Harnessing the power of emerging petascale platforms

John Mellor-Crummey

Associate Professor, Department of Computer Science, Rice University, Houston, TX, USA

E-mail: johnmc@cs.rice.edu

Abstract. As part of the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC-2) program, science teams are tackling problems that require computational simulation and modeling at the petascale. A grand challenge for computer science is to develop software technology that makes it easier to harness the power of these systems to aid scientific discovery. As part of its activities, the SciDAC-2 Center for Scalable Application Development Software (CScADS) is building open source software tools to support efficient scientific computing on the emerging leadership-class platforms. In this paper, we describe two tools for performance analysis and tuning that are being developed as part of CScADS: a tool for analyzing scalability and performance, and a tool for optimizing loop nests for better node performance. We motivate these tools by showing how they apply to S3D, a turbulent combustion code under development at Sandia National Laboratory. For S3D, our node performance analysis tool helped uncover several performance bottlenecks. Using our loop nest optimization tool, we transformed S3D's most costly loop nest to reduce execution time by a factor of 2.94 for a processor working on a 50^3 domain.

1. Introduction

Harnessing the power of emerging petascale computational platforms to aid scientific discovery poses a grand challenge for computer science. Programming petascale systems that consist of ensembles of tens to hundreds of thousands of microprocessors is difficult at many levels. At the highest level, one must make effective use of coarse-grain parallelism among the system's processors. A variety of problems reduce performance at this level including load imbalance, serialization, and communication overhead.

At the next level, one must make effective use of individual microprocessor-based nodes. Unfortunately, modern microprocessors suffer from a high degree of complexity and each new generation is harder to program near the peak of its capabilities than its predecessor. Today's processor architectures are difficult to use because internally they rely on multiple levels of parallelism including multiple cores, implicit instruction-level parallelism and explicit short-vector parallelism. They are deeply pipelined, out of order, and superscalar. From the perspective of data-intensive scientific codes, today's microprocessor-based architectures also suffer from too little memory bandwidth per operation. To increase the data bandwidth available to applications, microprocessors employ a multi-level memory hierarchy that includes two or more levels of cache, which can increase the effective data bandwidth for applications if they reuse data values. Using a microprocessor to its full potential requires making effective use of all levels of parallelism and exploiting reuse at the highest levels of the memory hierarchy.

As a result, effectively harnessing the power of microprocessor-based petascale systems is a daunting task. Performance tools that help pinpoint bottlenecks and quantify opportunities for improvement are essential for tuning codes to use these platforms efficiently. However, the rapid pace of change in computer architecture makes it impractical for scientists to invest significant effort tuning their applications for any particular architecture. Although dramatic performance improvements can be achieved by hand optimization to reuse data in cache, the tuning process must be automated to improve developer productivity. While high performance is desirable, for long-lived applications it is also important that they be written in a clear style that facilitates code development productivity and maintainability. As a result, compiler-based tools for tailoring codes to better exploit microprocessor-based architectures are essential to help application scientists achieve high performance without compromising the maintainability of their applications.

To help address the challenges of tailoring scientific programs to petascale platforms, the SciDAC-2 Center for Scalable Application Development Software (CScADS) is working to develop software systems including compilers, tools, and libraries to improve programmability, identify performance bottlenecks, and aid in manual and automatic performance tuning. This paper focuses on technologies for performance analysis and tuning being developed by CScADS at Rice University. The paper describes two enabling technologies: a suite of performance tools for pinpointing scalability bottlenecks and analyzing node performance, and a tool for optimizing loop nests for better node performance. We demonstrate the utility of these tools by describing our preliminary experiences using these tools for analysis and tuning of the performance of S3D [1], a turbulent combustion code under development at Sandia National Laboratory. Section 2 describes our tools and Section 3 describes our experiences using them to analyze and tune the performance of S3D. Section 4 offers some preliminary conclusions and briefly outlines our future plans.

2. Enabling technologies

As part of the research and development of enabling technologies for high performance computing in CScADS, Rice University is developing HPCTOOLKIT [2]—a suite of tools for rapidly pinpointing performance bottlenecks in large-scale parallel codes, and LoopTool [3]—a compiler-based tool for optimizing loop nests in Fortran codes. We briefly describe these tools in turn.

