Final Report

Reporting Period September 2006 – December 2012

Performance Engineering Research Institute[*]

Rice University Subproject

Cooperative Agreement No. DE-FC02-06ER25762

John Mellor-Crummey

Principal Investigator at Rice University

Department of Computer Science, MS 132
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Voice: 713-348-5179
FAX: 713-348-5930
Email: johnmc@cs.rice.edu

[*]Multi-institutional center Project Director: Robert F. Lucas, University of Southern California, rflucas@isi.edu

Contents

1 Introduction

This document reports on work performed for cooperative agreement DE-FC02-06ER25764, the Rice University effort of Performance Engineering Research Institute (PERI), which was an Enabling Technologies Institute of the Scientific Discovery through Advanced Computing (SciDAC-2) program as part as the Department of Energy's Office of Science (DOE SC) Advanced Scientific Computing Research (ASCR) program.

The main priority of this institute is to enhance the performance of SciDAC applications on petascale systems. To address this objective, PERI has implemented three research thrusts: (1) performance modeling and prediction; (2) automatic performance tuning; and (3) performance engineering of high profile applications.

The PERI effort at Rice University focused on (1) research and development of tools for measurement and analysis of application program performance, and (2) engagement with SciDAC-2 application teams.

The goal of the performance tools research at Rice was to advance the state of the art in measurement, attribution, analysis, and diagnosis of inefficiencies in executions of fully-optimized applications. The project team worked to extend Rice's HPCToolkit suite of multiplatform performance analysis tools to provide effective support for application triage. The project team delivered an open source implementation of HPCToolkit for measurement and analysis of application performance on clusters and DOE leadership-class systems.

2 Summary of Research Performed

In this section we briefly describe the research activities and accomplishments under PERI by Rice University researchers. These efforts were tightly coordinated with open source software development supported by Center for Scalable Application Development Software (CScADS).

2.1 Performance Tools for Application Triage

2.1.1 Call Path Profiling of Optimized Code

In modern, modular scientific programs, it is important to attribute the costs incurred by each procedure to the contexts in which the procedure is called. The costs of communication primitives, operations on data structures, and library routines can vary widely depending upon their calling context. Because there are often layered implementations within applications and libraries, it is insufficient to insert instrumentation at any one level, nor is it sufficient to distinguish costs based only upon the immediate caller.

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

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

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

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

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

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

1. p ends with an unconditional branch to an address a that is in the interior of another procedure q.
2. The instruction preceding a is conditional branch to the beginning of p.

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

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

2.1.2 Associating Measurements with Source Code Structure

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

To correlate performance data with the static structure of fully optimized binaries, we built a mapping between object code and its associated source code structure. Since the most important elements of the source code structure are procedures and loop nests — procedures embody the actual executable code while loops often consume the bulk of the executable time — we focus our efforts recovering them. We developed two novel binary analysis techniques: 1) on-the-fly analysis of optimized machine code to enable minimally intrusive and accurate attribution of costs to dynamic calling contexts; and 2) post-mortem analysis of optimized machine code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code. By combining the recovered static program structure with dynamic calling context information, we can accurately attribute performance metrics to calling contexts, procedures, loops, and in-lined instances of procedures.

This technique used only minimal symbolic information, for any portion of the calling context, even without the source code itself. Using binary analysis to recover source code structure addresses the complexity of real sys- tems in which source code for libraries is often missing. We conclude that our binary analyses enable a unique combination of call path data and static source code structure; and this combination provides unique insight into the performance of modular applications that have been subjected to complex compiler transformations. The publication of this technique [16] received the distinguished paper award at the ACM Conference on Programming Language Design and Implementation (PLDI) in 2009 for a paper about their work on call stack profiling of optimized code.

2.1.3 Pinpointing Performance Bottlenecks on Multicore Nodes

Understanding why the performance of a multithreaded program does not improve linearly with the number of cores in a shared-memory node populated with one or more multicore processors is a problem of great importance since the number of cores in multicore processors is steadily increasing. With support from PERI, we have developed new techniques for measurement and analysis of the performance of multithreaded programs and prototyped them in the context of the HPCToolkit performance tools.

Quantifying insufficient parallelism. To quantify insufficient parallelism, we describe how to efficiently and directly measure *parallel idleness*, i.e., when threads are idling or blocked and unable to perform useful work. Our measurements of idleness are based on sampling of a time-based counter such as the wall clock or a hardware cycle counter. Measurement overhead is low and controllable by adjusting the sampling frequency. When a sample event occurs, a signal handler collects the context for the sample and associates the sample count with its context.[1] Collecting parallel idleness on a node with n processor cores requires minor adjustments to traditional time-based sampling. The first adjustment is to extend the run-time system to always maintain n_w and $n_{\overline{w}}$, the number of working and idle processor cores, respectively. This can be done by maintaining a node-wide counter representing n_w. When a core acquires a unit of useful work (e.g., acquiring a procedure activation using work stealing or plucking a unit of work from a task queue), it atomically increments n_w. Similarly, when a core finishes a unit of work, it atomically decrements n_w to indicate that it is no longer actively working. In this scheme $n_{\overline{w}} = n - n_w$.

