Diagnosing Performance Bottlenecks in Emerging Petascale Applications

Nathan R. Tallent, John M. Mellor-Crummey
Laksono Adhianto, Michael W. Fagan, Mark Krentel
Rice University
tallent,johnmc,laksono,mfagan,krentel@rice.edu

ABSTRACT

Cutting-edge science and engineering applications require petascale computing. It is, however, a significant challenge to use petascale computing platforms effectively. Consequently, there is a critical need for performance tools that enable scientists to understand impediments to performance on emerging petascale systems. In this paper, we describe HPCToolkit—a suite of multi-platform tools that supports sampling-based analysis of application performance on emerging petascale platforms. HPCToolkit uses sampling to pinpoint and quantify both scaling and node performance bottlenecks. We study several emerging petascale applications on the Cray XT and IBM BlueGene/P platforms and use HPCToolkit to identify specific source lines — in their full calling context — associated with performance bottlenecks in these codes. Such information is exactly what application developers need to know to improve their applications to take full advantage of the power of petascale systems.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Measurement techniques, Performance attributes.

General Terms

Performance, Measurement, Scalability, Algorithms.

Keywords

Binary analysis, Call path profiling, Static analysis, Performance tools, HPCToolkit.

1. INTRODUCTION

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

If an application contains significant scaling bottlenecks, it cannot productively use the large number of cores in leadership computing platforms. Unfortunately, it is extremely difficult for applications to effectively use computing resources at this scale because seemingly benign inefficiencies emerge as major bottlenecks on a large number of processors. Understanding why an application does not scale can be quite difficult. To date, approaches to analyze scalability on petascale systems have required laborious human effort [1–3, 13, 22, 36], used instrumentation-based measurement techniques that can significantly dilate execution time and distort the nature of performance measurements [8, 14, 26, 35], or provide only qualitative information [34]. Moreover, at best these approaches only identify scaling bottlenecks at the procedure level because detailed instrumentation at a finer level (*e.g.*, loops) is too costly. As a result, there is a critical need for better tools that can accurately measure and attribute performance information in ways that enable scientists to understand in detail how impediments to scaling arise in parallel applications. Without detailed information about where scaling losses occur, addressing their underlying causes can be difficult.

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

The HPCToolkit[1] project has developed low-overhead techniques for sampling-based performance measurement and analysis that make it possible to precisely quantify and attribute both scalability losses and node performance losses. HPCToolkit can attribute both kinds of losses to individual lines of source code, in their full static and dynamic contexts [7, 33]. However, HPCToolkit's analysis relies on the accurate collection of precise performance measurements. Petascale platforms present two principal challenges to collecting such measurements.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC09 November 14-20, 2009, Portland, Oregon, USA
Copyright 2009 ACM 978-1-60558-744-8/09/11 ...$10.00.

[1] http://hpctoolkit.org

The first challenge is that of scale. One must take care to ensure that measurement approaches do not overly tax shared system resources, *e.g.*, the network or file system. Also, when analyzing data from many cores, reliance on serial algorithms is likely to be problematic. For instance, a measurement approach based on *tracing*, where performance information is distinguished by time, faces significant challenges at scale. Collecting traces at scale can burden file systems and interfere with application and system performance. Even with careful design, trace files can quickly become terabytes in size [37]. Some of these challenges are addressed by on-line data compression, but at the expense of coarser measurements [11]. For this reason we focus on *profiling*, which collapses the time dimension of measurements, as a more readily scalable approach to measurement.

There are different ways to profile. A profiler that uses instrumentation — whether source code [26, 35], static binary [8, 14], or dynamic binary [19] — can introduce significant measurement overhead in programs with small functions. For instance, a previous study [10] showed that simple instrumentation for the `gprof` [12] profiler introduced overhead with a geometric mean of 93% when monitoring the SPEC CPU2000 [30] integer benchmarks. The TAU performance tools [26] reduce instrumentation overhead at the expense of detail through the use of throttling and selective instrumentation [25]. However, selective instrumentation can be problematic because it introduces blind spots, often in critical places such as small, frequently executed routines that lie on hot paths. The alternative to instrumentation is statistical sampling. With an appropriate choice of sampling frequency, sampling-based tools can deliver precise measurements with little overhead. The HPCToolkit performance tools use event-based sampling in combination with call stack unwinding to collect detailed call path profiles; experiments with the SPEC CPU2006 [29] benchmarks show that HPCToolkit's measurement overhead ranges from 1-5% for reasonable sampling rates [33].

The second challenge that petascale systems had posed for measurement was that their compute node microkernels made sampling-based masurement impossible, in part because of a concern initially raised at SuperComputing. At SC '03, Petrini *et al.* showed that for large systems, asynchronous operating system activity, such as periodically monitoring I/O, could cause serious performance problems [13, 23]. As a result, minimizing interrupts to avoid OS "jitter" was a critical concern when designing the Catamount microkernel for the Cray XT3 [3]. As a side effect, it was not possible to use statistical sampling as a measurement approach on Catamount until we interceded with its developers at Sandia. In modern compute node kernels for the Cray XT and Blue Gene/P, the intent of their developers was to provide kernel support for sampling; however, before we exercised this capability with HPCToolkit, this support was non-functional in both kernels. In 2008, we engaged kernel developers at IBM and Cray to address the shortcomings of their implementations and in early 2009, kernel versions with working support for sampling were released and installed on the DOE's leadership computing platforms.

To support sampling-based peformance analysis on emerging petascale platforms, including x64-64-based systems running Linux (*e.g.*, the Ranger system at the University of Texas), x86-64-based Cray XT systems running Compute Node Linux, and PowerPC-based Blue Gene/P systems running IBM's compute node kernel, we added several new capabilities to HPCToolkit. Capabilities developed over the last year include (1) technology for monitoring processes, threads, and dynamic loading; (2) on-the-fly binary analysis to support call path profiling of optimized and partially stripped executables; and (3) support for injecting a monitoring library into a statically-linked executable. While support for statically-linked binaries is needed for Cray XT and Blue Gene/P platforms, support for dynamically-loaded shared libraries is needed for dynamically-linked binaries, which are typical on clusters that run more full-featured Linux kernels, *e.g.*, UT's Ranger.