2.1. Tools for pinpointing performance bottlenecks

HPCTOOLKIT consists of components for measuring the performance of optimized executables without adding instrumentation, analyzing application binaries to understand the structure of loop nests and inlined code that they contain, correlating performance measurements with program structure, and presenting performance data in a top-down fashion. We designed HPCTOOLKIT to:

- *Work at binary level for language independence.* This enables HPCToolkit to support measurement and analysis of multi-lingual codes with external binary-only libraries.
- *Profile rather than adding code instrumentation.* Profiling is less intrusive than code instrumentation and also requires only very modest data volume, which makes this approach practical for use on the US Department of Energy's leadership-class computer systems.
- *Collect and correlate multiple performance metrics.* Performance problems typically can't be diagnosed with only one species of event.
- *Compute derived metrics to aid analysis.* Synthetic metrics, such as memory bandwidth consumed, often provide insight for tuning.
- *Support top-down performance analysis.* HPCTOOLKIT's `hpcviewer` user interface presents performance data hierarchically, which helps one cope with application complexity.

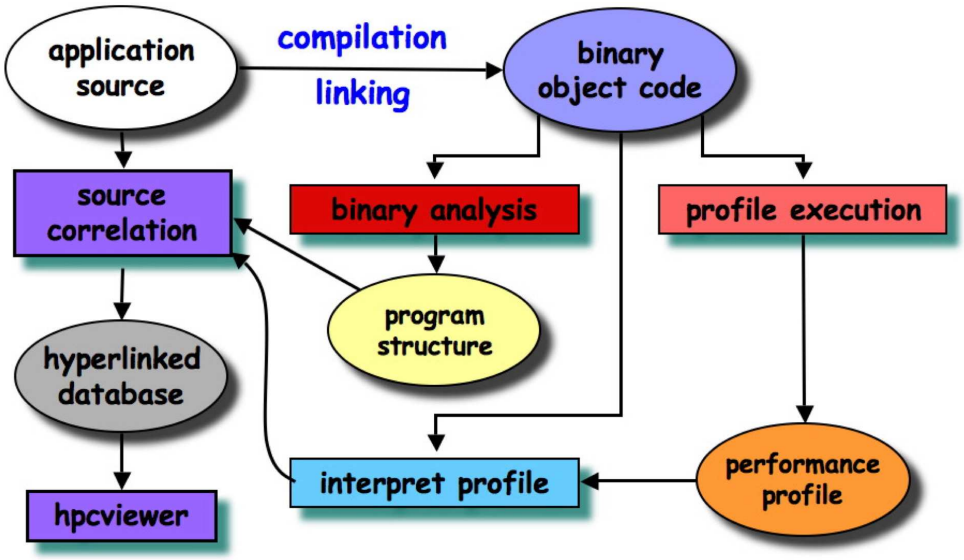

Figure 1. Overview of HPCTOOLKIT tools workflow.

- *Aggregate events for loops and procedures.* Samples based on hardware events cannot always be precisely attributed at the instruction level. Aggregating measurements for loops and procedures yields useful results despite fine-grain misattribution.

Using HPCTOOLKIT to analyze the performance of an application involves a sequence of steps as shown in Figure 1. One begins by launching an optimized application binary with one of HPCTOOLKIT's profiling tools. These tools can be used to profile either serial or parallel applications. While an application executes, the profiling tool records information about where in the execution various types of costs (e.g. time, cache misses, floating point operations, etc.) were incurred. Correlating these profiles with the static structure of fully-optimized binaries requires a mapping between object code and its associated source code structure. HPCTOOLKIT's `bloop` tool supports this correlation process by recovering information about procedures, loop nests, and inlined code from application binaries. To find loop nests within an executable, `bloop` decodes the executable's machine instructions, computes a control flow graph (CFG) for each procedure, and then analyzes the structure of the CFG to identify cyclic paths.