Consider a run-time system that has one worker thread per core. On each sample, each thread receives an asynchronous signal. If a sample event occurs in a thread that is not working, we ignore it. When a sample event occurs in a thread that is actively working, the thread attributes one sample to a *work* metric for the sample context. It then obtains n_w and $n_{\overline{w}}$ and attributes a fractional sample $n_{\overline{w}}/n_w$ to an *idleness* metric for the sample context. This charges the thread its proportional responsibility for not keeping the idle processors busy at that moment at that point in the program. For example, if three threads are active on a quad core processor, whenever a sample event for the cycle counter interrupts a working thread, the working thread will record one sample of work in its *work* metric, and 1/3 sample of idleness in its *idleness* metric. The 1/3 sample of idleness represents its share of the responsibility for the core that is sitting idle.

After measurement is completed, idleness can be computed for each program context. Since samples are accumulated during measurement, the idleness value for a given thread and context is $\sum n_{\overline{w}_i}$ over all samples i for that context. It is often useful to express this idleness metric as a percentage of the total idleness for the program. Total idleness may be computed post-mortem by summing idleness metric over all threads and contexts in the

[1] We attribute costs to their full calling context using call path profiling. In this section, we use the term *context* rather than *calling context* since idleness can be measured with or without full calling context.

parallel		
idleness	overhead	interpretation
low	low	effectively parallel; focus on serial performance
low	high	coarsen concurrency granularity
high	low	refine concurrency granularity
high	high	switch strategies; e.g., consider task parallelization

Table 1: Using parallel idleness and overhead to determine if the given application and input are effectively parallel on n cores.

program. The idleness value may be converted to a time unit by multiplying by the sample period. One can also divide the idleness for each context by the application's *total effort*—the sum of work and idleness everywhere across all threads—to understand the fraction of total effort that was wasted in each context.

This approach incurs $< 5\%$ overhead on a quantum chemistry application that makes extensive use of locking (65M distinct locks, a maximum of 340K live locks, and an average of 30K lock acquisitions per second per thread) and attributes lock contention to its full static and dynamic calling contexts [13].

Quantifying parallelization overhead. Parallel overhead occurs when a thread is performing miscellaneous work other than executing the user's computation. Sources of parallel overhead include costs such as those for synchronization or dynamically managing the distribution of work.

For library-based parallel programming models such as Pthreads, identifying parallel overhead is easy: any time spent in a routine in the Pthreads library can be labeled as parallel overhead. For language-based parallel programming models, one must rely on compiler support to identify inlined sources of parallel overhead. A compiler for a multi-threaded programming model, such as OpenMP or Cilk, can tag statements in its generated code to indicate which are associated with parallelization overhead. In a post-mortem analysis, we recover compiler-recorded information about overhead statements, identify instructions associated with overhead statements and run-time library routines, and attribute any samples of *work* associated with them to parallelization overhead [18]. The tags therefore partition the application code — the 'work' — into useful work and overhead (distinct from idleness).

This scheme has two important benefits. First, compiler generated tags may be designed to partition sources of overhead into multiple types, thereby providing detailed information to users or analysis tools. For example, it may be useful to distinguish between synchronization overhead and all other overhead. The second benefit is that, tags are only meta-information; they can be inserted and overhead can be associated with them using post-mortem analysis without affecting run time performance in *any* way. In particular, tags do *not* have any associated instrumentation. While the mapping between instructions and tags consume space, it need not induce any run time cost. For example, the mapping can be located within a section of a compiled binary that is not loaded into memory at run time or maintained in a separate file.

Analyzing efficiency. In a parallel program, we must consider two kinds of efficiency: parallel efficiency across multiple processor cores and efficiency on individual processor cores. With information about parallel idleness and overhead attributed hierarchically over loops,[2] procedures, and the calling contexts of a program, we can directly assess parallel efficiency and provide guidance for how to improve it (see Table 1). If a region of the program (e.g., a parallel loop) is attributed with high idleness and low overhead, the granularity of the parallelism could profitably be reduced to enhance parallel efficiency. If the overhead is high and the idleness low, the granularity of the parallelism should be increased to reduce overhead. If the overhead is high and there is still insufficient parallelism, the parallelism is inefficient and no granularity adjustment will help; perhaps the idle processors could be kept busy with a completely different type of work (functional parallelism).

One can assess the efficiency of *work* and identify rate limiting factors on individual processor cores by using metrics derived from hardware performance counter measurements. Many different factors can limit an application's performance such as instruction mix, memory bandwidth, memory latency, and pipeline stalls. For each of these factors, information from hardware performance counters can be used to compute derived metrics that quantify the extent to which the factor is a rate limiter. Consider how to assess whether memory bandwidth is a rate limiter. During an execution, one can sample hardware counter events for *total cycles* and *memory bus transactions*. By multiplying the sampling period by the sample count for each instruction, one can obtain an estimate of how many bus transactions are associated with each instruction. By multiplying the number of bus transactions by the transaction granularity (e.g., the line size for the lowest level cache), one can compute the amount of data transferred by each instruction. By dividing the amount of data transferred by instructions within a scope (e.g., loop) by the total number of cycles spent in that scope, one can compute the memory bandwidth consumed in that scope. By comparing that with a model of peak bandwidth achievable on the architecture, one can determine whether a loop is bandwidth bound or not. Attributing metrics to static scopes such as loops and dynamic contexts such as call paths to support such analysis of multithreaded programs is the topic of the next section.

