_sum_float_array(dest, len);
}

//! Low level hook to QMP_global_sum
inline void globalSumArray(double *dest, int len)
{
  QMP_sum_double_array(dest, len);
}

//! Global sum on a multi1d
                                    1d<T>& dest)

  // The implementation here is relying on the structure being packed
```

```
main                                                              5.80e05 97.5%
▼ void Chroma::doHMC<Chroma::HMCTrjParams>(QDP::multi1d<QDF        5.70e05 95.8%
  ▼ loop at hmc.cc: 311-435                                        5.65e05 95.0%
    ▼ Chroma::AbsHMCTrj<QDP::multi1d<QDP::OLattice<QDP::PScala     5.65e05 95.0%
      ▼ Chroma::LatColMatSTSLeapfrogRecursiveIntegrator::operator()(  5.25e05 88.2%
        ▼ loop at lcm_sts_leapfrog_recursive.cc: 129-131           4.85e05 81.5%
          ▼ Chroma::LatColMatExpSdtIntegrator::operator()(Chroma::A  4.25e05 71.4%
            ▼ loop at lcm_exp_sdt.cc: 85-88                        4.25e05 71.4%
              ▼ Chroma::LCMMDIntegratorSteps::leapP(QDP::multi1d<    4.25e05 71.4%
                ▼ Chroma::TwoFlavorExactWilsonTypeFermMonomial<      3.30e05 55.5%
                                        peFermMonomia
                                        DP::OLattice<QI
                                        h: 66-70
                    ▼ Chroma::SystemSolverResults_t Chroma::Inv
                      ▼ Chroma::SystemSolverResults_t Chroma::I      2.20e05 37.0%   1.00e04  1.7%
                        ▼ loop at invcg2.cc: 147-182                 1.85e05 31.1%   5.00e03  0.8%
                          ▶ Chroma::EvenOddPrecWilsonLinOp::o        1.05e05 17.6%   1.00e04  1.7%
                          ▶ Chroma::EvenOddPrecWilsonLinOp::o        7.00e04 11.8%   1.50e04  2.5%
                          ▶ [I] globalSumArray                       5.00e03  0.8%
                          ▶ [I] vaxpy3                               5.00e03  0.8%   5.00e03  0.8%
                          ▶ [I] local_sumsq
```

Figure 5: Overview of the `hpcviewer` user interface.

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.

Currently, the `hpcview` tool in HPCToolkit supports synthesis of computed metrics for flat profiles; ongoing work aims to add support for synthesizing computed metrics for call path profiles.

3.4 The `hpcviewer` User Interface

HPCToolkit provides the `hpcviewer` browser for interactive examination of performance databases. Figure 5 shows a screenshot of a brower window with panes and key controls labeled. The browser supports three different views of performance data. The browser window is divided into three panes. We first explain the views of performance data and then the role of the different panes.

Views. `hpcviewer` supports three principal views of an application's performance data: a top-down calling context view, a bottom-up caller's view, and a flat view. One selects the desired view by clicking on the corresponding view control tab. We briefly describe the three views and their corresponding purposes.

- *Calling context view.* This top-down view represents the dynamic calling contexts (call paths) in which costs were incurred. Using this view, one can explore performance measurements of an application in a top-down fashion to understand the costs[2] incurred by calls to a procedure in a particular calling context. A calling context for a procedure `f` consists of the stack of procedure frames active when the call was made to `f`. Using this view, one can readily see how much of the application's cost was incurred by `f` when called from a particular calling context. If finer detail is of interest, one can explore how the costs incurred by a call to `f` in a particular context are divided between `f` itself and the procedures it calls. HPCToolkit's call path profiler `csprof` and the `hpcviewer` user interface distinguish calling context precisely by individual call sites; this means that if a procedure `g` contains calls to procedure `f` in different places, these represent separate calling contexts.
- *Callers view.* This bottom up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are used 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 the structure of the parallelization in which the call is made. Serialization or load imbalance may cause long waits in some calling contexts while other parts of the program may have short waits because computation is balanced and communication is overlapped with computation.
- *Flat view.* This view organizes performance measurement data according to the static structure of an application. All costs incurred in *any* calling context by a procedure are aggregated together in the flat view. This complements the calling context view, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

Panes. The browser window is divided into three panes: the navigation pane, the source pane, and the metrics pane. We briefly describe the role of each pane.

- *Source pane.* The source pane displays the source code associated with the current entity selected in the navigation pane. When a performance database is first opened with `hpcviewer`, the source pane is initially blank because no entity is selected in the navigation pane. Selecting any entity in the navigation pane will cause the source pane to load the corresponding file, scroll to and highlight the line corresponding to the selection. Switching the source pane to view to a different source file is accomplished by making another selection in the navigation pane.
