CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

HPCTOOLKIT: Tools for performance analysis of optimized parallel programs[‡]

L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N. R. Tallent[*,†]

Rice University, Dept. of Computer Science
P.O. Box 1892, Houston, TX 77251-1892, USA

SUMMARY

HPCTOOLKIT is an integrated suite of tools that supports measurement, analysis, attribution, and presentation of application performance for both sequential and parallel programs. HPCTOOLKIT can pinpoint and quantify scalability bottlenecks in fully-optimized parallel programs with a measurement overhead of only a few percent. Recently, new capabilities were added to HPCTOOLKIT for collecting call path profiles for fully-optimized codes without any compiler support, pinpointing and quantifying bottlenecks in multithreaded programs, exploring performance information and source code using a new user interface, and displaying hierarchical space-time diagrams based on traces of asynchronous call stack samples. This paper provides an overview of HPCTOOLKIT and illustrates its utility for performance analysis of parallel applications.

KEY WORDS: performance tools; call path profiling; tracing; binary analysis; execution monitoring

1 INTRODUCTION

High performance computers have become enormously complex. Today, the largest systems consist of tens of thousands of nodes. Nodes themselves are equipped with one or more multi-core microprocessors. Often these processor cores support additional levels of parallelism, such as short vector operations and pipelined execution of multiple instructions. Microprocessor-based nodes rely on deep multi-level memory hierarchies for managing latency and improving data bandwidth to processor cores. Subsystems for interprocessor communication and parallel I/O add to the overall complexity of these platforms. Recently, accelerators such as graphics chips and other co-processors have started to become more common on nodes. As the complexity of HPC systems has grown, the complexity of

[*]Correspondence to: Rice University, Dept. of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, USA
[†]E-mail: tallent@rice.edu
[‡]WWW: `http://hpctoolkit.org`

Contract/grant sponsor: Department of Energy's Office of Science; contract/grant number: DE-FC02-07ER25800 and DE-FC02-06ER25762

Received 22 Aug 2008
Revised 22 Aug 2008

Copyright © 2008 John Wiley & Sons, Ltd.

applications has grown as well. Multi-scale and multi-physics applications are increasingly common, as are coupled applications. As always, achieving top performance on leading edge systems is critical. The inability to harness such machines efficiently limits their ability to tackle the largest problems of interest. As a result, there is an urgent need for effective and scalable tools that can pinpoint a variety of performance and scalability bottlenecks in complex applications.

Nearly a decade ago, Rice University began developing a suite of performance tools now known as HPCTOOLKIT. This effort initially began with the objective of building tools that would help guide our own 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. Since HPCTOOLKIT was developed in large part for our own use, our design goals 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 tools 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 presenting the resulting performance data in a top-down fashion to facilitate rapid analysis. Section 2 outlines the methodology that shaped HPCTOOLKIT's development and provides an overview of some of HPCTOOLKIT's key components. Sections 3 and 4 describe HPCTOOLKIT's components in more detail. We use a parallel particle-in-cell simulation of tubulent plasma in a tokamak to illustrate HPCTOOLKIT's capabilities for analyzing the performance of complex scientific applications. Section 7 offers some conclusions and sketches our plans for enhancing HPCTOOLKIT for emerging petascale systems.

2 METHODOLOGY

We have developed a performance analysis methodology, based on a set of complementary principles that, while not novel in themselves, form a coherent synthesis that is greater than the constituent parts. Our approach is *accurate*, because it assiduously avoids systematic measurement error (such as that introduced by instrumentation), and *effective*, because it associates useful performance metrics (such as parallel idleness or memory bandwidth) with important source code abstractions (such as loops) as well as dynamic calling context. The following principles form the basis of our methodology. Although we identified several of these principles in earlier work [19], it is helpful to revisit them as they continually stimulate our ideas for revision and enhancement.

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

Avoid code instrumentation. 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 tools to automatically add instrumentation can be problematic. For instance, using the Tau performance analysis tools to add source-level instrumentation to the Chroma code [15] from the US Lattice Quantum Chromodynamics project [33] required seven hours of recompilation [31]. Although binary instrumentation, such as that performed by Dyninst [13] or Pin [18], avoids this problem, any instrumentation-based measurement approach can be problematic. Adding instrumentation to every procedure can substantially dilate a program's execution time. Experiments with `gprof` [11], a well-known call graph profiler, and the SPEC integer benchmarks showed that on average `gprof` dilates execution time by 82% [10]. Also, we have seen instrumentation added by Intel's *VTune* [14] dilate execution time by as much as a factor of 31 [9]. Adding instrumentation to loops presents an even greater risk of high overhead. Unless analysis of instrumentation-based measurements compensate for measurement overhead, the cost of small routines can appear inflated. To avoid the pitfalls of instrumentation, HPCTOOLKIT uses statistical sampling to measure performance.

Avoid blind spots. Production applications frequently link against fully optimized and even partially stripped binaries, *e.g.*, math and communication libraries, for which source code is not available. To avoid systematic error, one must measure costs for routines in these libraries; for this reason, source code instrumentation is insufficient. However, fully optimized binaries create challenges for call path profiling and hierarchical aggregation of performance measurements (see Sections 3 and 4.1). To deftly handle optimized and stripped binaries, HPCTOOLKIT performs several types of binary analysis.

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 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 upon their calling context. Because there are often layered implementations within applications and libraries, it is insufficient either to insert instrumentation at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCTOOLKIT supports call path profiling to attribute costs to the full calling contexts in which they are incurred.