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

The rest of this paper is organized as follows. Section 2 describes HPCToolkit's approach to measurement and analysis and shows how it enables costs, including scalability bottlenecks, to be attributed to their full static and dynamic contexts. In Section 3, we use HPCToolkit to analyze the scaling of several applications slated for use on petascale systems. Section 4 compares our approach with related work. Section 5 summarizes our conclusions and outlines our ongoing work.

2. MEASUREMENT AND ANALYSIS

In this section we summarize HPCToolkit's measurement and analysis capabilities and its method of pinpointing and quantifying scalability bottlenecks.

2.1 Call path profiling

Without accurate measurement, performance analysis results may be of questionable value. As a result, a principal focus of work on HPCToolkit has been the design and implementation of techniques for providing accurate fine-grain measurements of production applications running at scale. For tools to be useful on production applications on large-scale parallel systems, large measurement overhead is unacceptable. For measurements to be accurate, performance tools must avoid introducing measurement error. Both source-level and binary instrumentation can distort application performance through a variety of mechanisms [21]. In addition, source-level instrumentation can distort application performance by interfering with inlining and template optimization. To avoid these effects, tools such as TAU intentionally refrain from instrumenting certain procedures [25]. Ironically, the more this approach reduces overhead, the more it introduces *blind spots*, *i.e.*, portions of unmonitored execution. For example, a common selective instrumentation technique is to ignore small frequently executed procedures — but these may be just the thread synchronization library routines that are critical. Sometimes, a tool unintentionally introduces a blind spot. A typical ex-

ample is that source code instrumentation necessarily introduces blind spots when source code is unavailable, a common condition for math and communication libraries.

To avoid these problems, HPCToolkit eschews instrumentation and favors the use of *statistical sampling* to measure and attribute performance metrics. During a program execution, sample events are triggered by periodic interrupts induced by an interval timer or overflow of hardware performance counters. One can sample metrics that reflect work (*e.g.*, instructions, floating-point operations), consumption of resources (*e.g.*, cycles, memory bus transactions), or inefficiency (*e.g.*, stall cycles). For reasonable sampling frequencies, the overhead and distortion introduced by sampling-based measurement is typically much lower than that introduced by instrumentation [10].

For all but the most trivially structured programs, it is important to associate the costs incurred by each procedure with the contexts in which the procedure is called. Knowing the context in which each cost is incurred is essential for understanding why the code performs as it does. This is particularly important for code based on application frameworks and libraries. For instance, costs 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 how they are used in a particular 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 uses call path profiling to attribute costs to the full calling contexts in which they are incurred.

HPCToolkit's `hpcrun` call path profiler uses call stack unwinding to attribute execution costs of optimized executables to the full calling context in which they occur. Unlike other tools, to support asynchronous call stack unwinding during execution of optimized code, `hpcrun` uses on-line binary analysis to locate procedure bounds and compute an unwind recipe for each code range within each procedure [33]. These analyses enable `hpcrun` to unwind call stacks for optimized code with little or no information other than an application's machine code.

2.2 Recovering static program structure

To enable effective analysis, call path profiles for executions of optimized programs must be correlated with important source code abstractions. Since measurements refer only to instruction addresses within an executable, it is necessary to map measurements back to the program source. To associate measurement data with the static structure of fully-optimized executables, we need a mapping between object code and its associated source code structure.[2] HPCToolkit constructs this mapping using binary analysis; we call this process "recovering program structure" [33].

HPCToolkit focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, HPCToolkit's `hpcstruct` utility parses a load module's machine instructions, reconstructs a control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and account for loop transformations [33].

Two important 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 scalarized loops that arise from compilation of 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. Second, we combine (post-mortem) the recovered static program structure with dynamic call paths to expose inlined frames and loop nests. This enables us to attribute the performance of samples in their full static and dynamic context and correlate it with source code.

2.3 Pinpointing scaling losses

To pinpoint and quantify scalability bottlenecks *in context*, we compute a metric that quantifyies scaling loss by scaling and differencing call path profiles from a pair of executions [7].

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

Using HPCToolkit, we collect call path profiles on each of p and q processors to measure the cost associated with each calling context in each execution. HPCToolkit's `hpcrun` profiler uses a data structure called a *calling context tree* (CCT) to record a call path profile. Each node in a CCT is identified by a code address. In a CCT, the path from any node to the root represents a calling context. Each node has a weight $w \geq 0$ indicating the exclusive cost attributed to the path from that node to the root. Given a pair of CCTs, one collected on p processors and another collected on q processors, with perfect weak scaling, the cost attributed to all pairs of corresponding CCT nodes[3] should be identical. Any additional cost for a CCT node on q processors when compared to its counterpart in a CCT for an an execution on p processors represents *excess work*. This process is shown pictorially in Figure 1. The fraction of excess work, *i.e.*, the amount of excess work in a calling context in a q process execution divided by the total amount of work in a q process execution represents the scalability loss attributed to that calling context. By scaling the costs attributed in a CCT before differencing them to compute excess work, one can also use this strategy to pinpoint and quantify strong scalability losses [7]. As long as the CCT's are expected to be similar, this analysis strategy is independent of the programming model and bottleneck cause.

Above, we described applying our scalability analysis technique *across* nodes in a cluster. This technique can also be used to pinpoint scaling bottlenecks within multicore nodes. For instance, one might want to understand how perfor-

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

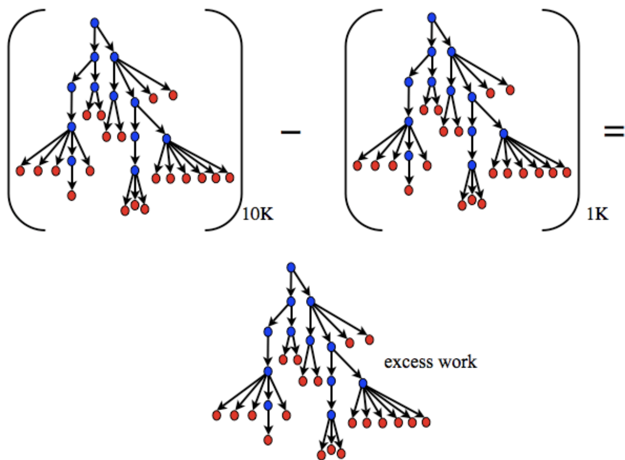

Figure 1: A pictorial representation of differencing call path profiles to pinpoint (weak) scaling bottlenecks.

mance scales when using all of the cores in a node with multicore processors instead of just a single core. This can be accomplished by measuring an execution on a single core, measuring an execution on all cores, and then comparing the costs incurred by a core in each of the executions using the strategy described above for analysis of weak scaling. We have used this strategy to pinpoint and quantify scaling bottlenecks on multicore nodes at the loop level [31]. Measurements of L2 cache misses showed that contention in the memory hierarchy was the problem.