- *Navigation pane.* The navigation pane presents a hierarchical tree-based structure that is used to organize the presentation of an application's performance data. Entities that occur in the navigation pane's tree include load modules, files, procedures, procedure activations, inlined code, loops, and source lines. Selecting any of these entities will cause its corresponding source code (if any) to be displayed in the source pane. One can reveal or conceal children in this hierarchy by "opening" or "closing" any non-leaf (*i.e.*, individual source line) entry in this view.

  The nature of the entities in the navigation pane's tree structure depends upon whether one is exploring the calling context view, the callers view, or the flat view of the performance data.

  - In the calling context view, entities in the navigation tree represent procedure activations, inlined code, loops, and source lines. While most entities link to a single location in source code, procedure activations link to two: the call site from which a procedure was called and the procedure itself.
  - In the callers view, entities in the navigation tree are procedure activations. Unlike procedure activations in the calling context tree view in which call sites are paired with the called procedure, in the caller's view, call sites are paired with the calling procedure to facilitate attribution of costs for a called procedure to multiple different call sites and callers.

[2]We use the term *cost* rather than simply *time* since `hpcviewer` can present a multiplicity of measured (e.g. cycles, or cache misses) or derived metrics (e.g. cache miss rates or bandwidth consumed) that that are other indicators of execution cost.

  - In the flat view, entities in the navigation tree correspond to source files, procedure call sites (which are rendered the same way as procedure activations), loops, and source lines.

  The header above the navigation pane contains some controls for the navigation view. In Figure 5, they are labeled as "flatten/zoom control." Depressing the *up arrow* button will zoom in to show only information for the selected line and its descendants. One can zoom out (reversing a prior zoom operation) by depressing the down arrow button. The remaining two buttons for enable one to flatten and unflatten the navigation hierarchy. Clicking on the flatten button (the icon that shows a tree node with a slash through it) will replace each top-level scope shown with its children. If a scope has no children (*i.e.*, it is a leaf), the node will remain in the view. This flattening operation is useful for relaxing the strict hierarchical view so that peers at the same level in the tree can be viewed and ranked together. For instance, this can be used to hide procedures in the flat view so that outermost loops can be ranked and compared to one another. The inverse of the flatten operation is the unflatten operation, which causes an elided node in the tree to be made visible once again.

- *Metric pane.* The metric pane displays one or more performance metrics associated with entities to the left in the navigation pane. Entities in the tree view of the navigation pane are sorted at each level of the hierarchy by the metric in the selected column. When `hpcviewer` is launched, the leftmost metric column is the default selection and the navigation pane is sorted according to the values of that metric in descending order. One can change the selected metric by clicking on a column header. Clicking on the header of the selected column toggles the sort order between descending and ascending.

  During analysis, one often wants to consider the relationship between two metrics. This is easier when the metrics of interest are in adjacent columns of the metric pane. One can change the order of columns in the metric pane by selecting the column header for a metric and then dragging it left or right to its desired position. The metric pane also includes scroll bars for horizontal scrolling (to reveal other metrics) and vertical scrolling (to reveal other scopes). Vertical scrolling of the metric and navigation panes is synchronized.

4 Analyzing Applications with HPCToolkit

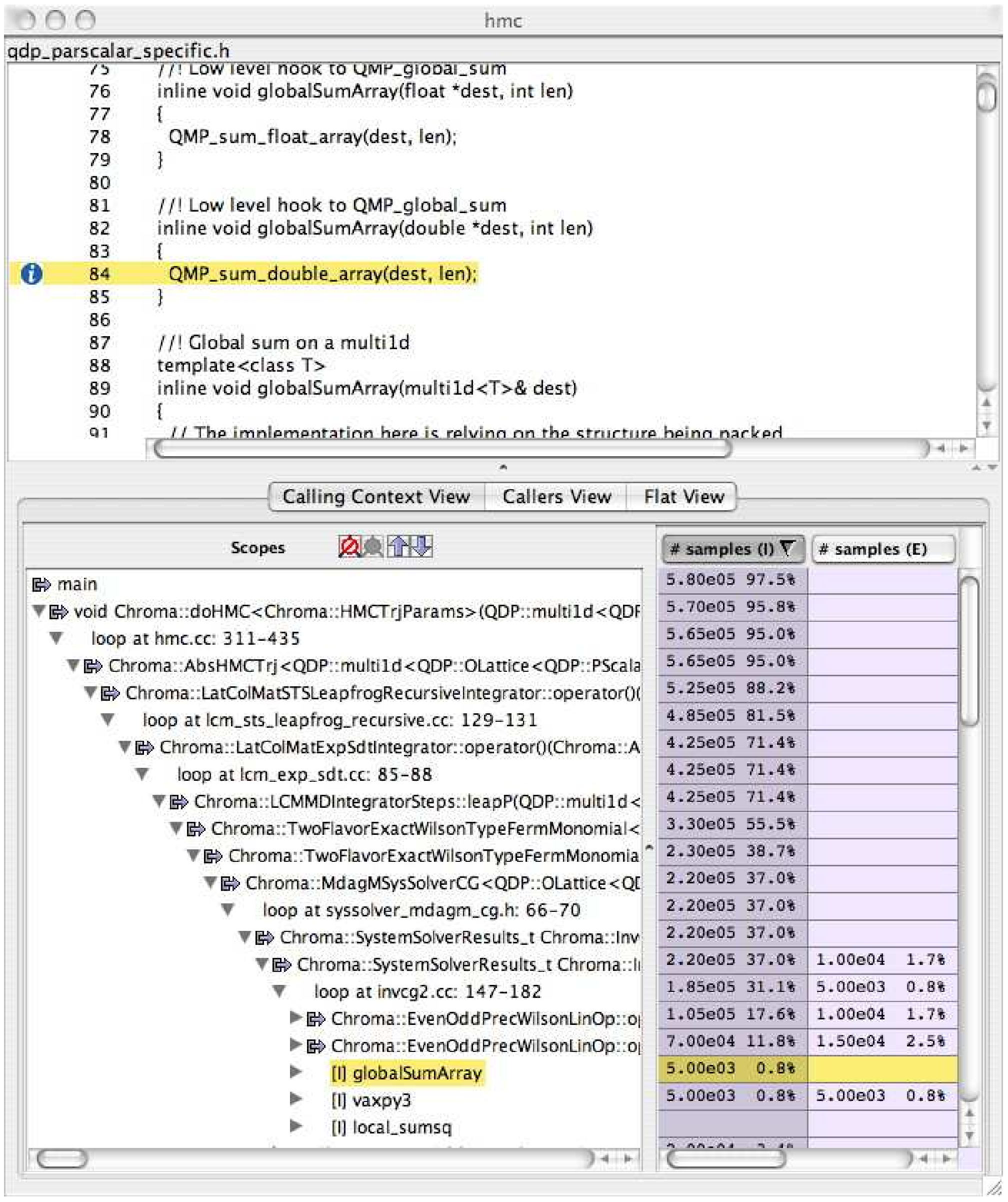
To demonstrate HPCToolkit's capabilities for analyzing application performance on the Cray XD1, we present a few screenshots of the `hpcviewer` browser displaying performance data collected for two modern scientific codes under development with funding from the Department of Energy's Office of Science. The first application is a C++ application for lattice quantum chromodynamics developed as part of the US Lattice Quantum Chromodynamics project [16]. The second application we show is S3D, a Fortran code being developed at Sandia National Laboratory to support high fidelity simulation of turbulent reacting flows [11].

4.1 Chroma's `hmc`

We first demonstrate the detailed attribution of performance data HPCToolkit provides for `hmc`, a parallel application for lattice quantum chromodynamics. `hmc` is built upon the Chroma library [7] for lattice field theory, which was developed as part of the US Lattice Quantum Chromodynamics project [16]. Chroma is based upon QDP++ [14], a C++ framework developed to support a data-parallel programming model for quantum chromodynamics, and QMP [15], a high performance message passing interface for lattice QCD computing.

The QDP++ package, upon which the Chroma library is based, uses 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 Cray XD1. For our study here, we compiled `hmc` with GCC, version 4.1.

Figure 6 shows a calling context tree view of a call path profile of `hmc`. The navigation pane shows a partial expansion of the calling context tree. The information presented in the navigation pane is a fusion of both static and dynamic context information. HPCToolkit's call path profiler measured dynamic call