Any one performance measure produces a myopic view. Measuring time or only one species of event seldom diagnoses a correctable performance problem. One set of metrics may be necessary to identify a problem and another set may be necessary to diagnose its causes. For example, counts of cache misses indicate problems only if both the *miss rate* is high and the latency of the misses is not hidden. HPCTOOLKIT supports collection, correlation and presentation of multiple metrics.

Derived performance metrics are essential for effective analysis. Typical metrics such as elapsed time are useful for identifying program hot spots. However, tuning a program usually requires a measure of not where resources are consumed, but where they are consumed *inefficiently*. For this purpose, derived measures such as the difference between peak and actual performance are far more useful than raw data such as operation counts. HPCTOOLKIT's `hpcviewer` user interface supports computation of user-defined derived metrics and enables users to rank and sort program scopes using such metrics.

Performance analysis should be top down. It is unreasonable to require users to wade through mountains of data to hunt for evidence of important problems. To make analysis of large programs tractable, performance tools should present measurement data in a hierarchical fashion, prioritize what appear to be important problems, and support a top-down analysis methodology that helps users quickly locate bottlenecks 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 metrics.

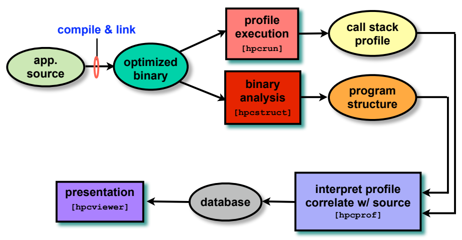
Hierarchical aggregation is vital. The amount of instruction-level parallelism in processor cores can make it difficult or expensive for hardware counters to precisely attribute particular events to specific instructions. However, even if fine-grain attribution of events is flawed, total event counts within loops or procedures will typically be accurate. In most cases, it is the balance of operation counts within loops that matters—for instance, the ratio between floating point arithmetic and memory operations. HPCTOOLKIT's hierarchical attribution and presentation of measurement data deftly addresses this issue; loop level information available with HPCTOOLKIT is particularly useful.

Measurement and analysis must be scalable. Today, large parallel systems may have tens of thousands of nodes, each equipped with one or more multi-core processors. For performance tools to be useful on these systems, measurement and analysis techniques must scale to tens and even hundreds of thousands of threads. HPCTOOLKIT's sampling based measurements are compact and the data for large-scale systems won't be unmanageably large. Furthermore, as we describe later, HPCTOOLKIT supports a novel approach for quantifying and pinpointing scalability bottlenecks conveniently on systems independent of scale.

2.1 From principles to practice

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

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



HPCTOOLKIT workflow.

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

At this level of detail, much of the HPCTOOLKIT workflow approximates other peformance analysis systems, with the most unusual step being binary analysis. In the following sections, we outline

[†]For the most detailed attribution of application performance data using HPCTOOLKIT, one should ensure that the compiler includes line map information in the object code it generates. While HPCTOOLKIT does not need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully-optimized code, this requirement need not require a special build process.

how the methodological principles described above suggest several novel approaches to both accurate measurement (Section 3) and effective analysis (Section 4).

3 ACCURATE PERFORMANCE MEASUREMENT

This section highlights the ways in which we apply the methodological principles from Section 2 to measurement. Without accurate performance measurements for fully optimized applications, analysis is unproductive. Consequently, one of our chief concerns has been designing an accurate measurement approach that simultaneously exposes low-level execution details while avoiding systematic measurement error, either through large overheads or through systematic dilation of execution. For this reason, HPCTOOLKIT avoids instrumentation and favors *statistical sampling*.

Statistical sampling. Statistical sampling uses a recurring event trigger to send signals to the program being profiled. When the event trigger occurs, a signal is sent to the program. A signal handler then records the context where the sample occurred. The recurring nature of the event trigger means that the program counter is sampled many times, resulting in a histogram of program contexts. 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.

Event triggers. Different kinds of event triggers measure different aspects of program performance. Event triggers can be either asynchronous or synchronous. Asynchronous triggers are not initiated by direct program action. HPCTOOLKIT triggers asynchronous samples using either an interval timer or hardware performance counter events. Hardware performance counters enable HPCTOOLKIT to statistically profile events such as cache misses and issue stall cycles. Synchronous triggers, on the other hand, are generated via direct program action. Examples of interesting events for synchronous profiling are memory allocation, I/O, and inter-process communication. For such events, one might measure bytes allocated, written, or communicated, respectively.

Measuring dynamically and statically linked executables. To support measurement of unmodified, dynamically-linked, optimized application binaries, HPCTOOLKIT uses the library preloading feature of modern dynamic loaders to preload a profiling library as an application is launched. For asynchronous triggers, the library's initialization routine allocates and initializes profiler state, configures signal handlers and asynchronous event triggers, and then initiates profiling. The library's finalization routine halts profiling and writes the profile state to disk for post-mortem analysis. Synchronous sampling does not need signal handlers or asynchronous event triggers; instead, dynamic preloading overrides library routines of interest and logs information of interest when the routine is called in addition to performing the requested operation. Because static linking is mandatory on some lightweight operating systems such as Catamount and Compute Node Linux for the Cray XT, we have developed a script that arranges for symbols to be overridden at link time.

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

When synchronous or asynchronous events occur, `hpcrun` records the full calling context for each event. A calling context collected by `hpcrun` is a list of instruction pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program address at which the event occurred. The rest of the list contains the return address for each active procedure frame. Rather than storing the call path independently for each sample event, we represent all of the call paths for events as a calling context tree (CCT) [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. Since the calling context for a sample may be completely represented by a node id in the CCT, to form a trace we simply augment a CCT profile with a sequence of tuples, each consisting of a 32-bit CCT node id and a 64-bit time stamp.