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

3. APPLICATION STUDIES

To demonstrate the utility of HPCToolkit for performance analysis of applications on emerging petascale applications, we apply it to study the performance of three codes: PFLOTRAN, FLASH, and MILC. We studied these applications on core counts up to 8192.[4] Our performance studies were performed on two systems: Jaguar — a Cray XT system at Oak Ridge National Laboratory's National Center for the Computational Sciences — and Intrepid — a Blue Gene/P at Argonne National Laboratory's Leadership Computing Facility. We describe these machines as they exist in spring 2009.

Jaguar consists of 84 Cray XT4 racks and 200 Cray XT5 racks linked together. There are 7,832 XT4 compute nodes and 18,772 XT5 compute nodes for a total of 181,504 cores. Each XT4 node contains a quad-core 2.1 GHz Opteron (Budapest), 8 GB memory and a SeaStar2 network interface card; each XT5 node contains two quad-core 2.3 GHz Opterons (Barcelona), twice the memory and twice the memory bandwidth, but with one SeaStar2+ interface card.

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

Nodes in the system are arranged in a 3-D torus topology. Compute nodes run Cray's Compute Node Linux (CNL) microkernel. In early February 2009, CNL version 2.1 was installed which corrects bugs that inhibited sampling in prior versions.

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

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

3.1 PFLOTRAN

PFLOTRAN is a code for modeling multi-phase, multi-component subsurface flow and reactive transport using massively parallel computers [16, 20]. The code is designed to predict the migration of contaminants underground. "PFLOTRAN solves a coupled system of mass and energy conservation equations for multiple compounds and phases including H_2O, supercritical CO_2, black oil, and a gaseous phase" [20]. With support from the DOE SciDAC program, the authors of PFLOTRAN plan to use it to understand radionuclide migration at the DOE Hanford facility and model sequestration of CO_2 in deep geologic formations. Typical simulations involve massive computation due to ten or more chemical degrees of freedom on a grid of millions of nodes. PFLOTRAN employs the PETSc library's Newton-Krylov solver framework.

In this study, we use HPCToolkit to examine study the performance of PFLOTRAN when strong scaling from 512 to 8192 cores of a Cray XT4. (A strong scaling study employs different numbers of cores on the same test problem.) The test problem used for this study is a steady-state groundwater flow problem in heterogeneous porous media on a 512^3 element discretization. It uses PETSc's IBCGS (Improved Stabilized version of BiConjugate Gradient Squared) solver [24, 38] to solve for flow.

Figure 2 shows a screen snapshot from HPCToolkit's `hpcviewer` user interface displaying a top-down *calling context view* of how PFLOTRAN spends its time on 512 processors. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, partially expanded. One can see several procedure instances along the call paths in the calling context tree. Each entry in the navigation pane is associated with metric values in the metric pane to its right. The line selected in the navigation pane is displayed in the source pane (top sub-pane). For the steady state flow problem measured, on 512 processors the selected line shows that PFLOTRAN spends 98% of its time (measured as *inclusive* processor cycles using the PAPI [5] interface to hardware counters) inside PETsc's `SNESolve` procedure, called from

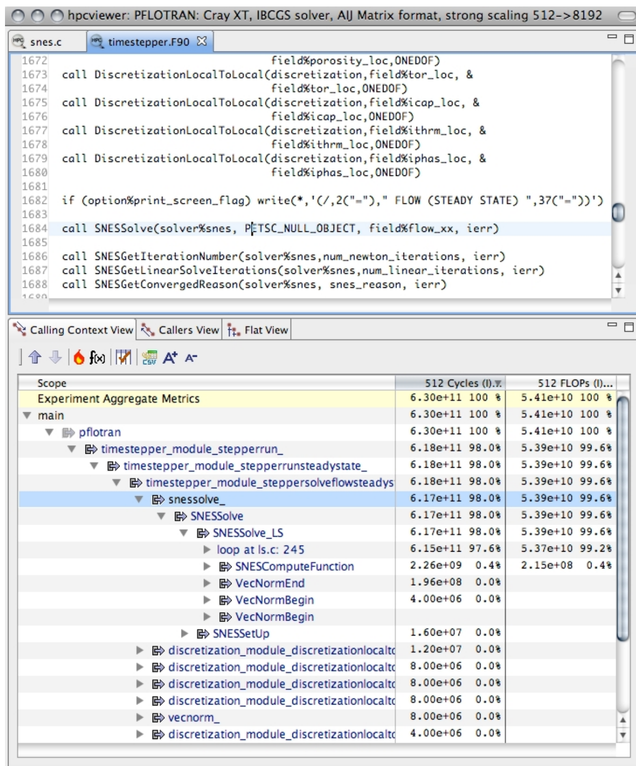

Figure 2: A calling context tree view of costs for PFLOTRAN on a Cray XT4.

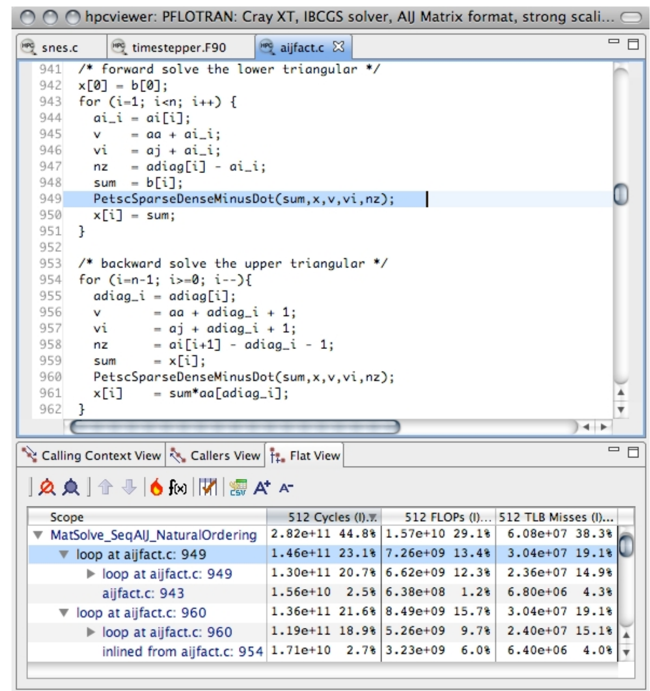

Figure 3: A flat view of costs for PFLOTRAN on a Cray XT4.