```
//! Low level hook to QMP_global_sum
inline void globalSumArray(float *dest, int len)
{
  QMP_sum_float_array(dest, len);
}

//! Low level hook to QMP_global_sum
inline void globalSumArray(double *dest, int len)
{
  QMP_sum_double_array(dest, len);
}

//! Global sum on a multi1d
template<class T>
inline void globalSumArray(multi1d<T>& dest)
{
  // The implementation here is relying on the structure being packed
```

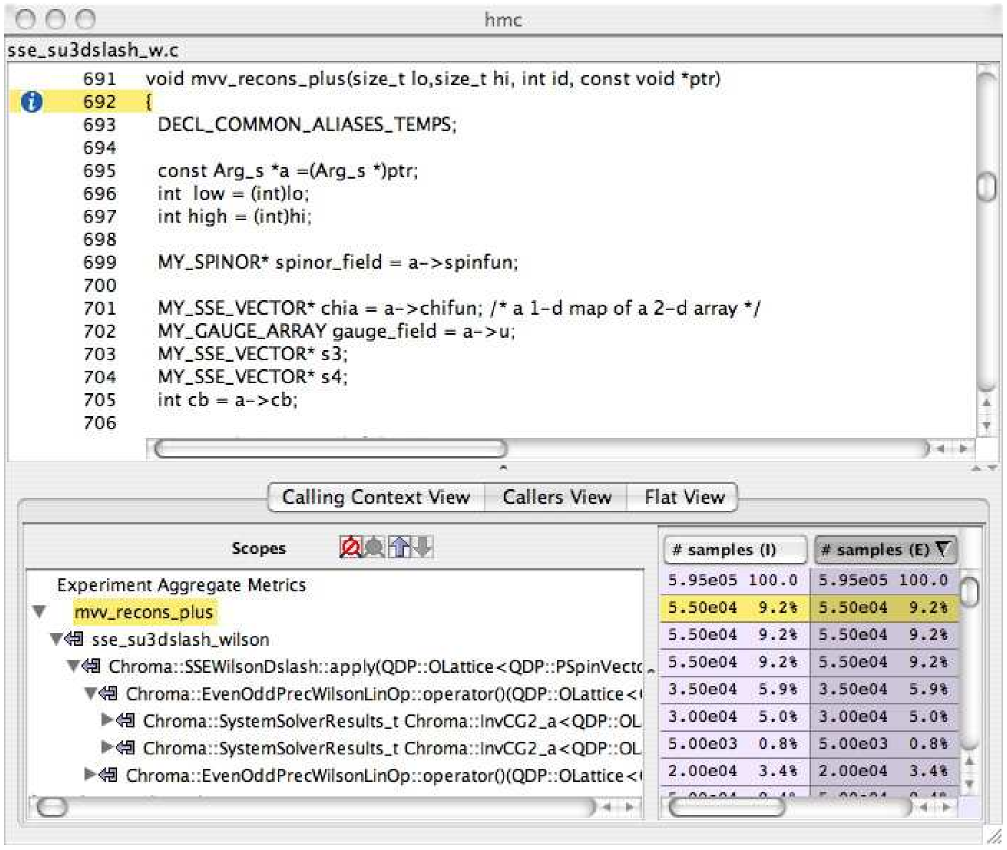
```
Scopes                                                         # samples (I)   # samples (E)
main                                                           5.80e05 97.5%
▼ void Chroma::doHMC<Chroma::HMCTrjParams>(QDP::multi1d<QDF    5.70e05 95.8%
  ▼ loop at hmc.cc: 311-435                                    5.65e05 95.0%
   ▼ Chroma::AbsHMCTrj<QDP::multi1d<QDP::OLattice<QDP::PScala  5.65e05 95.0%
    ▼ Chroma::LatColMatSTSLeapfrogRecursiveIntegrator::operator()( 5.25e05 88.2%
     ▼ loop at lcm_sts_leapfrog_recursive.cc: 129-131          4.85e05 81.5%
      ▼ Chroma::LatColMatExpSdtIntegrator::operator()(Chroma::A 4.25e05 71.4%
       ▼ loop at lcm_exp_sdt.cc: 85-88                         4.25e05 71.4%
        ▼ Chroma::LCMMDIntegratorSteps::leapP(QDP::multi1d<    4.25e05 71.4%
         ▼ Chroma::TwoFlavorExactWilsonTypeFermMonomial<       3.30e05 55.5%
          ▼ Chroma::TwoFlavorExactWilsonTypeFermMonomia        2.30e05 38.7%
           ▼ Chroma::MdagMSysSolverCG<QDP::OLattice<QI         2.20e05 37.0%
            ▼ loop at syssolver_mdagm_cg.h: 66-70              2.20e05 37.0%
             ▼ Chroma::SystemSolverResults_t Chroma::Inv       2.20e05 37.0%
              ▼ Chroma::SystemSolverResults_t Chroma::I        2.20e05 37.0%   1.00e04  1.7%
               ▼ loop at invcg2.cc: 147-182                    1.85e05 31.1%   5.00e03  0.8%
                ▶ Chroma::EvenOddPrecWilsonLinOp::o            1.05e05 17.6%   1.00e04  1.7%
                ▶ Chroma::EvenOddPrecWilsonLinOp::o            7.00e04 11.8%   1.50e04  2.5%
                ▶ [I] globalSumArray                           5.00e03  0.8%
                ▶ [I] vaxpy3                                   5.00e03  0.8%   5.00e03  0.8%
                ▶ [I] local_sumsq
```

Figure 6: hpcviewer displaying a calling context tree view of a timer-based call path profile for hmc.

path information during execution of hmc. This dynamic information was combined with information about inlining and loops recovered by bloop through static analysis of hmc's executable. In the navigation pane, 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 shows the procedure globalSumArray, which has been inlined into its caller shown returning type Chroma::SystemSolverResults_t[3]. HPCToolkit is unique in its use of binary analysis to recover information about loops and inlined code.

In the calling context tree view of hmc shown in Figure 6, 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 to calls, loops or inlined code within. The inclusive time metric in the calling context tree view for

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



```
void mvv_recons_plus(size_t lo,size_t hi, int id, const void *ptr)
{
  DECL_COMMON_ALIASES_TEMPS;

  const Arg_s *a =(Arg_s *)ptr;
  int  low = (int)lo;
  int high = (int)hi;

  MY_SPINOR* spinor_field = a->spinfun;