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

To apply our binary analysis to compute unwind recipes, we must know where each routine starts and ends. When working with applications, one often encounters partially-stripped libraries or executables that are missing information about function boundaries. To address this problem, we developed a binary analyzer that infers routine boundaries by noting instructions that are reached by call instructions or instructions following unconditional control transfers (jumps and returns) that are not reachable by conditional control flow.

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

Handling dynamic loading. Modern operating systems such as Linux enable programs to load and unload shared libraries at run time, a process known as *dynamic loading*. Dynamic loading presents the possibility that multiple functions may be mapped to the same address at different times during a program's execution. As `hpcrun` only collects a sequence of one or more program counter values when a sample is taken, post-mortem analysis must map these instruction addresses to the functions that contained them. For this reason, `hpcrun` identifies each sample recorded with an *epoch*, a list of shared objects that can be loaded *without conflict* during one execution phase. `hpcrun` forms epochs lazily, *i.e.*, just before a shared library is loaded into the space formerly occupied by another library. Epochs may also naturally be used to divide phases of a program's execution.

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

4 ANALYSIS

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

4.1 Correlating performance metrics with optimized code

To enable effective analysis, measurements of fully optimized programs must be correlated with important source code abstractions. Since measurements are made with reference to executables and shared libraries, for analysis, it is necessary to map measurements back to the program source. To perform this translation, *i.e.*, to associate sample-based performance measurements with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure.[‡] HPCTOOLKIT's `hpcstruct` constructs this mapping using binary analysis; we call this process "recovering program structure."

`hpcstruct` focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, `hpcstruct` parses a load module's machine instructions, reconstructs a control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and account for transformations to loops [31].[§]

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

4.2 Computed metrics

Identifying performance problems and opportunities for tuning may require synthetic performance metrics. To identify where an algorithm that is not effectively using hardware resources, one should compute a metric that reflects *wasted* rather than consumed resources. For instance, when tuning a floating-point intensive scientific code, it is often less useful to know where the majority of the floating-point operations occur than where floating-point performance is low. Knowing where the most cycles are spent doing things other than floating-point computation hints at opportunities for tuning. Such a metric can be directly computed by taking the difference between the cycle count and FLOP count

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

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

divided by a target FLOPs-per-cycle value, and displaying this measure for loops and procedures. Our experiences with using multiple computed metrics such as miss ratios, instruction balance, and "lost cycles" underscore the power of this approach.

4.3 Analyzing scalability of parallel programs

One novel application of HPCTOOLKIT's call path profiles is to use them to pinpoint and quantify scalability bottlenecks in SPMD parallel programs [5]. Combining call path profiles with program structure information, HPCTOOLKIT can use this *excess work* metric to quantify scalability losses and attribute them to the full calling context in which these losses occur. In addition, we recently developed general techniques for effectively analyzing multithreaded applications [32]. Using them, HPCTOOLKIT can attribute precise measures of parallel work, idleness, and overhead to *user-level* calling contexts—even for multithreaded languages such as Cilk [8], which uses a work-stealing run-time system.

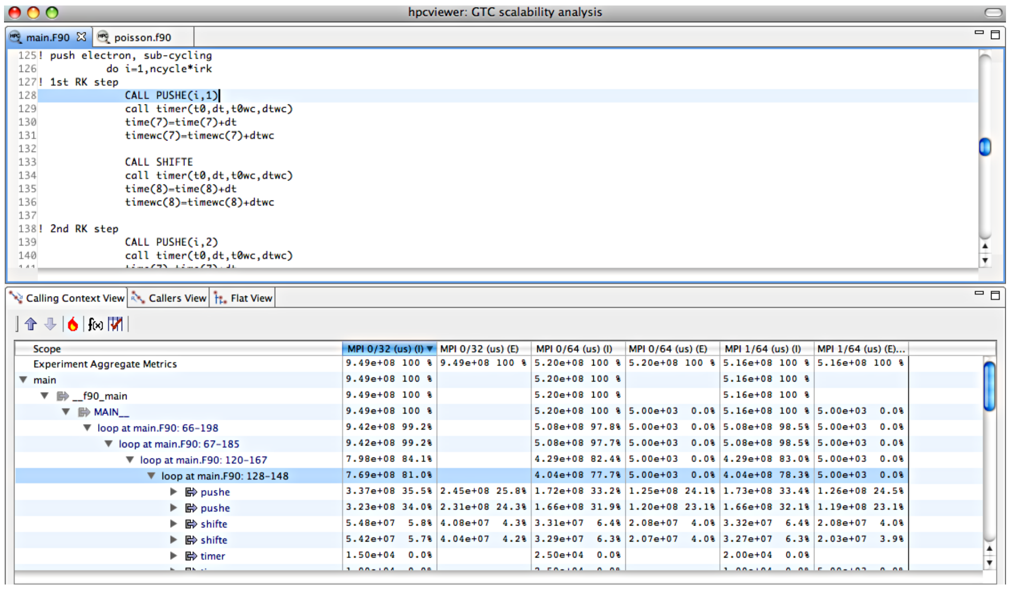
5 PRESENTATION

This section describes `hpcviewer` and `hpctraceview`, HPCTOOLKIT's two presentation tools. We illustrate the functionality of these tools by applying them to measurements of parallel executions of the Gyrokinetic Toroidal Code (GTC) [17]. GTC is a particle-in-cell (PIC) code for simulating turbulent transport in fusion plasma in devices such as the International Thermonuclear Experimental Reactor (ITER). GTC is a production code with 8M processor hours allocated to its executions during 2008. To briefly summarize the nature of GTC's computation, each time step repeatedly executes `charge`, `solve`, and `push` operations. In the `charge` step, it deposits the charge from each particle onto grid points nearby. Next, the `solve` step computes the electrostatic potential and field at each grid point by solving the Poisson equation on the grid. In the `push` step, the force on each particle is computed from the potential at nearby grid points. Particles move according to the forces on them.