PFLOTRAN's `StepperSolveFlowSteadyState` procedure in module `Timestepper_module`. Comparing the cycles spent in `SNESolve` with the floating point operations performed (shown in the rightmost column), we see that the solver executes only one floating point operation about every 11 cycles. This low performance bears further investigation.

Figure 3 shows a flat view of the most costly procedure, PETSc's `MatSolve_SeqAIJ_NaturalOrdering`, where the 512 processor execution of PFLOTRAN spent 44.8% of the total execution time when executing the steady state flow problem. A strength of HPCToolkit is that it attributes costs not only at the routine level, but at the loop level too. The second line of the metric pane shows the most costly loop in the aforementioned routine: a forward solve of a lower triangular matrix, which accounts for 23.1% of the total cycles during execution. Almost all of the loop's costs are attributed to line 949 of file `aijfact.c` since the PGI compiler only associates one source line number with each basic block. By comparing the cycles with the second column, floating point operations, we see that the loop executes only about one floating point operation every 20 cycles. The fact that we can pinpoint and quantify the nature of this performance loss shows off HPCToolkit's abilities for locating node performance bottlenecks.

In the loop highlighted in Figure 3, L2 misses (elided in the figure) are lower than average: the loop accounts for only 8.0% of the L2 misses even though it accounts for 23.1% of the cycles. Execution time for the loop correlates more closely with TLB misses: 19.1% of TLB misses and 23.1% of the program cycles. Comparing the number of TLB misses to the number of floating point operations shows that there is a TLB miss for every 239 floating point operations. These measurements suggest that the performance on the Opteron architecture might be improved by reducing TLB misses. To reduce the TLB miss rate, we tried using 2MB jumbo pages; however, we found that this change had little effect on overall runtime. This suggests that we should use other hardware counters to further investigate the reason for the low performance.

Figure 4 shows a bottom-up caller's view of the losses when scaling from solving the test problem on 512 cores to 8192 cores (strong scaling). The caller's view apportions the cost of a procedure (in context) to each call site in each of its callers. For inclusive costs (as shown in this figure), `hpcviewer`'s bottom-up view attributes costs incurred within. For each calling context c in the program executions in this scaling study, we compute the percent of scaling losses as $100(16\ \mathrm{T}_{c,8192} - \mathrm{T}_{c,512})/(16\ \mathrm{T}_{r,8192})$, where r is the root of the calling context tree, and $T_{i,n}$ represents the time spent in context i in an n core execution. In English, the quantity $(16\ \mathrm{T}_{c,8192} - \mathrm{T}_{c,512})$ calculates the difference in parallel work performed by the executions on 512 and 8192 cores for a particular calling context c. The factor of 16 arises because when strong scaling from 512 to 8192 processors, the amount of work per processor is a factor of 16 smaller on the larger number of processors. We divide through by 16 $\mathrm{T}_{r,8192}$, the total amount of work performed on 8192 cores, to compute the relative fraction of the execution that corresponds to parallel overhead. We multiply through by 100 to express this number in percent. In Figure 4, the percent relative scaling loss in the 8192-core execution is represented using scientific notation. The percentages shown in that column show the percentage of the total scaling loss that is associated with each line in the display.

Figure 4 shows that 112.2% of the scaling loss in the application is attributed to the routine `MPIDI_CRAY_Progress_`

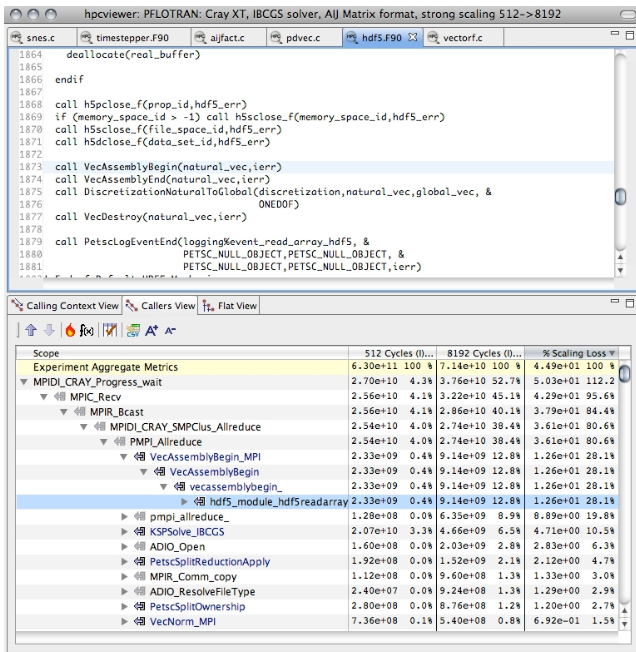

Figure 4: A caller's view of scaling losses for PFLOTRAN on a Cray XT4.

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

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

3.2 FLASH

Next we consider FLASH [9], a code for modeling astrophysical thermonuclear flashes. We performed a weak scaling study of a white dwarf explosion by executing 256-core and 8192-core simulations on both Jaguar (Cray XT) and Intrepid (IBM BlueGene/P). Both the input and the number of cores are 32x larger for the 8192-core execution. With perfect scaling, we would expect identical run times and call path profiles for both configurations.

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

IBM BG/P.