  MY_SSE_VECTOR* chia = a->chifun; /* a 1-d map of a 2-d array */
  MY_GAUGE_ARRAY gauge_field = a->u;
  MY_SSE_VECTOR* s3;
  MY_SSE_VECTOR* s4;
  int cb = a->cb;

```

Scopes	# samples (I)		# samples (E)	
Experiment Aggregate Metrics	5.95e05	100.0	5.95e05	100.0
mvv_recons_plus	5.50e04	9.2%	5.50e04	9.2%
sse_su3dslash_wilson	5.50e04	9.2%	5.50e04	9.2%
Chroma::SSEWilsonDslash::apply(QDP::OLattice<QDP::PSpinVectc	5.50e04	9.2%	5.50e04	9.2%
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<(	3.50e04	5.9%	3.50e04	5.9%
Chroma::SystemSolverResults_t Chroma::InvCG2_a<QDP::OL	3.00e04	5.0%	3.00e04	5.0%
Chroma::SystemSolverResults_t Chroma::InvCG2_a<QDP::OL	5.00e03	0.8%	5.00e03	0.8%
Chroma::EvenOddPrecWilsonLinOp::operator()(QDP::OLattice<(	2.00e04	3.4%	2.00e04	3.4%

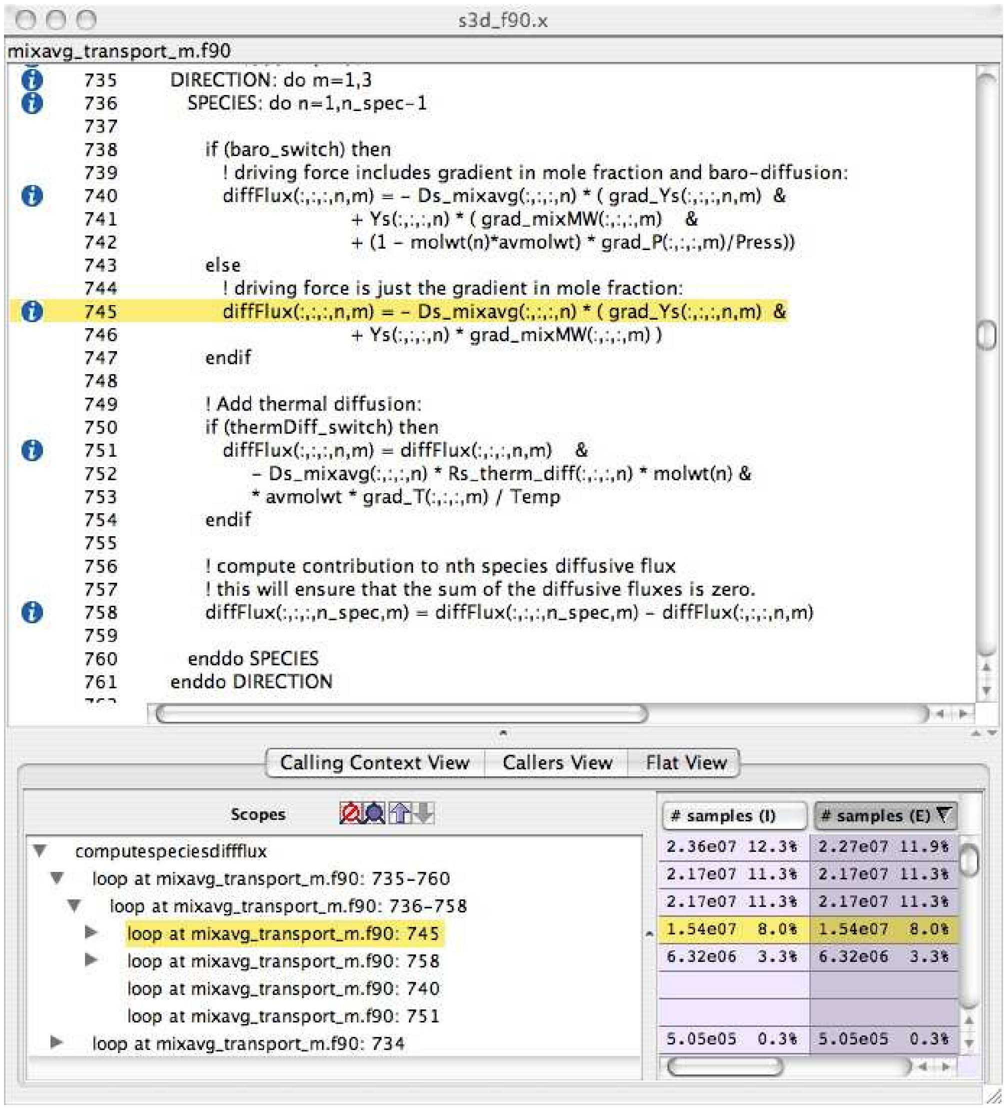
Figure 7: `hpcviewer` displaying a caller's view of a timer-based call path profile for `hmc`.

the call to `Chroma::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. The overhead of statistical sampling with HPCToolkit's call path profiler is typically less than 5%.

Figure 7 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 `Chroma::SSEWilsonDslash::apply`. This template instantiation is invoked from several distinct instantiations of the expression template for `Chroma::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 `Chroma::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 HPCToolkit for attributing costs incurred in a routine to the multiple contexts in which it was called.

4.2 S3D

As described in SciDAC Review [11], the S3D code being developed at Sandia National Laboratories is a massively parallel solver for turbulent reacting flows. The code includes multiple physical and chemical aspects, such as detailed chemistry and molecular transport. S3D uses Direct Numerical Simulation (DNS) for understanding the physics of turbulence. S3D is primarily written in Fortran 90, with some supporting routines written in Fortran 77. This code is a focus of current analysis and optimization to prepare it for



s3d_f90.x

mixavg_transport_m.f90