Quantifying OpenMP performance bottlenecks. We have developed a measurement methodology that attributes blame for work and inefficiency back to program contexts. We show how to integrate prior work on measurement methodologies that employ directed and undirected blame shifting and extend the approach to support dynamic thread-level parallelism in both time-shared and dedicated environments.

We also developed a novel deferred context resolution method that supports online attribution of performance metrics to full calling contexts within an OpenMP program execution. This approach enables us to collect compact call path profiles for OpenMP program executions without the need for traces. Support for our approach is an integral part of an emerging standard performance tool application programming interface for OpenMP. We demonstrate

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

the effectiveness of our approach by applying our tool to analyze four well-known application benchmarks that cover the spectrum of OpenMP features. In case studies with these benchmarks, insights from our tool helped us significantly improve the performance of these codes.

We demonstrated that an implementation of these techniques in HPCToolkit provides deep insight into the performance of threaded program executions by measuring and attributing informative metrics including idleness, work, overhead, and lock waiting. Our OpenMP profiler employs online deferred context resolution to efficiently and accurately attribute these metrics to full calling contexts in compact profiles, avoiding the space overhead of traces required by prior tools. Reducing the space overhead is an important aspect of our strategy that will enable it to scale to large parallel systems [8].

2.1.4 Logical Call-Path Profiles

To enable effective performance analysis of higher-level programming languages it is necessary to bridge the gap between the user's abstractions and their instantiation at run time. A key aspect of this is recovering *logical* user-level calling contexts. We extend the notion of call path profiling by defining *logical call paths* and describing how to generally and efficiently obtain logical call path profiles using a *logical calling context tree*. Note that this technique applies to both parallel and serial applications. To explore the utility of of logical call path profiling, we used it to construct a system for measurement and analysis of multithreaded Cilk program executions, which are managed by a work-stealing run-time system [15].

2.2 ARRA Supplement: Support for Accelerated Computing

Rice University received supplemental funding from the ARRA/DOE-S: SciDAC-e program to work with the University of North Carolina and Lawrence Berkeley National Laboratory on a project entitled *Enhancing Productivity of Materials Discovery Computations for Solar Fuels and Next Generation Photovoltaics.* This project has brought together a group of computer scientists and applied mathematicians from the SciDAC Performance Engineering Research Institute (PERI) to work on improving the productivity of the computational activities of the University of North Carolina's EFRC, as well as its overall research program. Distinct from other PERI activities, these researchers have applied the full breadth of their knowledge and technologies, beyond that used in SciDAC projects, to this problem.

Here, we describe the contributions of Rice University to the overall Scidac-e project. Rice University's specific charge for this project was two-fold: (1) develop methods for performance analysis of programming systems that employed NVIDA GPUs, and (2) explore strategies for GPU-acceleration of linear algebra.

2.2.1 Measurement and Analysis of GPU-accelerated Applications

Our first task was to develop viable methods for analyzing the performance of GPU-enhanced codes. There were two subtasks of this effort: (1) understand methods for analyzing the performance of GPU kernels, and (2) develop methods for analyzing the systemic behavior of CPU+GPU systems. Below we describe our work on each of these tasks.

Modern GPUs have performance counters, but access to these counters is tedious and limited. Prior to the development of NVIDIA's CUDA Profiling Tool Inferface (CUPTI), programmers instrumented their GPU-accelerated programs by hand. For each kernel of interest, GPU programmers would start and stop some embedded performance counters, storing the results in auxiliary program variables. The performance values would be analyzed later.

NVIDIA did make some progress to ease the burden of analyzing GPU kernels with later versions of CUPTI. Regrettably, the latest CUPTI is *not* sufficient to allow a comprehensive treatment of GPU performance analysis. From our perspective, CUPTI has the following weaknesses:

- *No sampling on GPUs.* This is the most crucial weakness with GPU performance analysis. No sampling means that performance analysis *must* come from instrumentation. Extensive instrumentation makes data collection very slow — slow enough to actually distort results. The lack of sampling capability on GPUs is in stark contrast to the situation on standard GPUs.

- *Must Serialize Kernels.* Caliper-based measurement of GPU performance metrics cannot be separated on a per-kernel basis. Best practice has programmers instrumenting each kernel, and collecting hardware-counter performance data one kernel at a time. Furthermore, even when collecting data on a single kernel, the kernels must be serialized. Otherwise, data from one thread (or stream) would bleed into another thread or stream.

In spite of these weaknesses, we were able to integrate the collection and analysis of GPU-specific performance information into the HPCToolkit framework.

To date, much of the work on performance analysis of heterogeneous architectures focused on identifying performance problems in GPU kernels. While identifying GPU kernel-level issues is important, this is only one aspect of the larger problem. Whole application performance analysis is equally important for tuning large GPU-accelerated applications. Such analysis requires a system-level view of performance data. Hence, the data collection question reduces to deciding what kinds of *system-level* analyses can best augment standard component-level profiles and traces.