5.1 `hpcviewer`

HPCTOOLKIT's `hpcviewer` user interface presents performance metrics correlated to program structure (Section 4.1) and mapped to a program's source code, if available. Figure 1 shows a snapshot of the `hpcviewer` user interface viewing data from several parallel executions of GTC. The user interface is composed of two principal panes. The top pane displays program source code. The bottom pane associates a table of performance metrics with static or dynamic program structure. `hpcviewer` provides three different views of performance measurements collected using call path profiling. We briefly describe the three views and their corresponding purposes.

- **Calling context view**. This top-down view associates an execution's dynamic calling contexts with their costs. Using this view, one can readily see how much of the application's cost was incurred by a function when called from a particular context. If finer detail is of interest, one can explore how the costs incurred by a call in a particular context are divided between the callee itself and the procedures it calls. HPCTOOLKIT distinguishes calling context precisely by individual call sites; this means that if a procedure g contains calls to procedure f in different places, each call represents a separate calling context. Figure 1 shows a calling context view. This view is created by integrating static program structure (*e.g.*, loops) with dynamic calling contexts gathered by `hpcrun`. Loops appear explicitly in the call chains shown in Figure 1.



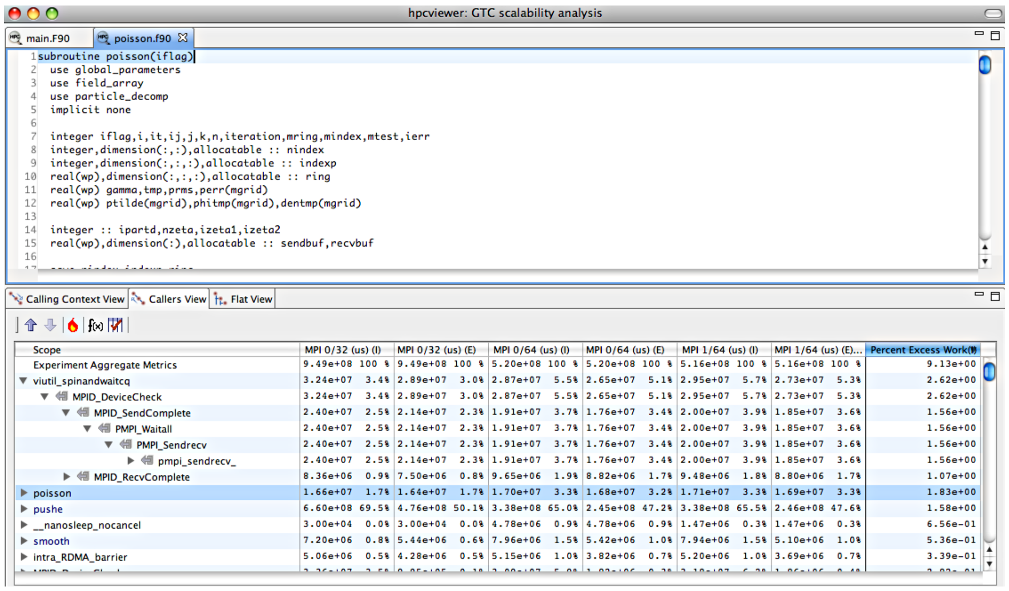
```
! push electron, sub-cycling
           do i=1,ncycle*irk
! 1st RK step
              CALL PUSHE(i,1)
              call timer(t0,dt,t0wc,dtwc)
              time(7)=time(7)+dt
              timewc(7)=timewc(7)+dtwc

              CALL SHIFTE
              call timer(t0,dt,t0wc,dtwc)
              time(8)=time(8)+dt
              timewc(8)=timewc(8)+dtwc

! 2nd RK step
              CALL PUSHE(i,2)
              call timer(t0,dt,t0wc,dtwc)
```

Scope	MPI 0/32 (us) (I)	MPI 0/32 (us) (E)	MPI 0/64 (us) (I)	MPI 0/64 (us) (E)	MPI 1/64 (us) (I)	MPI 1/64 (us) (E)...
Experiment Aggregate Metrics	9.49e+08 100 %	9.49e+08 100 %	5.20e+08 100 %	5.20e+08 100 %	5.16e+08 100 %	5.16e+08 100 %
main	9.49e+08 100 %		5.20e+08 100 %		5.16e+08 100 %	
__f90_main	9.49e+08 100 %		5.20e+08 100 %		5.16e+08 100 %	
MAIN__	9.49e+08 100 %		5.20e+08 100 %	5.00e+03 0.0%	5.16e+08 100 %	5.00e+03 0.0%
loop at main.F90: 66-198	9.42e+08 99.2%		5.08e+08 97.8%	5.00e+03 0.0%	5.08e+08 98.5%	5.00e+03 0.0%
loop at main.F90: 67-185	9.42e+08 99.2%		5.08e+08 97.7%	5.00e+03 0.0%	5.08e+08 98.5%	5.00e+03 0.0%
loop at main.F90: 120-167	7.98e+08 84.1%		4.29e+08 82.4%	5.00e+03 0.0%	4.29e+08 83.0%	5.00e+03 0.0%
loop at main.F90: 128-148	7.69e+08 81.0%		4.04e+08 77.7%	5.00e+03 0.0%	4.04e+08 78.3%	5.00e+03 0.0%
pushe	3.37e+08 35.5%	2.45e+08 25.8%	1.72e+08 33.2%	1.25e+08 24.1%	1.73e+08 33.4%	1.26e+08 24.5%
pushe	3.23e+08 34.0%	2.31e+08 24.3%	1.66e+08 31.9%	1.20e+08 23.1%	1.66e+08 32.1%	1.19e+08 23.1%
shifte	5.48e+07 5.8%	4.08e+07 4.3%	3.31e+07 6.4%	2.08e+07 4.0%	3.32e+07 6.4%	2.08e+07 4.0%
shifte	5.42e+07 5.7%	4.04e+07 4.2%	3.29e+07 6.3%	2.07e+07 4.0%	3.27e+07 6.3%	2.03e+07 3.9%
timer	1.50e+04 0.0%		2.50e+04 0.0%		2.00e+04 0.0%	