```
735     DIRECTION: do m=1,3
736       SPECIES: do n=1,n_spec-1
737
738         if (baro_switch) then
739           ! driving force includes gradient in mole fraction and baro-diffusion:
740           diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
741                          + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m)    &
742                          + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Press))
743         else
744           ! driving force is just the gradient in mole fraction:
745           diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
746                          + Ys(:,:,:,n) * grad_mixMW(:,:,:,m) )
747         endif
748
749         ! Add thermal diffusion:
750         if (thermDiff_switch) then
751           diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m)    &
752              - Ds_mixavg(:,:,:,n) * Rs_therm_diff(:,:,:,n) * molwt(n) &
753              * avmolwt * grad_T(:,:,:,m) / Temp
754         endif
755
756         ! compute contribution to nth species diffusive flux
757         ! this will ensure that the sum of the diffusive fluxes is zero.
758         diffFlux(:,:,:,n_spec,m) = diffFlux(:,:,:,n_spec,m) - diffFlux(:,:,:,n,m)
759
760       enddo SPECIES
761     enddo DIRECTION
```

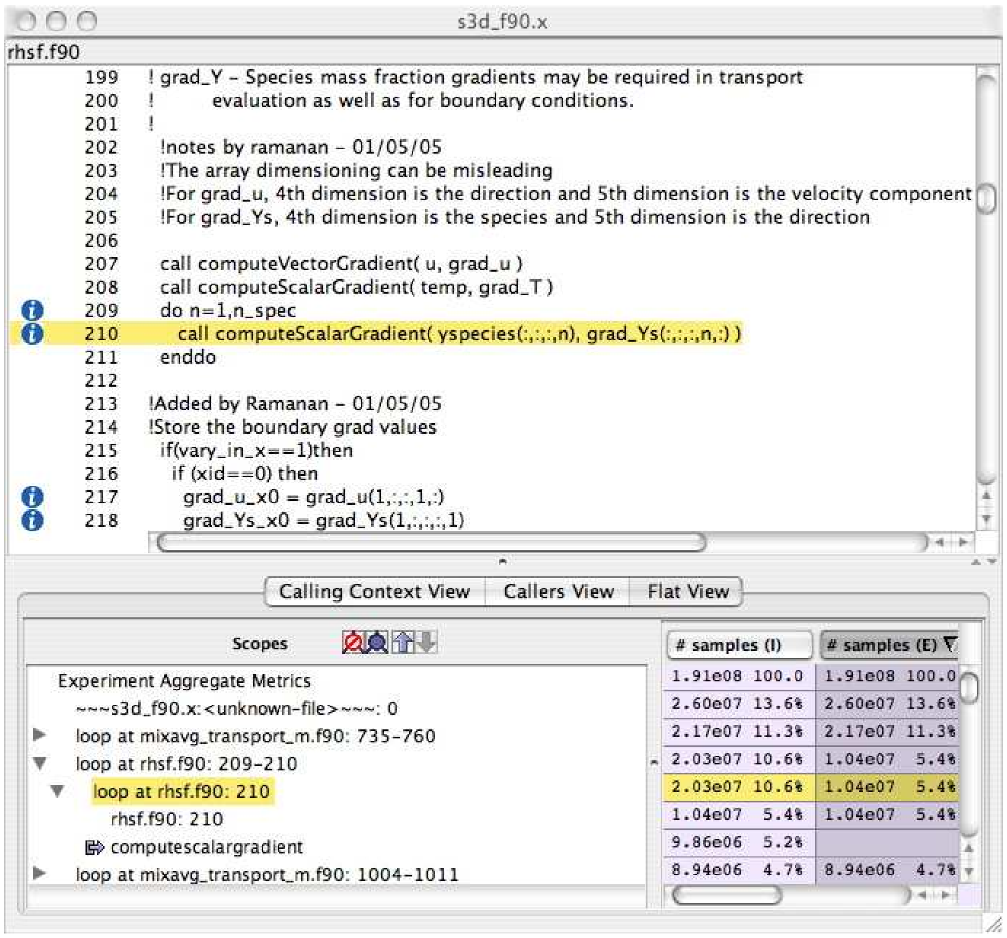
Scopes	# samples (I)	# samples (E)
▼ computespeciesdiffflux	2.36e07 12.3%	2.27e07 11.9%
▼ loop at mixavg_transport_m.f90: 735-760	2.17e07 11.3%	2.17e07 11.3%
▼ loop at mixavg_transport_m.f90: 736-758	2.17e07 11.3%	2.17e07 11.3%
▶ loop at mixavg_transport_m.f90: 745	1.54e07 8.0%	1.54e07 8.0%
▶ loop at mixavg_transport_m.f90: 758	6.32e06 3.3%	6.32e06 3.3%
loop at mixavg_transport_m.f90: 740		
loop at mixavg_transport_m.f90: 751		
▶ loop at mixavg_transport_m.f90: 734	5.05e05 0.3%	5.05e05 0.3%

Figure 8: `hpcviewer` displaying a flat view of a timer-based call path profile for `S3D`.