Studies have demonstrated that dynamically partitioning an application's work between CPU and GPU is important for delivering high efficiency for a variety of applications. Consequently, we developed an analysis technique to address the work-partitioning issue. Evaluating the effectiveness of an application's work partitioning is a systemic question. It is not easily addressed by focusing on individual components.

Any tool that focuses on *hot spot analysis* can only quantify *where* a program spends its resources. Each component may have different hot spots. At best, hot spot analysis measures and reports the symptoms of performance problems. Hot spot analysis doesn't necessarily guide the developer towards root causes of performance problems in GPU-accelerated applications. The sample shown in Figure 1 highlights this point. If the CPU code executing during the interval labeled `Kernel A` cannot be tuned further, then improving `Kernel A`, a GPU kernel whose execution is overlapped with `A`, will not shorten the execution by more than 5%—the time that the CPU sits idle awaiting the results of `Kernel A`. However, in the same application, the CPU sits idle for 40% of the execution awaiting the completion of GPU `Kernel B`; hence, tuning `Kernel B` could reduce the execution time by up to 40%. Hot spot analysis would point to `Kernel A` as the most time consuming GPU kernel, and thus fail to guide a programmer to `Kernel B`. `Kernel B` represents a better opportunity for tuning. This problem is exacerbated in full applications with several kernels and more complicated execution schedules.

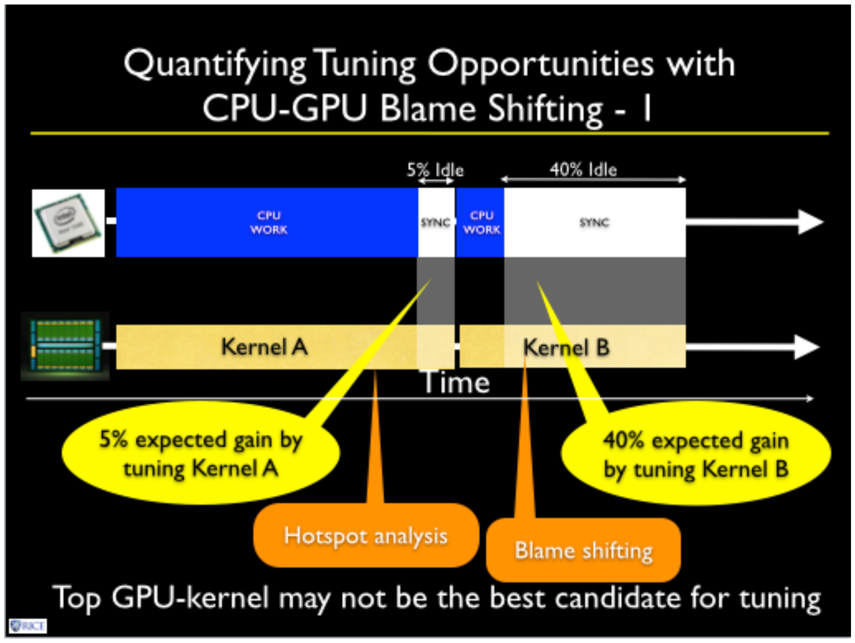
Quantifying Tuning Opportunities with CPU-GPU Blame Shifting - I
5% Idle
40% Idle
CPU WORK
SYNC
CPU WORK
SYNC
Kernel A
Kernel B
Time
5% expected gain by tuning Kernel A
40% expected gain by tuning Kernel B
Hotspot analysis
Blame shifting
Top GPU-kernel may not be the best candidate for tuning

Figure 1: Tuning the *right* kernel

Likewise, choosing the appropriate CPU function to optimize in a CPU+GPU system can benefit from a blame-shifting analysis. In Figure 2, the interval labeled `CPU Part A` has the GPU idle for a significant fraction of the execution time. `CPU Part B`, however has extremely good overlap with the GPU. Hotspot analysis, however, would identify `CPU Part B` as the routine that takes more time. In Figure 2, improving `CPU Part B` would not yield any improvement — more time would be spent idling waiting for the GPU to finish.

The difference between hot-spot analysis and blame-shifting analysis can be dramatic. Figure 3 illustrates these differences for a LULESH (a sophisticated Euler-Lagrange dynamics code)

To address the limitations of hot spot analysis, we supplement it with novel systemic *idleness* analysis. Our idleness analysis identifies *CPU* code regions that cause *GPU* resources to sit idle. Symmetrically, our approach also pinpoints *GPU* kernels that cause *CPU* cores to sit idle. Moreover, our analysis quantifies the amount of idleness due to each offending CPU code region or GPU kernel. Normally, this sort of systemic analysis would require postmortem analysis of execution *traces*. Our analysis, however, requires only a *profile*. The reason the idleness analysis can be done without traces is due to a technique that we developed called *CPU-GPU blame shifting*.