Figure 1. Using `hpcviewer` to assess the hotspots in GTC.

- **Callers view**. This bottom-up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are called in more than one context. For instance, a message-passing program may call `MPI_Wait` in many different calling contexts. The cost of any particular call will depend upon its context. Serialization or load imbalance may cause long waits in some calling contexts but not others. Figure 2 shows a caller's view of costs for processes from two parallel runs of GTC.
- **Flat view**. This view organizes performance data according to an application's static structure. All costs incurred in any calling context by a procedure are aggregated together in the flat view. This complements the calling context view, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

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



```
subroutine poisson(iflag)
  use global_parameters
  use field_array
  use particle_decomp
  implicit none

  integer iflag,i,it,ij,j,k,n,iteration,mring,mindex,mtest,ierr
  integer,dimension(:,:),allocatable :: nindex
  integer,dimension(:,:,:),allocatable :: indexp
  real(wp),dimension(:,:,:),allocatable :: ring
  real(wp) gamma,tmp,prms,perr(mgrid)
  real(wp) ptilde(mgrid),phitmp(mgrid),dentmp(mgrid)

  integer :: ipartd,nzeta,izeta1,izeta2
  real(wp),dimension(:),allocatable :: sendbuf,recvbuf
```

Scope	MPI 0/32 (us) (I)	MPI 0/32 (us) (E)	MPI 0/64 (us) (I)	MPI 0/64 (us) (E)	MPI 1/64 (us) (I)	MPI 1/64 (us) (E)...	Percent Excess Work(I)
Experiment Aggregate Metrics	9.49e+08 100 %	9.49e+08 100 %	5.20e+08 100 %	5.20e+08 100 %	5.16e+08 100 %	5.16e+08 100 %	9.13e+00
▼ viutil_spinandwaitcq	3.24e+07 3.4%	2.89e+07 3.0%	2.87e+07 5.5%	2.65e+07 5.1%	2.95e+07 5.7%	2.73e+07 5.3%	2.62e+00
▼ MPID_DeviceCheck	3.24e+07 3.4%	2.89e+07 3.0%	2.87e+07 5.5%	2.65e+07 5.1%	2.95e+07 5.7%	2.73e+07 5.3%	2.62e+00
▼ MPID_SendComplete	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
▼ PMPI_Waitall	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
▼ PMPI_Sendrecv	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
▶ pmpi_sendrecv_	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
▶ MPID_RecvComplete	8.36e+06 0.9%	7.50e+06 0.8%	9.65e+06 1.9%	8.82e+06 1.7%	9.48e+06 1.8%	8.80e+06 1.7%	1.07e+00
▶ poisson	1.66e+07 1.7%	1.64e+07 1.7%	1.70e+07 3.3%	1.68e+07 3.2%	1.71e+07 3.3%	1.69e+07 3.3%	1.83e+00
▶ pushe	6.60e+08 69.5%	4.76e+08 50.1%	3.38e+08 65.0%	2.45e+08 47.2%	3.38e+08 65.5%	2.46e+08 47.6%	1.58e+00
▶ __nanosleep_nocancel	3.00e+04 0.0%	3.00e+04 0.0%	4.78e+06 0.9%	4.78e+06 0.9%	1.47e+06 0.3%	1.47e+06 0.3%	6.56e-01
▶ smooth	7.20e+06 0.8%	5.44e+06 0.6%	7.96e+06 1.5%	5.42e+06 1.0%	7.94e+06 1.5%	5.10e+06 1.0%	5.36e-01
▶ intra_RDMA_barrier	5.06e+06 0.5%	4.28e+06 0.5%	5.15e+06 1.0%	3.82e+06 0.7%	5.20e+06 1.0%	3.69e+06 0.7%	3.39e-01

Figure 2. Using `hpcviewer` to assess the scalability of particle decomposition in GTC.

5.1.1 Using `hpcviewer`

In this section, we illustrate the capabilities of `hpcviewer` by using it to examine profile data collected for GTC. The version of GTC that we studied uses a domain decomposition along the toroidal dimension of a tokamak. Each toroidal domain contains one poloidal plane. One or more MPI processes can be assigned to each toroidal domain. In GTC, many of the more expensive loops are parallelized using OpenMP. Particles in each poloidal plane are randomly distributed in equal number to MPI processes assigned to a toroidal domain. Particles move between poloidal planes via MPI communication.

We used `hpcrun` to collect call stack profiles of three parallel configurations using timer-based asynchronous sampling. All three configurations use the same problem size and domain decomposition along the toroidal dimension; only the degree and type of parallelism within each poloidal plane varies. The baseline configuration uses a single MPI process in each of 32 poloidal planes. The second configuration doubles the amount of parallelism by assigning a second MPI process to each plane. The third configuration uses a hybrid MPI+OpenMP approach, with two threads in each plane.