large-scale simulation runs on a large-scale Cray XT3/XT4 at Oak Ridge National Laboratory. For this study, we compiled this application on our Cray XD1 using the Portland Group's `pgf90` compiler, version 6.1.2, using the `-fast` option.

Figure 8 shows part of a flat view of a timer-based call path profile of a single-processor execution of S3D on a Cray XD1. The source code pane shows a loop over a 5-dimensional data structure. Two loops over the direction and the number of species appear explicitly in the source code. Other 3-dimensional loops are implicit in the Fortran 90 array notation. The navigation pane shows the attribution of costs among the loops in the loop nest in the enclosing `computespeciesdiffflux` routine. Notice that HPCToolkit provides a high level of detail about the application performance. The navigation pane explicitly shows loops representing the Fortran 90 vector statements; the presence of these loops was recovered by HPCToolkit's `bloop` in its analysis of the S3D executable. In this view, one can readily see that the vector statement on line 745 ran more than twice as long as the vector statement on line 758.

Figure 9 shows part of a loop-level flat view of a timer-based call path profile of a single-processor execution of S3D on a Cray XD1. 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


s3d_f90.x

rhsf.f90

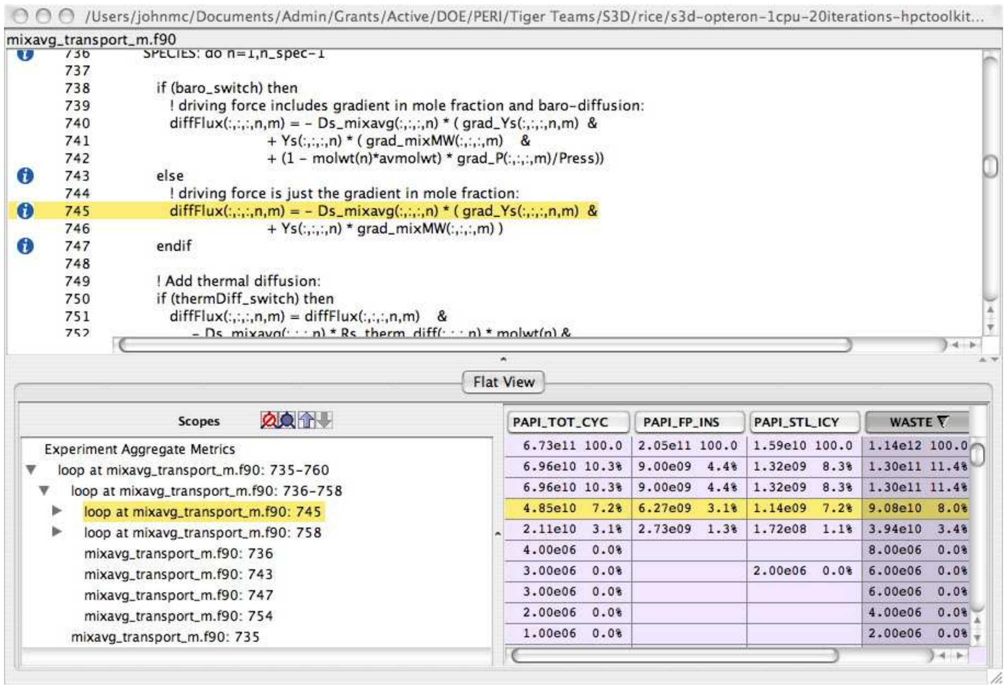
```
199  ! grad_Y - Species mass fraction gradients may be required in transport
200  !         evaluation as well as for boundary conditions.
201  !
202    !notes by ramanan - 01/05/05
203    !The array dimensioning can be misleading
204    !For grad_u, 4th dimension is the direction and 5th dimension is the velocity component
205    !For grad_Ys, 4th dimension is the species and 5th dimension is the direction
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
212
213  !Added by Ramanan - 01/05/05
214  !Store the boundary grad values
215    if(vary_in_x==1)then
216      if (xid==0) then
217        grad_u_x0 = grad_u(1,:,:,1,:)
218        grad_Ys_x0 = grad_Ys(1,:,:,:,1)
```