To quickly understand where the scaling losses for the initialization and simulation phases are aggregated, we turn to the bottom-up callers view. Recall that the caller's view apportions the cost of a procedure (in context) to its callers. We sort the callers view by the exclusive scaling loss metric, thus highlighting the scaling loss for each procedure in the application, exclusive of callees. Two routines in the BG/P communication library immediately emerge as responsible for the bulk of the scaling loss: `TreeAllreduce::advance` and `globalBarrierQueryDone`.[5] To determine how these library calls relate to user-level code, we look up their call chains; the result is shown in Figure 5. When we look up the first call chain, we find calls to `MPI_Allreduce`. The first call, which accounts for 57% of the scaling loss (14.1% of run time), is highlighted in blue; the others, which are inconsequential, are hidden by an image overlay indicated by the thick horizontal black line. As the corresponding source code shows, this call to `MPI_Allreduce` is a global max reduce for a scalar that occurs in code managing the adaptive mesh. HPCToolkit is uniquely able to pinpoint this one crucial call to `MPI_Allreduce` and distinguish it from several others that occur in the application.

Next, we peer up the `globalBarrierQueryDone` call chain. The "Hot path" button automatically expands the unambiguous portion of the hot path. By expanding this hot path automatically, we hone in on the one call to `MPI_Barrier` that disproportionately affects scaling. The call site is within `Grid_fillGuardCells` and is visible at the bottom of Figure 5; it accounts for 13.6% of the scaling loss (or 3.31% of the run time).

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

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

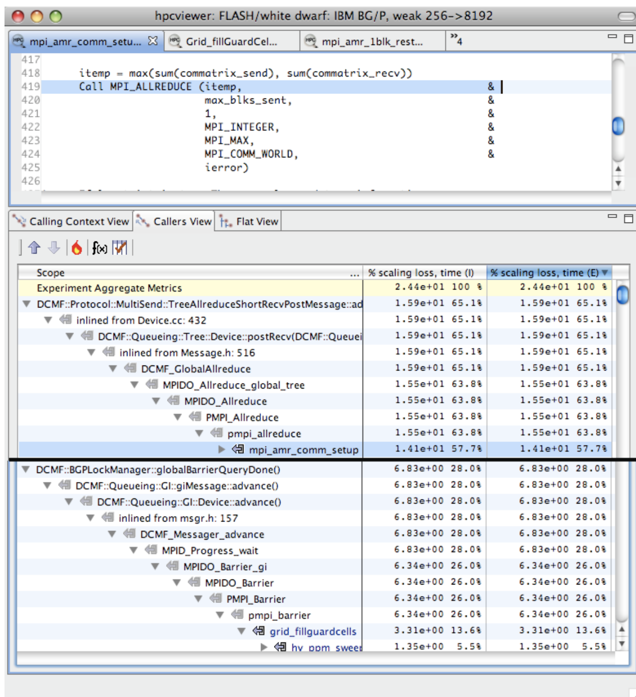

Figure 5: A Caller's (bottom-up) view of scaling loss (wallclock) for FLASH on an IBM BG/P.

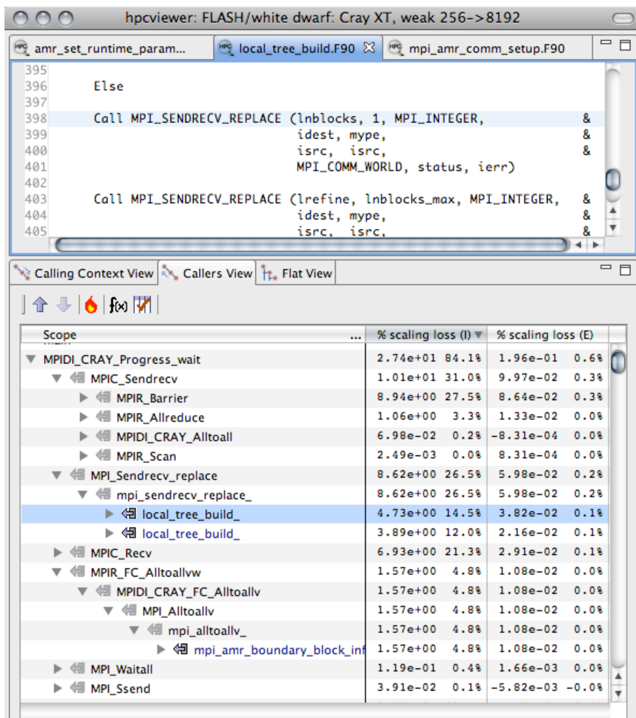

Figure 6: A Callers (bottom-up) view of scaling loss (cycles) for FLASH on a Cray XT4.

Cray XT.

In Figure 6, we turn to the same bottom-up callers view that we used to analyze scaling losses on BG/P. We first sort by the exclusive scaling loss metric. However, because losses are more finely distributed than on BG/P, we sort by inclusive losses, which includes losses for callees. Since 100% of the scaling loss occurs in or below FLASH's "main" routine, it appears at the top. The next procedure, `MPIDI_CRAY_Progress_Wait`, accounts for 84.1% of the scaling loss, is related to MPI communication, is shown in Figure 6. By inspecting the callers of this procedure, we see the breakdown of scaling losses among different types of communication. When using the `hpcviewer` interface interactively, one can expand the tree further to show the full context in the user program where these losses originate.

By inspecting the callers of `MPIC_Sendrecv`, one can see that 27.5% of the losses are due to barrier synchronization. Exploring a few levels deeper in the sub-tree rooted at `MPIR_Barrier`, we find that 12.1% of the scaling losses are due to barrier synchronization in the routine `amr_setup_runtime_parameters`. This routine contains a loop that iterates over each of the processor IDs. On each iteration of the loop, the processor whose ID is equal to the loop induction variable opens the input file, reads a set of program input parameters, and then closes the file. All processors meet at the bottom of the loop at a barrier. This represents a scaling bottleneck whose severity increases with the number of processors. Fortunately, it has a remedy: one processor can open the input file and broadcast its contents to the rest of the processors; this change transforms the operation from $O(p)$ time to $O(\log p)$ time. Implementing and testing this solution on the Cray XT reduced the scaling loss due to `amr_setup_runtime_parameters` on 8192 cores to almost zero.