Figure 1 shows side-by-side views of profile data collected for the MPI rank 0 process of the 32-processor run, along with data for the MPI ranks 0 and 1—the processes for the first poloidal plane in a 64-process MPI execution. For each MPI process, we show two metrics, representing the inclusive and the exclusive wall time spent in each scope. The bottom left of Figure 1 shows the hot call path for the MPI process in the 32-process configuration. A loop nested four levels deep in the main routine accounts for 81% of the total execution time. This loop simulates electron motion. hpcviewer's ability to attribute cost to individual loops comes from information provided by hpcstruct's binary analysis. The cost of simulating electron motion is so high in this simulation because electrons move through the tokamak much faster than ions and need to be simulated at a much finer time scale. From Figure 1 we notice that when we increase the parallelism by a factor of two, the contribution of the electron sub-cycle loop to the total execution time drops to approximately 78%. This is due to a less efficient scaling of other sections of the program, which we explore next.

Figure 2 presents a second snapshot of hpcviewer displaying a bottom-up view of the profile data shown in Figure 1. The last metric shown in this figure is a derived metric representing the percentage of excess work performed in the 64-process run relative to the 32-process run. As we doubled the amount of parallelism within each poloidal plane, the total amount of work performed by the two MPI processes for a plane is roughly 9% larger than the amount of work performed by a single MPI process in the 32-process run. Sorting the program scopes by this derived metric, as shown in Figure 2, enables us to pinpoint those routines whose execution cost has been dilated the most in absolute terms.

We notice that the routine accounting for the highest amount of excess work is viutil_spinandwaitcq. Expanding the calling contexts that lead to this routine reveals that this is an internal routine of the MPI library that waits for the completion of MPI operations. The second most significant routine according to our derived metric is poisson, a GTC routine that solves Poisson equations to compute the electrostatic potential. While this routine accounts for only 1.7% of the execution time in the baseline configuration, we see that its execution time *increases* as we double the level of parallelism in each poloidal plane. In fact, the work performed by this routine is replicated in each MPI process working on a poloidal plane. As a result, the contribution of poisson increases to 3.3% of the total time for the 64-process run. This routine may become a bottleneck as we increase the amount of parallelism within each poloidal plane by higher factors. On a more positive note, Figure 2 shows that routine pushe, which performs electron simulation and accounts for 50% of the total execution time, has very good scaling. Its execution time is dilated less than that of poisson, causing it to be ranked lower according to the excess work metric.

This brief study of GTC shows how the measurement, analysis, attribution, and presentation capabilities of HPCTOOLKIT make it straightforward to pinpoint and quantify the reasons for subtle differences in the relative scaling of different parallel configurations of an application.

5.2 hpctraceview

hpctraceview is a prototype visualization tool that was added to HPCTOOLKIT in summer 2008. hpctraceview renders space-time diagrams that show how a parallel execution unfolds over time. Figure 3 shows a screen snapshot of hpctraceview displaying an interval of execution for a hybrid version of the GTC code running on 64 processors. Although hpctraceview's visualizations on the surface seem rather similar to those by many contemporary tools, the nature of its visualizations and the data upon which they are based is rather different than that of other tools, as we explain.

As we describe in more detail in Section 6, other tools for rendering execution traces of parallel programs rely on embedded program instrumentation that *synchronously* records information about the entry and exit of program procedures, communication operations, and/or program phase markers. Unlike other tools, hpctraceview's traces are collected using *asynchronous* sampling. Each time line in hpctraceview represents a sequence of asynchronous samples taken over the life of a thread or process, which we refer to hereafter as threads in both cases. Also, hpctraceview's samples are

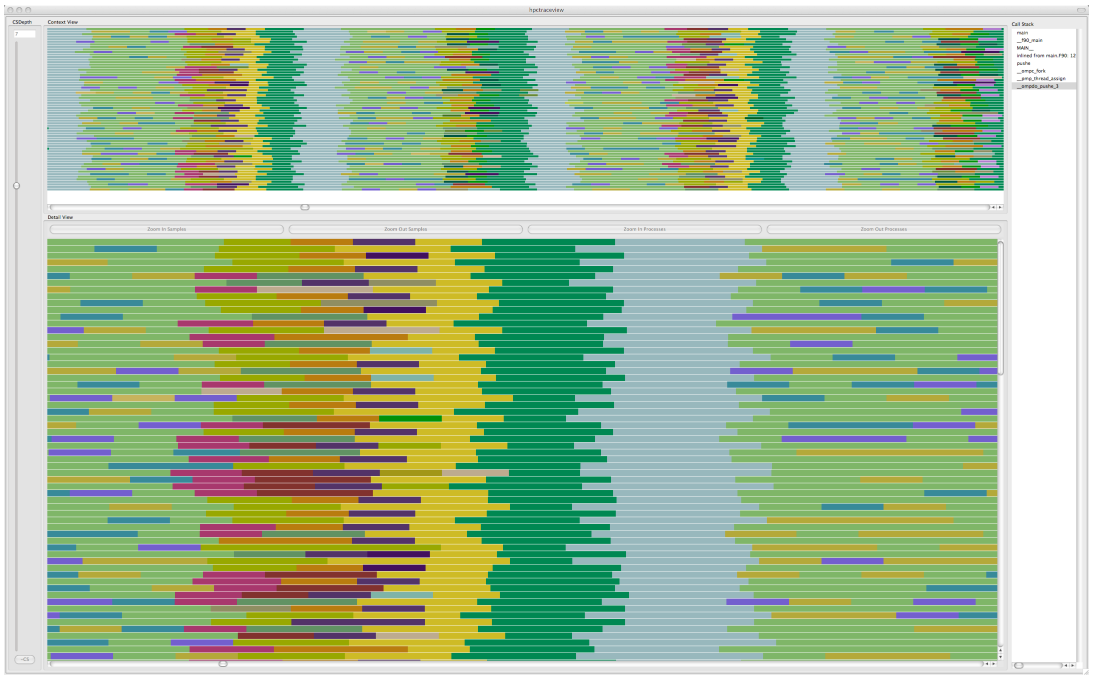


Figure 3. `hpctraceview` showing part of a execution trace for GTC.