```
Scopes                                             # samples (I)    # samples (E)
Experiment Aggregate Metrics                       1.91e08 100.0    1.91e08 100.0
  ~~~s3d_f90.x:<unknown-file>~~~: 0                2.60e07  13.6%   2.60e07  13.6%
▶ loop at mixavg_transport_m.f90: 735-760          2.17e07  11.3%   2.17e07  11.3%
▼ loop at rhsf.f90: 209-210                        2.03e07  10.6%   1.04e07   5.4%
  ▼ loop at rhsf.f90: 210                          2.03e07  10.6%   1.04e07   5.4%
      rhsf.f90: 210                                1.04e07   5.4%   1.04e07   5.4%
    computescalargradient                          9.86e06   5.2%
▶ loop at mixavg_transport_m.f90: 1004-1011        8.94e06   4.7%   8.94e06   4.7%
```

Figure 9: `hpcviewer` displaying a flat loop-level view of a timer-based call path profile for `S3D`.

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. The second loop at lines 735-760 was shown in Figure 8. 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 doesn't 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 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`. The ability to explicitly discover and attribute costs to such compiler-generated loops is a unique strength of HPCToolkit.

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 10 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. This



/Users/johnmc/Documents/Admin/Grants/Active/DOE/PERI/Tiger Teams/S3D/rice/s3d-opteron-1cpu-20iterations-hpctoolkit...

mixavg_transport_m.f90

```
736        SPECIES: do n=1,n_spec-1
737
738          if (baro_switch) then
739            ! driving force includes gradient in mole fraction and baro-diffusion:
740            diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
741                            + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m)    &
742                            + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Press))
743          else
744            ! driving force is just the gradient in mole fraction:
745            diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
746                            + Ys(:,:,:,n) * grad_mixMW(:,:,:,m) )
747          endif
748
749          ! Add thermal diffusion:
750          if (thermDiff_switch) then
751            diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m)    &
752              - Ds_mixavg(:,:,:,n) * Rs_therm_diff(:,:,:,n) * molwt(n) &
```