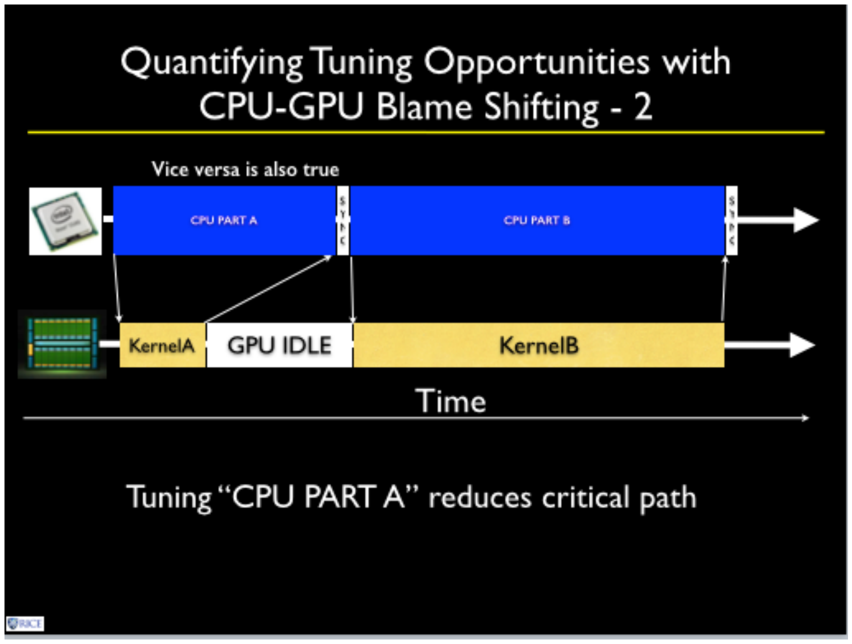
Quantifying Tuning Opportunities with CPU-GPU Blame Shifting - 2
Vice versa is also true
CPU PART A
CPU PART B
sync
sync
KernelA
GPU IDLE
KernelB
Time
Tuning "CPU PART A" reduces critical path

Figure 2: Picking the *right* CPU function

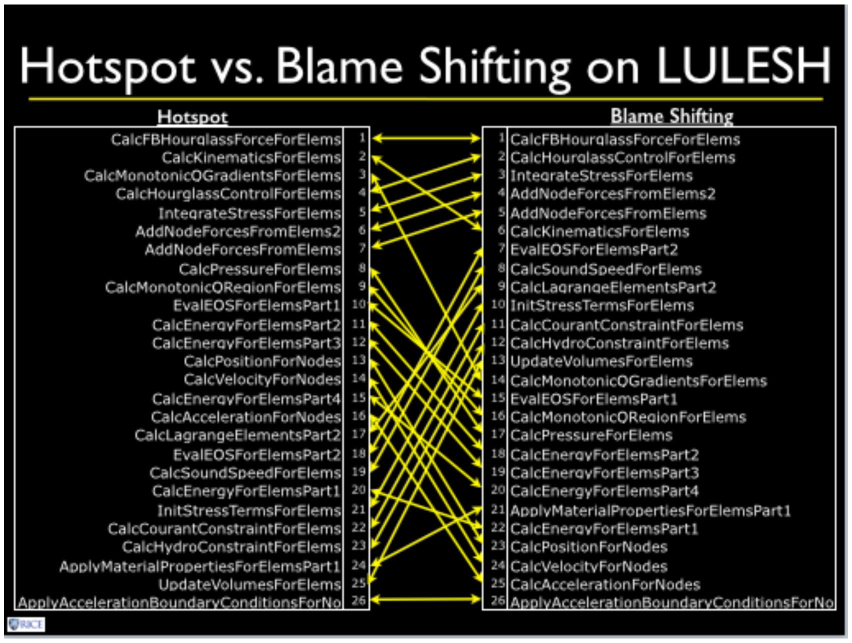
Hotspot vs. Blame Shifting on LULESH
Hotspot
CalcFBHourglassForceForElems 1
CalcKinematicsForElems 2
CalcMonotonicQGradientsForElems 3
CalcHourglassControlForElems 4
IntegrateStressForElems 5
AddNodeForcesFromElems2 6
AddNodeForcesFromElems 7
CalcPressureForElems 8
CalcMonotonicQRegionForElems 9
EvalEOSForElemsPart1 10
CalcEnergyForElemsPart2 11
CalcEnergyForElemsPart3 12
CalcPositionForNodes 13
CalcVelocityForNodes 14
CalcEnergyForElemsPart4 15
CalcAccelerationForNodes 16
CalcLagrangeElementsPart2 17
EvalEOSForElemsPart2 18
CalcSoundSpeedForElems 19
CalcEnergyForElemsPart1 20
InitStressTermsForElems 21
CalcCourantConstraintForElems 22
CalcHydroConstraintForElems 23
ApplyMaterialPropertiesForElemsPart1 24
UpdateVolumesForElems 25
ApplyAccelerationBoundaryConditionsForNo 26
Blame Shifting
1 CalcFBHourglassForceForElems
2 CalcHourglassControlForElems
3 IntegrateStressForElems
4 AddNodeForcesFromElems2
5 AddNodeForcesFromElems
6 CalcKinematicsForElems
7 EvalEOSForElemsPart2
8 CalcSoundSpeedForElems
9 CalcLagrangeElementsPart2
10 InitStressTermsForElems
11 CalcCourantConstraintForElems
12 CalcHydroConstraintForElems
13 UpdateVolumesForElems
14 CalcMonotonicQGradientsForElems
15 EvalEOSForElemsPart1
16 CalcMonotonicQRegionForElems
17 CalcPressureForElems
18 CalcEnergyForElemsPart2
19 CalcEnergyForElemsPart3
20 CalcEnergyForElemsPart4
21 ApplyMaterialPropertiesForElemsPart1
22 CalcEnergyForElemsPart1
23 CalcPositionForNodes
24 CalcVelocityForNodes
25 CalcAccelerationForNodes
26 ApplyAccelerationBoundaryConditionsForNo