multi-level. Each sample for a thread represents the entire stack of the thread's active procedures at the instant when the sample event occurred. As described previously, at a sample event `hpcrun` uses call stack unwinding to identify the stack of active procedures. Call stacks for sample events include *all* active procedures, not just procedures the user wrote. Namely, stack unwinding may reveal calls to math library routines, routines in the MPI API, device driver routines layered underneath an MPI implementation, and even machine-generated procedures outlined from user code when compiling parallel loops in OpenMP programs.

A closer look at Figure 3 reveals `hpctraceview`'s capabilities. The heart of the display is the two center panels that display a set of time lines for threads. Within each panel, time lines for different threads are stacked top to bottom. Even numbered threads (starting from 0) represent MPI processes; odd-numbered threads represent OpenMP slave threads. A thread's activity over time unfolds left to right. The top center panel represents a low-resolution view of the time lines, known as the context view. Each distinct color on the time lines represents a different procedure. Casual inspection of the context view shows three complete repetitions of a "pattern" and part of a fourth. A closer look reveals that the first and third repetitions are somewhat different in nature than the second and fourth: the aforementioned patterns contain a band of yellow bars, whereas the latter do not. Space-time visualizations are good for spotting and understanding such temporal varying behavior. The bottom center pane shows a detailed view of timelines in the context view. One selects a region in the context view to display it in the detail view. Within the detail view, one can scroll and zoom to adjust the content of the view. Unlike other trace visualizers, `hpctraceview`'s visualizations are hierarchical. Since each sample in each thread timeline represents a call stack, we can view the thread timelines at different call stack depths. To the left of the context and detail panes is a slider that can be used to set the level of the view. The space-time diagrams show colored bars representing samples at the selected stack depth; any samples at shallower depth are shown at their deepest level. Figure 3 shows the trace at a call stack depth of seven. In the detailed view, one can use a pointing device to select a colored sample in the detailed view. The rightmost pane of the display shows the complete call stack for the yellow bar representing the conditional copy loop for the topmost process in the context view.

6 RELATED WORK

Many performance tools focus on a particular dimension of measurement. For example, several tools use tracing [4, 12, 26, 37, 39, 41] to measure how an execution unfolds over time. Tracing can provide valuable insight into phase and time-dependent behavior and is often used to detect MPI communication inefficiencies. In contrast, profiling may miss time-dependent behavior, but its measurement, analysis, and presentation strategies scale more easily to long executions. For this reason, other tools employ profiling [2,6,19]. Some tools [14,16,28,29], now including HPCTOOLKIT, support both profiling and tracing. Because either profiling or tracing may be the best form of measurement for a given situation, tools that support both forms have a practical advantage.

Either profiling or tracing may expose aspects of an execution's state such as calling context to form call path profiles or call path traces. Although other tools [14, 28, 36] collect calling contexts, HPCTOOLKIT is unique in supporting both call path profiling and call path tracing. In addition, our call path measurement has novel aspects that make it more accurate and impose lower overhead than other call graph or call path profilers; a detailed comparison can be found elsewhere [9, 10].

Tools for measuring parallel application performance are typically model dependent, such as libraries for monitoring MPI communication (*e.g.*, [35, 36, 40]), interfaces for monitoring OpenMP programs (*e.g.*, [4, 24]), or global address space languages (*e.g.*, [30]). In contrast, HPCTOOLKIT can pinpoint contextual performance problems independent of model—and even within stripped, vendor-supplied math and communication libraries.

Although performance tools may measure the same dimensions of an execution, they may differ with respect to their measurement methodology. Tau [28], OPARI [24], and Pablo [27] among others add instrumentation to source code during the build process. Model-dependent strategies often use instrumented libraries [4, 20, 21, 23, 35]. Other tools analyze unmodified application binaries by using dynamic instrumentation [3, 7, 14, 22] or library preloading [6, 9, 16, 25, 29]. These different measurement approaches affect a tool's ease of use, but more importantly fundamentally affect its potential for accurate and scalable measurements. Tools that permit monitoring of unmodified executables are critical for applications with long build processes or for attaching to an existing production run. More significantly, source code instrumentation cannot measure binary-only library code, may affect compiler transformations, and incurs large overheads. Binary instrumentation may also have blind spots and incur large overheads. For example, the widely used VTune [14] call path profiler employs binary instrumentation that fails to measure functions in stripped object code and imposes enough overhead that Intel explicitly discourages program-wide measurement. HPCTOOLKIT's call path profiler uniquely combines preloading (to monitor unmodified dynamically-linked binaries), asynchronous sampling (to control overhead), and binary analysis (to assist handling of unruly object code) for measurement.

Tracing on large-scale systems is widely recognized to be costly and to produce massive trace files [35]. Consequently, many scalable performance tools manage data by collecting summaries based on synchronous monitoring (or sampling) of library calls (*e.g.*, [35, 36]) or by profiling based on asynchronous events (*e.g.*, [2, 6, 19]). HPCTOOLKIT's call path tracer uses asynchronous sampling and novel techniques to manage measurement overhead and data size better than a flat tracer.