The highlighted line in Figure 6 shows one of two call sites for `local_tree_build`. This routine is part of the PARAMESH library [17] used by Flash. Together, the function's two call sites account for 26.5% of the scaling losses and 8.62% of execution time on 8192 processors. This function builds an oct-tree as part of the structured adaptive mesh refinement. It scales poorly as the number of processors is increased. `local_tree_build` uses a communication pattern known as a *digital orrery*, in which all-to-all communication is implemented by circulating content from each processor around a ring of all processors. The communication phase takes $O(p)$ time. By consulting the calling context view (not shown) we found that `local_tree_build` is called both within FLASH's initialization and simulation phases. In the initialization phase it accounts for 18.5% of the scaling loss; in simulation it accounts for about 7.9%. We have had preliminary discussions with the FLASH team about how to improve the scaling of `local_tree_build`.

Figure 6 shows that 21.3% of the scaling loss results from `MPI_Recv`. Expanding the sub-tree rooted at that point, one discovers that almost all of these costs are due to calls to `MPI_AllReduce`. 15.5% of the total scaling loss is for `MPI_AllReduce` calls that are used to exchange information about blocks to set up communication prior to guard cell filling and flux conservation. In contrast, the same max reduction on BG/P accounts for 40.6% of the scaling loss.

Summary.

In the span of minutes, we have used HPCToolkit to pinpoint and quantify the scaling losses in each system deriving from just a few crucial call sites. HPCToolkit enables us to focus on the key areas and ignore the other losses, which are more finely distributed. Moreover, HPCToolkit

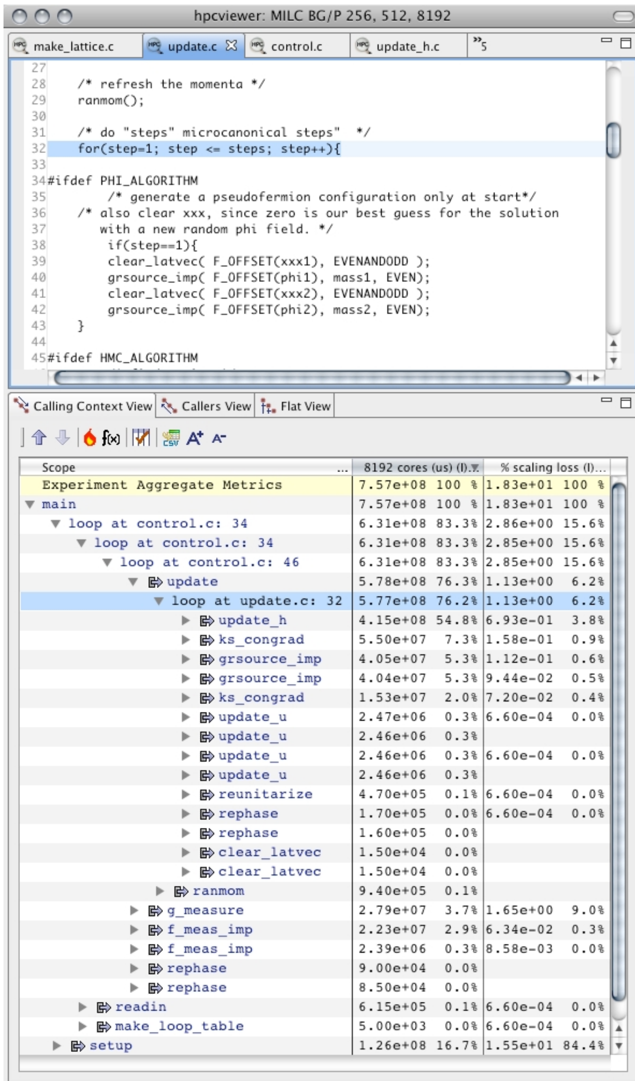

Figure 7: A calling context view of scaling (cycles) loss for MILC on a BG/P.

obtains accurate call paths and precise measurements despite several layers of communication library calls for which no source code is available to application developers. The static program structure information computed by HPCToolkit even reports inlining within these layers.

3.3 MILC

The third application we analyze is a lattice quantum chronodynamics (QCD) simultation with dynamical Kogut-Susskind fermions from MILC, or MIMD Lattice Computation package [4]. MILC is a Lattice Quantum Chromodynamics code that is one of six application benchmarks in a suite used to evaluate bids for an NSF-funded petascale computer. We performed a weak scaling study by profiling 512-core and 8192-core simulations on both Jaguar (Cray XT) and Intrepid (IBM Blue Gene/P). To keep execution time for the scaling study reasonable, we altered the default NSF problem size by decreasing the number of trajectories. In our scaling study, the input data and the number of cores are scaled by a factor of 16 so if scaling is ideal we should expect identical run times and call path profiles for both core counts.

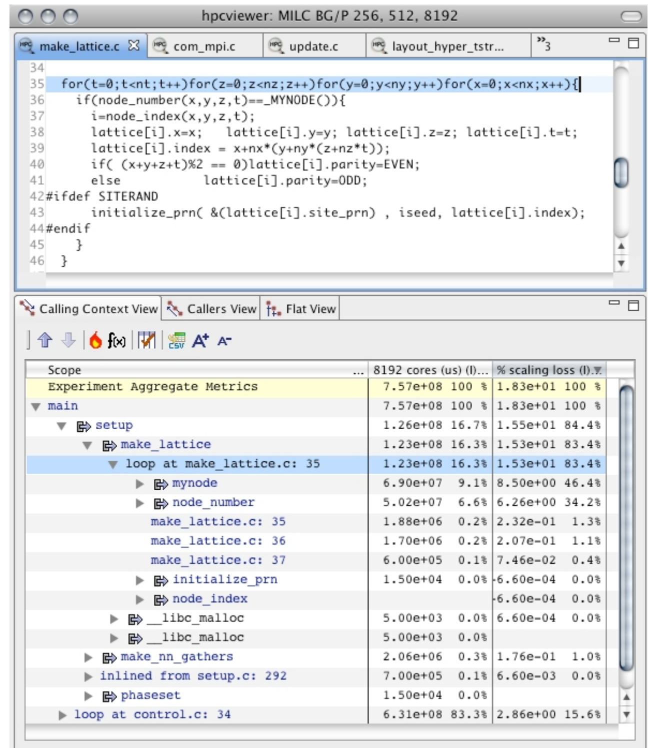

Figure 8: A closer look at scaling losses for MILC on a BG/P.