Figure 3: Hot-spot vs. Blame-shifting in LULESH

The effectiveness of blame-shifting for CPU+GPU systems led us to implement the approach in HPCToolkit. A publication about our GPU blame shifting work appeared in SC13 [4].

2.2.2 GPU-accelerated Linear Algebra

At the start of this subcontract, the BLAS2 linear algebra routines has been implemented in the University of Tennessee's MAGMA package. MAGMA, at the inception of this subcontract, lacked the dense eigenvalue (BLAS3) routines that are an important component in computational quantum chemistry. As part of our charge, we began to develop the missing eigenvalue routines. Before the contract was completed, however, the MAGMA team released MAGMA version 1.3. This MAGMA release has all of the necessary eigenvalue routines, and works on multiple GPUs. Furthermore, the MAGMA approach was better than our approach. Our approach was to do the bulk of the calculation on the GPU, much like the BLAS2 implementations. The MAGMA team found a way to favorably divide eigenvalue computation between the CPU and GPU leading to better overall performance.
In conclusion, we evaluated the MAGMA offering, and since it was better than our own, that was our recommendation.

3 Application Engagement

Applying Rice's HPCToolkit to SciDAC and INCITE applications has been one of the principal modes of engaging with the SciDAC application teams.

As part of the Tiger Team efforts in PERI, HPCToolkit has been used to study many SciDAC application. Below, we briefly mention summarize our engagement work with a few of these applications.

Chroma. Chroma is a C++ application for lattice quantum chromodynamics developed as part of the US Lattice Quantum Chromodynamics project. Chroma served as a test case for binary analysis of inlined and templated C++ code. Chroma is built upon the QDP++ package, which uses a highly modular design that makes extensive use of C++ expression templates. Because of its use of expression templates, at compile time complex templates are instantiated, customized for the many different contexts in which they are used, and sometimes inlined.

A challenge for performance tools is being able to cope with the dramatic transformations of user programs performed by the C++ compiler as it compiles template metaprograms that use expression templates. To address this issue, HPCToolkit's call path profiler measured dynamic call path information during execution of Chroma. This dynamic information was combined with information about inlining and loops recovered by HPCToolkit's binary analyzer through static analysis of Chroma's executable [11]. In 2009, at RENCI's request Rice enhanced the support for binary analysis in HPCToolkit to handle the complexity that arises from the use of template metaprogramming. In collaboration with researchers at Jefferson Laboratory, RENCI was able to use the improved HPCToolkit to identify several performance issues previously buried under the template framework. HPCToolkit is unique in its use of binary analysis to recover information about loops and inlined code.

FLASH. FLASH is 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.

We discovered that 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.

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.

On Cray/XT, we discovered that 27.5% of the losses are due to barrier synchronization (the execution of `MPIR_Barrier` routine from the MPI library). By inspecting the callers of this routine, we found 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.

We also investigated `local_tree_build` routine which is part of the PARAMESH library used by Flash. 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. 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`.

Our tool also showed that on Cray/XT, 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.

iMesh. The iMesh interface for meshes is being developed by the ITAPS SciDAC project. As part of work to indentify opportunities for applying dynamic reoptimization to codes and frameworks based on the Common Component Architecture (CCA), we studied the iMesh wrapper for ANL's MOAB mesh library using HPCToolkit. Surprisingly, we found that the implementation of the iMESH wrapper added more overhead than expected because of memory copies introduced in the wrapper. We discussed possible alternate implementation strategies to avoid the copy overhead with the developers of the MOAB package.

MADNESS. We have investigated Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS) quantum chemistry code to identify performance bottleneck with a collaboration with RENCI and ORNL. This led to a deep engagement that involved analyzing node-level performance for multi-core, multi-socket systems. The result of these measurements was that the code was spending increasing amounts of time in locks when scaled from 2 to 4 quad-core processors. Using our new blame-shifting technique, we discovered that the contention was due to contention by threads to access a single shared work queue on which futures are enqueued. Rice provided information back to MADNESS team about the insight into lock contention that we were able to glean using the new support for measurement and analysis in HPCToolkit. ORNL team adjusted MADNESS to include support for dynamically controlling the granularity of work taken from the queue. This improved efficiency and helped the code run efficiently on 12 cores. Previously, performance degraded substantially beyond 6-8 cores. A paper about the new strategy for analyzing lock contention appeared in PPoPP 2010 [13].

MILC. MILC is a lattice quantum chronodynamics (QCD) simultation with dynamical Kogut-Susskind fermions from MILC, or MIMD Lattice Computation package. MILC 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.