Tools for analyzing bottlenecks in parallel programs are typically *problem focused*. Paradyn [22] uses a performance problem search strategy and focused instrumentation to look for well-known causes of inefficiency. Strategies based on instrumentation of communication libraries, such as Photon and mpiP, focus only on communication performance. Vetter [34] describes an assisted learning based system that analyzes MPI traces and automatically classifies communication inefficiencies, based on the duration of primitives such as blocking and non-blocking send and receive. EXPERT [38] also examines communication traces for patterns that correspond to known inefficiencies. In contrast, HPCTOOLKIT's scaling analysis is *problem-independent*.

7 CONCLUSIONS AND FUTURE DIRECTIONS

Much of the focus of the HPCTOOLKIT project has been on measurement, analysis, and attribution of performance within processor nodes. Our early work on measurement focused on "flat" statistical sampling of hardware performance counters that attributed costs to the instructions and loops that incurred them. As the scope of our work broadened from analysis of computation-intensive Fortran programs (whose static call graphs were often tree-like) to programs that make extensive use of multi-layered libraries, such as those for communication and math, it became important to gather and attribute information about costs to the full calling contexts in which they were incurred. HPCTOOLKIT's use of binary analysis to support both measurement (call stack unwinding of unmodified optimized code) and attribution to loops and inlined functions has enabled its use on today's grand challenge applications—multi-lingual programs that leverage third-party libraries for which source code and symbol information may not be available.

Our observation that one could use differential analysis of call path profiles to pinpoint and quantify scalability bottlenecks led to an effective technique that can be used to pinpoint scalability bottlenecks of all types on systems of any size, independent of the programming model. We have applied this approach to pinpoint synchronization, communication, and I/O bottlenecks on applications on large-scale distributed-memory machines. In addition, we've used this technique to pinpoint scalability bottlenecks on multi-core processors—program regions where scaling from one core to multiple cores is less than ideal.

A blind spot when our tools used profiling exclusively was understanding program behavior that differs over time. Call stack tracing and the `hpctraceview` visualizer enables us to address this issue. A benefit of our tracing approach based on asynchronous rather than synchronous sampling is that we can control measurement overhead by reducing sampling frequency, whereas synchronous sampling approaches have less effective options.

We are in the process of working to scale our measurement and analysis techniques to emerging petascale systems. Many challenges remain. Collecting and saving measurement data to disk for tens to hundreds of thousands of cores is a problem in its own right. The size of the data is not the problem; it is OS limits on the maximum number of open files or files in a directory. This is not an insurmountable problem, but it cannot be ignored either. Our current approach of post-processing call path profiles for parallel programs after execution on a head node of a cluster becomes unrealistic when one wants to analyze profiles for tens of thousands of nodes. Clearly, we need to bring parallelism to bear in our post-processing of measurement data. At present, `hpctraceview` is a prototype tool constructed as a proof of concept that visualizing traces of call stack samples for parallel programs leads to insight into temporal behavior of programs. In building the prototype, we focused on functionality first rather than designing a tool meant to scale to long traces for thousands of processors. Making this tool useful for the largest systems of today will require a redesign so that the tool manipulates traces out of core rather than requiring that all data be memory resident.

ACKNOWLEDGEMENTS

HPCTOOLKIT include project alumni Nathan Froyd and Robert Fowler. Cristian Coarfa was involved in the development of scalability analysis using call path profiles.

REFERENCES

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.
2. T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.
3. B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
4. J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proc. of the Intl. Workshop on OpenMP Appl. and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.
5. C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of spmd codes using expectations. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 13–22, New York, NY, USA, 2007. ACM.
6. D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin 2000 and Onyx2 performance tuning and optimization guide. Technical Report 007-3430-003, Silicon Graphics, Inc., 2001.
7. L. DeRose, J. Ted Hoover, and J. K. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.
8. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.
9. N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.
10. N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *GCC Summit '06: Proceedings of the GCC Developers' Summit, 2006*, pages 21–36, 2006.
11. S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.
12. W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.
13. J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference (SHPCC)*, pages 841–850, 1994.
14. Intel Corporation. Intel VTune performance analyzers. `http://www.intel.com/software/products/vtune/`.
15. Jefferson Lab. The Chroma library for lattice field theory. `http://usqcd.jlab.org/usqcd-docs/chroma`.
16. Krell Institute. Open SpeedShop for Linux. `http://www.openspeedshop.org`, 2007.
17. Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281(5384):1835–1837, September 1998.

18. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
19. J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.
20. Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface Standard*, 1997.
21. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1999.
22. 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.
23. B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.
24. B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001. CD-ROM.
25. P. J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. `http://icl.cs.utk.edu/m̃ucci/papiex/ papiex.html`, 2007.
26. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
27. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proc. of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.
28. S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.
29. Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. Technical Report 007-3311-011, SGI, 2003.
30. H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. B. III, and A. D. George. GASP! a standardized performance analysis tool interface for global address space programming models. Technical Report LBNL-61659, Lawrence Berkeley National Laboratory, 2006.
31. N. Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. M.S. thesis, Department of Computer Science, Rice University, May 2007.
32. N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. Technical report, Rice University, August 2008.
33. USQCD. U.S. lattice quantum chromodynamics. `http://www.usqcd.org`.
34. J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, 2000.
35. J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 240–250, NY, NY, USA, 2002. ACM Press.
36. J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the* 8^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.
37. F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Julich, May 2004.
38. F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proc. of the European Conference on Parallel Computing*, Pisa, Italy, Aug. 2004.
39. P. H. Worley. MPICL: a port of the PICL tracing logic to MPI. `http://www.epm.ornl.gov/picl`, 1999.
40. C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.
41. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.