Figures 7 and 8 respectively focus on the breakdown of execution time and scaling losses (relative to a 512-core execution) for MILC in an 8192-core execution on a BG/P. The most time-consuming part of the code is the lattice update. In Figure 7, we can see that this phase accounts for 76.3% of the time on BG/P in an 8192-core execution; in an execution on a Cray XT, this phase accounted for 83.3% of the execution time. Within the update phase, execution time is distributed among routines called from the loop on line 32 in `update` and routines they call.

The total inclusive scaling loss for the application is shown in the yellow highlighted line as a percentage written in scientific notation. As shown in both figures, MILC has 18.3% total scaling loss on a BG/P. The lattice update phase scales relatively well and only has a 6.2% scaling loss. Most of the scaling losses in the update phase are due to waiting for scatter-gatter communication to complete. For the short execution studied, Figure 8 shows that MILC's `setup` phase accounts for most of the scaling losses.

In Figure 8, the highlighted loop on line 35 in `make_lattice` accounts for 83.4% of the scaling loss and 16.3% of the run time. The reason that this loop causes a scaling loss is that it initializes local data for an MPI process by having each processor iterate over the *entire* lattice (all possible x, y, z, and t values), test each lattice point to see if it belongs to the current process, and then perform initialization only when the test succeeds. To avoid this kind of scaling loss, the application would need to be reworked to iterate only over a process's local lattice points rather than over the entire domain. Without a deeper understanding of

the application, it is unclear whether this is feasible. Furthermore, it is not clear that losses due to initialization will be significant for production executions. The point of this example is not to focus on a shortcoming of the MILC code; rather, it is to show that HPCToolkit is capable of pinpointing and quantifying losses of this nature. Scaling losses need not be caused by communication.

4. RELATED WORK

Most studies of application scaling on petascale systems have relied on manual analysis rather than sophisticated performance tools to understand scalability [1–3, 13]. Usually the analysis consists of 1) measuring key system performance characteristics using micro-benchmarks; 2) isolating scaling bottlenecks by creating scaling curves for different phases or procedures within the application; and 3) determining causes of bottlenecks by comparing an application's expected performance with its actual performance. Oliker *et al.* performed an early and insightful evaluation of application scaling on candidate petascale systems [22]. Even though they invested considerable effort in manual analysis, they had difficulty pinpointing and quantifying bottlenecks, and were only able to offer educated guesses such as "[the scalability loss] is probably due to the increase in [Allreduce operations]." HPCToolkit could could directly pinpoint which operations were problematic and quantify the scaling loss for each. While the focus of these prior studies was to characterize system performance rather than advocate a method for pinpointing scaling bottlenecks, it was still necessary to understand such bottlenecks as part of their work.

Current performance tools for petascale systems identify scaling bottlenecks at the procedure level at best. The most important reason for this is that it is not feasible to make fine-grained measurements using instrumentation. Moreover, most of these tools require additional effort to analyze scaling. For example, Wright *et al.* used IPM [27] to distinguish between scaling bottlenecks in the communication or computation portions of an application [36]. To achieve low overhead (<5%), they collected profiles of instrumented MPI routines. These coarse measurements — only at the (MPI) procedure level, and without calling context — resulted in two deficiencies. First, because the application's computational component was not directly measured, the authors had to manually correct for communication-computation overlap to understand computational scaling. Second, to achieve further insight, the authors supplemented the measurements with labor-intensive analytical analysis.

mpiP [34] synchronously monitors MPI routines and collects a stack trace for each call. It qualitatively evaluates MPI scaling problems by using a rank-based correlation strategy. Because of this selective instrumentation, it incurs low overhead. However, it misses scaling problems in computational and non-MPI code.

Although other tools measure more comprehensively than IPM and mpiP, their measurements are still relatively coarse, typically at the procedure level. For example, tools such as TAU [18, 26], SCALASCA [35, 37], Cray's CrayPAT [8] and IBM's HPC Toolkit [14] collect the calling context of procedures rather than of statements. Because these tools collect calling context information using procedure-level instrumentation, their measurements are subject to distortion from measurement overhead associated with small procedures. By using sampling, HPCToolkit is able to attribute costs to their full static and dynamic context with only about 1-5% overhead [33], which in most cases is significantly less than procedure-level instrumentation [10]. HPCToolkit has the ability to collect the full calling context of any sample point, even exposing layers of calls in communication and math libraries for which source code is unavailable.

HPCToolkit's approach to computing scalability losses is similar to differential profiling support in other systems, *e.g.* [28]. However HPCToolkit is unique in its capability to attribute scalability losses to their full calling context, including inlined functions, loops and even individual statements. Furthermore, by providing top-down, bottom-up, and flat views of scalability losses in context, HPCToolkit offers several different ways of analyzing the data. Different views provide different perspectives on bottlenecks that can make them easier to understand.

Although application traces can be very valuable (*e.g.*, for identifying load imbalance), the volume of trace information makes scaling difficult. SCALASCA [37] selectively traces based on information from a prior profile. Others have explored (manual) selective tracing based on application characteristics [6]. Gamblin *et al.* developed a technique for compressing trace information on-the-fly [11]. They report impressively low overheads, but by using selective instrumentation that results in coarse measurements.

The STAT tool has been used on BG/L to sample call paths to aid parallel debugging at scale [15]. This tool uses third-party sampling mechanism that relies on daemons, running on I/O nodes, to periodically collect trace samples. In contrast, we use first-party sampling (in which the application samples itself), which requires no communication and permits much higher sampling rates.

5. CONCLUSIONS AND FUTURE WORK

The key metric for parallel performance is scalability, either weak or strong. This is especially true at the petascale. Consequently, there is an acute need for application scientists to understand and address scaling bottlenecks in codes targeted for petascale systems. We have shown that it is possible, for minimal overhead, to pinpoint and quantify scaling bottlenecks on petascale systems to source code lines, in their full static and dynamic context using HPCToolkit. The analysis is rapid and its results are actionable.

Our work depends upon accurate and precise sampling-based measurement — a form of measurement that until now has been unavailable on petascale systems. This measurement both grounds and enables our powerful and elegant method for rapidly pinpointing and quantifying scaling bottlenecks. Past scaling analyses are either laborious, inaccurate (with respect to measurement) or imprecise (with respect to bottleneck detection).