Our tool identified that 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, MILC's `setup` phase accounts for most of the scaling losses.

Routine `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.

PFLOTRAN PFLOTRAN is a code for predicting the migration of contaminants underground by modeling multi-phase, multi-component flow, and reactive transport. It solves a coupled system of mass and energy conservation equations for multiple compounds. PFLOTRAN is designed for execution on platforms with large-scale parallelism. The code employs the PETSc library's Newton-Krylov solver framework.

Performance analysis of large-scale parallel runs of PFLOTRAN by members of the PERI tiger team at NC State showed that a significant amount of time was spent in MPI_AllReduce. Scatter plots of time spent in MPI_AllReduce across all nodes showed a wide discrepancy across the nodes of the system. Did this indicate a problem with the implementation of MPI_AllReduce, or something about how it was used in PFLOTRAN? Rice University members of the PERI team took on understanding the nature of this issue as a challenge problem for the HPCToolkit performance tools. A second question of interest was identifying targets of opportunity for node-level autotuning within PFLOTRAN or its supporting libraries.

To understand the first issue, in collaboration with the Center for Scalable Application Development Software, the PERI team at Rice University augmented their HPCToolkit performance tools with support for analyzing the variability of costs across nodes in a parallel system. The HPCToolkit performance tools measure the performance of node threads/processes in a parallel run using statistical sampling to collect call path profiles. This enables HPCToolkit to precisely attribute costs to the full calling context in which they are incurred. Using this new capability for analyzing variability, it became clear that much of the variability in time that was spent in MPI_AllReduce was due to load imbalance in the PETSc Newton-Krylov solver.

The call to SNESComputeJacobian occurs before the call to SNES_KSPSolve. What became apparent from this plot and others similar ones was that (1) there is load imbalance in PFLOTRAN within the PETSc solver, and (2) the load imbalance contributed to uneven waiting times in MPI_AllReduce. The load imbalance in SNESComputeJacobian is evident because the higher numbered cores spend less time in the routine than the lower numbered cores; this is evident from the downward slope of the graph as one moves to the right along the X axis. One can clearly see that the nodes that spent the least time in SNESComputeJacobian spent the most time in SNES_KSPSolve and routines it calls. This time was actually spent waiting in a call to MPI_AllReduce at the beginning of SNES_KSPSolve. Thus, load imbalance was reflected in the imbalance in core timings for MPI_AllReduce. As part of the enhancements to HPCToolkit to support this analysis of PFLOTRAN, the Rice team developed a technique for automatically pinpointing such instances of load imbalance in large-scale parallel executions. In essence, HPCToolkit can automatically identify at what points in the code load imbalance occurs, quantify its effect on program performance, and guide the user to inspect these points in a program using HPCToolkit's `hpcviewer` user interface. This work was presented at SuperComputing 2010 [14].

Early profiling of PFLOTRAN with the TAU performance tools failed to capture enough detail about the location of performance losses to identify opportunities for autotuning. Using Rice University's HPCToolkit performance tools to collect call path profiles of PFLOTRAN executions revealed some problematic kernels in the PETSc solver that are targets of opportunity for autotuning. Each Opteron core on an XT5 node has a maximum peak performance of four double-precision floating point operations (FLOPs) per cycle. Therefore, one can compute floating point waste by subtracting actual floating point throughput from ideal throughput as follows: (4*cycles) - FLOPs. The presentation tool computes this FP waste metric using the cycle and FLOPs summary metrics generated by summing over all processes in the execution. This metric is exclusive, meaning that it excludes callees (hence the 'E' modifier). The second metric is inclusive FLOPs per cycle. Overall, this execution of PFLOTRAN performed 0.160 floating point operations per cycle, which is only 4.0% of peak.

Although this dgemv_n (matrix-vector multiply routine) consumes 14.4% of the execution's cycles, it has a floating point efficiency of 0.187 FLOPs/cycle. For comparison, the matrix-matrix multiply routine dgemm_kernel (not shown) delivers 2.28 floating point operations per cycle. This low efficiency was worthy of further investigation and analysis to see if it can be addressed with autotuning.

4 Products of the Research

4.1 Tools for Performance Analysis of Parallel Programs

With support from the PERI project, we augmented Rice University's HPCToolkit performance tools to support measurement and attribution of multithreaded and optimized codes to support application Triage for the PERI project. HPCToolkit is available as open source software.

4.2 Performance Monitoring of GPU-accelerated Applications

Support for performance analysis of GPU-accelerated code is included in the open-source HPCToolkit performance tools. This software uses blame-shifting to identify idleness of either CPU or GPU hardware and blame that idleness on the software apparently responsible for the idleness.

5 Technical Communications

5.1 Theses

- Nathan Tallent, Doctor of Philosophy. "Performance Analysis for Parallel Programs: From Multicore to Petascale." (2010).[3] He is currently an senior computer scientist at Pacific Northwest National Laboratory.

- Xu Liu. Doctor of Philosophy. Performance Analysis of Program Executions on Modern Parallel Architectures (2014).[4] He is currently an assistant professor at The College of William and Mary.

[3]Nathan Tallent's Ph.D. research was partially supported by PERI.

[4]Xu Liu's Ph.D. research was partially supported by PERI.