Scopes	PAPI_TOT_CYC	PAPI_FP_INS	PAPI_STL_ICY	WASTE
Experiment Aggregate Metrics	6.73e11 100.0	2.05e11 100.0	1.59e10 100.0	1.14e12 100.0
loop at mixavg_transport_m.f90: 735-760	6.96e10 10.3%	9.00e09 4.4%	1.32e09 8.3%	1.30e11 11.4%
loop at mixavg_transport_m.f90: 736-758	6.96e10 10.3%	9.00e09 4.4%	1.32e09 8.3%	1.30e11 11.4%
loop at mixavg_transport_m.f90: 745	4.85e10 7.2%	6.27e09 3.1%	1.14e09 7.2%	9.08e10 8.0%
loop at mixavg_transport_m.f90: 758	2.11e10 3.1%	2.73e09 1.3%	1.72e08 1.1%	3.94e10 3.4%
mixavg_transport_m.f90: 736	4.00e06 0.0%			8.00e06 0.0%
mixavg_transport_m.f90: 743	3.00e06 0.0%		2.00e06 0.0%	6.00e06 0.0%
mixavg_transport_m.f90: 747	3.00e06 0.0%			6.00e06 0.0%
mixavg_transport_m.f90: 754	2.00e06 0.0%			4.00e06 0.0%
mixavg_transport_m.f90: 735	1.00e06 0.0%			2.00e06 0.0%

Figure 10: `hpcviewer` displaying a loop-level view of a flat hardware performance counter profile for `S3D`.

example highlights HPCToolkit's ability to compute derived metrics.

5 Status

While we use HPCToolkit on a day-to-day basis on a Cray XD1, HPCToolkit it is not currently supported on the Cray XT3 and XT4 platforms. Until spring 2007, the Catamount microkernel lacked kernel support for asynchronous sampling based on hardware performance counters. Development versions of both Catamount and Cray's forthcoming Compute Node Linux operating systems include support for asynchronous sampling. Work will soon begin to port HPCToolkit to these operating systems.

HPCToolkit is currently undergoing major changes to transform it from a research prototype into a tool suitable for production use. Below, we list some of the topics of ongoing work:

- HPCToolkit's profilers currently only support measurement of dynamically-linked binaries. To support Catamount, we are preparing a version of the measurement tools for use with statically-linked applications.
- HPCToolkit's call path profiler currently only collects samples using an interval timer. Its flat profiler supports sampling based on arbitrary hardware performance counter events. The call path profiler will be augmented to support sampling based on hardware performance counters.
- In the absence of compiler information to support call stack unwinding, HPCToolkit's call path profiler may be unable to unwind the call stack if an asynchronous event occurs while a procedure is executing instructions in its epilogue as it prepares to return to its caller. At present, when the call path profiler attempts an unwind and if it fails, it drops the asynchronous sample and notes that it has done so. We are independently pursuing several approaches to avoid loss of samples that occur within code that lacks adequate information to permit asynchronous unwinding (e.g., in epilogues as well as hand-coded math library routines). First, we are coordinating with PGI and PathScale, who have agreed to consider augmenting the unwind information generated by their compilers. Second, we are beginning to explore approaches based on binary analysis and/or emulation that will enable us

to recover sufficient information to support fully-precise asynchronous unwinding even for hand-coded assembler routines (common in math libraries) that use custom calling conventions.

- Previously, we developed a technique for associating counts with edges in the calling context tree in a call path profile [4]. That capability works in a research prototype of a call path profiler that relies on having compiler information to support unwinding in procedure epilogues. We are working to support this capability in HPCToolkit's call path profiler in the absence of compiler support.
- Presently, `bloop`, our binary analyzer, assumes incomplete but correct information. Recent experiments with compilers on the Cray XD1 have shown that they sometimes misattribute procedure calls to lines. Such misattribution can erroneously cause call sites to be reported inside loops. We plan to augment the analyzer to be more defensive against misattribution.
- Currently, `hpcviewer` focuses on presentation of performance data for a single process. We are extending it to perform comparative analysis between processes within and across executions.

Acknowledgments

Development of HPCToolkit is currently being supported by the Department of Energy's Office of Science as part of the SciDAC Performance Engineering Research Institute under Cooperative Agreement No. DE-FC02-06ER25762 and as part of the Center for Scalable Application Development Software under Cooperative Agreement No. DE-FC02-07ER25800. Experiments described in this paper were performed on a Cray XD1 that was purchased with support from the National Science Foundation under Grant CNS-0421109, and a partnership between Rice University, Advanced Micro Devices, and Cray.

References

[1] Glenn Ammons, Thomas Ball, and James 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] Kirk W. Cameron, Yong Luo, and Janes Scharmeier. Instruction-levle microprocessor modeling of scientific applications. In *ISHPC 1999*, pages 29 – 40, Japan, May 1999.

[3] Free Standards Group. DWARF debugging information format, version 3. `http://dwarf.freestandards.org`. 20 December 2005.

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

[5] Susan L. Graham, Peter B. Kessler, and Marshall 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.

[6] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

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

[8] Jon Cargille Jeffrey K. Hollingsworth, Barton P. Miller. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference (SHPCC)*, pages 841–850, 1994.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim 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.

[10] John Mellor-Crummey, Robert Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

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

[12] Nathan 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.

[13] UNIX International. DWARF debugging information format. `http://dwarf.freestandards.org`. 27 July, 1993.

[14] U.S. Lattice Quantum Chromodynamics Project. QDP++: A data-parallel programming environment suitable for Lattice QCD. `http://usqcd.jlab.org/usqcd-docs/qdp++`.

[15] U.S. Lattice Quantum Chromodynamics Project. QMP: A message passing library for lattice QCD. `http://usqcd.jlab.org/usqcd-docs/qmp`.

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