It is a truism that a microkernel for a petascale platform should include what is necessary but dispense with excess: "just enough, but not too much"! The difficulty is in deciding what actually is necessary. We believe our results provide strong evidence that sampling-based performance analysis is so useful on these systems, that future microkernels for large-scale parallel systems should find a way to support it. Because petascale systems are designed for performance, it makes little sense to invest in computing resources that are powerful on paper but that cannot be exploited in practice.

HPCToolkit's support for sample-based performance

analysis can provide insight into scalability and performance problems both within and across nodes. Gaining insight into node performance bottlenecks on large-scale parallel systems is a problem of growing importance. Today, parallel systems typically have between 4-16 cores per node. In emerging systems, we expect the core count per node to be higher. By sampling on hardware performance counters, one can distinguish between node performance bottlenecks caused by a variety of factors including inadequate instruction-level parallelism, memory latency, memory bandwidth, and contention.

Ongoing work in the HPCToolkit project spans a range of topics. First, we have been developing methods to gain higher-level insight into the performance of multithreaded node programs including analysis of performance losses due to load imbalance in computations based on work stealing [32] and serialization due to lock contention. Second, we are working to combine information from profiles gathered on each node to produce summary statistics that will provide a means for assessing similarities and differences in performance across the nodes in a parallel system. For machines at the scale of the leadership computing platforms, it will be necessary to perform this analysis in parallel. We aim to rework our tools to provide not only summary statistics for overall system performance, but also to preserve the ability to drill down into the details of performance on individual nodes. To work with measurement data from large-scale parallel systems, our presentation tools will need to manage data out-of-core. Finally, we are developing a new user interface to visualize how execution behavior unfolds over time. While there are similarities between this work and tools that use space-time diagrams to analyze the performance of MPI programs, a fundamental difference is that our data is based on traces of call stack samples rather than traces of instrumented operations. Our traces are compact (just 12 bytes per entry) and by carefully choosing the sampling frequency, we can reduce the data rate for traces to a level that will make it possible to use this strategy on very large scale executions.

6. ACKNOWLEDGMENTS

We are grateful to Anshu Dubey and Chris Daley of the FLASH team and to Peter Lichtner, Glenn Hammond and other members of the PFLOTRAN team. Both teams graciously provided us with a copy of their codes to study, configuration advice for their code, and a test problem of interest.

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

7. REFERENCES

[1] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of IBM BlueGene/P. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[2] S. R. Alam, R. F. Barrett, M. R. Fahey, J. A. Kuehn, O. B. Messer, R. T. Mills, P. C. Roth, J. S. Vetter, and P. H. Worley. An evaluation of the Oak Ridge National Laboratory Cray XT3. *International Journal of High Performance Computing Applications*, 22(1):52–80, 2008.

[3] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley. Cray XT4: An early evaluation for petascale scientific simulation. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[4] C. Bernard, T. Burch, C. DeTar, S. Gottlieb, E. Gregory, U. Heller, J. Osborn, R. Sugar, and D. Toussaint. QCD thermodynamics with three flavors of improved staggered quarks. *Phys. Rev.*, D71:034504, 2005.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[6] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu. MPI performance analysis tools on Blue Gene/L. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, page 123, New York, NY, USA, 2006. ACM.

[7] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *ICS '07: Proc. of the 21st Annual International Conference on Supercomputing*, pages 13–22, NY, NY, USA, 2007. ACM.

[8] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[9] A. Dubey, L. B. Reid, and R. Fisher. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta*, 132:014046, 2008.

[10] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th Annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[11] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software—Practice & Experience*, 13(8):671–685, August 1983.

[13] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A performance comparison through benchmarking and modeling of three leading supercomputers: Blue Gene/L, Red Storm, and Purple. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, page 74, New York, NY, USA, 2006. ACM.

[14] IBM. IBM system Blue Gene solution: Performance analysis tools. Redpaper REDP-4256-01, November

2008.

[15] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[16] P. Lichtner *et al.* PFLOTRAN project web site. `https://software.lanl.gov/pflotran`.

[17] P. MacNeice, K. M. Olson, C. Mobarry, R. deFainchtein, and C. Packer. PARAMESH : A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126:330–354, 2000.

[18] A. D. Malony, S. Shende, A. Morris, S. Biersdorff, W. Spear, K. Huck, and A. Nataraj. Evolution of a parallel performance system. In *Tools for High Performance Computing*, pages 169–190. Springer Berlin Heidelberg, 2008.

[19] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[20] R. T. Mills, C. Lu, P. C. Lichtner, and G. E. Hammond. Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN. *Journal of Physics Conference Series*, 78(012051), 2007. `doi:10.1088/1742-6596/78/1/012051`.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[22] L. Oliker, A. Canning, J. Carter, C. Iancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific application performance on candidate petascale platforms. *International Parallel and Distributed Processing Symposium*, 0:69, 2007.

[23] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[24] PETSc Team. Kspibcgs. in *petsc: portable, extensible toolkit for scientific computation.* `http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manualpages/KSP/KSPIBCGS.html`.

[25] S. Shende, A. Malony, and A. Morris. *Optimization of Instrumentation in Parallel Performance Evaluation Tools*, volume 4699 of *LNCS*, pages 440–449. Springer, 2008.

[26] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[27] D. Skinner *et al*. IPM: Integrated performance monitoring. `http://ipm-hpc.sourceforge.net/`.

[28] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. of the 2004 International Conference on Parallel Processing*, pages 63–72, Washington, DC, USA, 2004. IEEE Computer Society.

[29] SPEC Corporation. SPEC CPU2006 benchmark suite. `http://www.spec.org/cpu2006`. 3 November 2007.

[30] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`. 29 April 2005.

[31] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPCToolkit: Performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088 (5pp), 2008.

[32] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[33] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[34] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.

[35] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing*, pages 157–167. Springer Berlin Heidelberg, 2008.

[36] N. J. Wright, W. Pfeiffer, and A. Snavely. Characterizing parallel scaling of scientific applications using IPM. In *Proc. of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[37] B. J. N. Wylie, M. Geimer, and F. Wolf. Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Sci. Program.*, 16(2-3):167–181, 2008.

[38] L. Yang and R. Brent. The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures. In *Proc. of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 324–328, 2002.