5.2 Other Synergistic Activities

The Rice research team is active in many IEEE and ACM technical conferences and workshops such as CGO, PPoPP, PGAS conferences, Supercomputing, and PLDI. Research on performance analysis of multithreaded computations laid the foundation for blame-shifting performance analysis in the OMPT performance tool API, which is an emerging standard interface for performance analysis of OpenMP node programs.

5.3 Awards

- A paper about HPCToolkit's use of static and dynamic binary analysis to attribute performance measurements to full calling contexts for optimized binaries [16] received the distinguished paper award at the ACM Conference on Programming Language Design and Implementation (PLDI) in 2009.
- Nathan Tallent, a member of the HPCToolkit project team, was named one of two George Michael Memorial HPC Fellows for 2009.

5.4 Publications

[1] L. Adhianto et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[2] L. Adhianto, J. Mellor-Crummey, and N. R. Tallent. Effectively presenting call path profiles of application performance. In *Proc. of the 2010 Workshop on Parallel Software Tools and Tool Infrastructures, held in conjunction with the 2010 International Conference on Parallel Processing*, 2010.

[3] M. Chabbi and J. Mellor-Crummey. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 124–134, New York, NY, USA, 2012. ACM.

[4] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for gpu-accelerated supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 43:1–43:12, New York, NY, USA, 2013. ACM.

[5] M. Charney. XED2 user guide. `http://www.pintool.org/docs/24110/Xed/html`.

[6] 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.

[7] M. Krentel. Libmonitor: A tool for first-party monitoring. In *Proceedings of the first Workshop on High-performance Infrastructure for Scalable Tools*, WHIST'11, 2011.

[8] X. Liu, J. Mellor-Crummey, and M. Fagan. A new approach for performance analysis of openmp programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 69–80, New York, NY, USA, 2013. ACM.

[9] G. Marin, G. Jin, and J. Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. *Journal of Physics: Conference Series*, 125(1):012087, 2008.

[10] J. Mellor-Crummey. Harnessing the power of emerging petascale platforms. *Journal of Physics: Conference Series*, 78(1):012048, 2007.

[11] J. Mellor-Crummey and N. Tallent. A methodology for accurate, effective and scalable performance analysis of application programs. In *Workshop on Tools, Infrastructures and Methodologies for the Evaluation of Research Systems, in conjuction with the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 4–11, February 2008.

[12] 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.

[13] N. Tallent, J. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–280, New York, NY, USA, 2010. ACM.

[14] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *SC 2010 International Conference for High-Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, November 2010. ACM.

[15] 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.

[16] 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 PLDI*, pages 441–452, NY, NY, USA, 2009. ACM.

[17] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[18] N. R. Tallent and J. M. Mellor-Crummey. Identifying performance bottlenecks in work-stealing computations. *Computer*, 42(12):44–50, 2009.

[19] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, 2009.

5.5 Presentations

- Milind Chabbi. Critically Missing Pieces on Accelerators: A Performance Tools Perspective. Birds of a Feather session on Critically Missing Pieces on Accelerators, SC13, Denver, CO. November, 2013. Invited Talk.

- Milind Chabbi. HPCToolkit: Performance Tools for Tuning Application on Heterogeneous Supercomputers. Talk at Dagstuhl Seminar on Automatic Application Tuning for HPC Architectures, Germany. October 2013.

- Milind Chabbi, Karthik Murthy, and John Mellor-Crummey. HPCToolkit: A Tool for Performance Analysis of Heterogeneous Supercomputers. Talk at Nvidia GPU Technology Conference GTC 2013, San Jose, CA. March 2013.

- Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Performance Tools for Heterogeneous Supercomputers. Student poster at the 6th Oil and Gas High Performance Computing Workshop, Houston, TX. February 2013.

- David Goodwin, Guido Juckeland, Allen Malony, Milind Chabbi, Stan Tomov. Using the CUDA Profiling API and Related Third Party Tools. Panelist at Nvidia GPU Technology Conference GTC 2013, San Jose, CA. March 2013.

- J. Mellor-Crummey, HPCToolkit: Sampling-based performance tools for leadership computing. Center for Scalable Application Development Software's Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies. July 1922, 2010, Snowbird, Utah, USA.

- N. Tallent. Identifying Scalability Bottlenecks In Large-scale Parallel Programs Using HPCToolkit. Dagstuhl Seminar 10181: Program Development for Extreme-Scale Computing, Dagstuhl, Germany, May 2010. http://bit.ly/dms4bB

- J. Mellor-Crummey, Gaining Insight into Parallel Program Performance Using Sampling. IBM Watson Research Center, Yorktown Heights, NY, October 11, 2010.

- N. Tallent, Scalable Identification of Load Imbalance in Parallel Executions using Call Path Profiles, International Conference on High Performance Computing, Networking, Storage and Analysis (SC10), New Orleans, LA, November 17, 2010.

- N. Tallent, Performance Analysis for Parallel Programs: From Multicore to Petascale, George Michael Memorial HPC Ph.D. Fellowship Presentation, International Conference on High Performance Computing, Networking, Storage and Analysis (SC10), New Orleans, LA, November 17, 2010.