HPCTOOLKIT supports two types of profiling. The first is *flat* profiling, which involves sampling a processor's program counter while an application executes and recording a histogram that represents the set of program counter locations at which an event trigger, such as a timer interrupt or a cache miss, occurred. The second is *call path* profiling, which uses timer-based sampling to attribute execution time and waiting to calling contexts [4, 5]. A calling context for an event represents the set of procedure frames active when the event occurred. Our call path profiler collects profiles in the form of a *calling context tree* (CCT) [6]. In a CCT, the path from each node to the root of the tree represents a distinct calling context. Sample counts attached to each node in the tree associate execution costs with the calling context in which they were recorded. After post-mortem processing, CCTs contain three types of nodes: procedure frames, call sites and simple statements.

By scaling and differencing call path profiles for two executions of a parallel program on different numbers of processors, we can pinpoint lines in the program, along with the calling context in which they were executed, that suffer from scalability bottlenecks. A more detailed description of using HPCTOOLKIT for scalability analysis can be found elsewhere [7]. In this paper, we focus on applying HPCTOOLKIT's profiler to analyze node performance.

2.2. A tool for tuning application node performance
LoopTool is a compiler-based tool that helps expert human programmers improve the performance of Fortran loop nests by applying complex patterns of transformations to tailor the loop nests for a target microprocessor [3]. LoopTool automates the application of well-known source-to-source transformations that improve data reuse at various levels of the memory hierarchy, adjust instruction mix, and generate code that can be scheduled more efficiently by a conventional Fortran compiler. To use LoopTool, one takes a Fortran procedure and annotates the code with directives that specify a transformation recipe. LoopTool then applies the recipe to perform both the explictly-specified transformations along with other supporting transformations that need not be specified explicitly. Here, we briefly outline several of the key transformations supported by LoopTool.

Controlled multi-level loop fusion. LoopTool performs multi-level loop fusion by adjusting the alignment of statements in different loops relative to a common iteration space. Loop fusion is useful for improving reuse of data in cache by transforming multiple loop nests that access the same data into a single loop nest. Guided by user directives that specify exactly which loops to fuse, LoopTool verifies the legality of fusion before performing the transformation.

Scalarization. Scalarization transforms a computation specified using Fortran 90 array notation into a loop nest that iterates over each element in the index space and performs the computation elementwise. LoopTool performs scalarization so that the resulting loops can be optimized with other loop transformations such as fusion.

Unroll-and-jam. Applying unroll-and-jam to a loop nest unrolls an outer loop and then fuses resulting copies of the loop it encloses. Applying this transformation to a loop nest can bring together accesses to the same data by different outer loop iterations; this transformation can improve data reuse in cache.

Array contraction. If both the definition and all uses of a temporary array fall in the same loop nest after fusion, often LoopTool can automatically reduce the storage footprint by applying array contraction if it can prove that only a subset of the values need to be live simultaneously. Reducing a computations storage footprint enhances data reuse and can reduce the memory bandwidth required if the reduced-size array fits into cache at some level.

Loop unswitching. Unswitching a loop means hoisting a conditional within a loop nest out of one or more levels of enclosing loops and creating a custom version of the loop nest for the true and false branches of the conditional. By creating condition-free loop bodies, unswitching enables instructions to be scheduled more effectively.

Guard-free core generation. When fusing multiple loop nests that represent different but overlapping iteration spaces into a single loop nest, the resulting loop will require conditionals inside the loop to execute each statement instance only on the proper iterations. If you specify that you desire a guard free core, LoopTool will compute a subset of the fused iteration space for which no conditionals are necessary and transform the single loop nest into the sequence of loop nests necessary to realize the guard free core.

In the next section, we describe how we used LoopTool to increase the performance of a data-intensive loop nest of the S3D combustion code (described in the next section) by a factor of 2.94x. Figure 4 shows a loop nest annotated with LoopTool directives. Figure 5 shows a cartoon that explains how LoopTool transforms the code. Both of these figures are explained in more detail in the next section.

3. An application case study: S3D

S3D is a state-of-the-art massively-parallel combustion simulation code being developed at Sandia National Laboratory's Combustion Research Facility by the SciDAC project "Terascale Simulation of Turbulent Combustion" (PI: Jaqueline H. Chen, SNL). S3D performs direct numerical simulation (DNS) of turbulent combustion to solve Navier-Stokes equations for compressible, reacting flows [1]. The code uses a hierarchy of molecular transport models, and simulates detailed chemistry along with fuel sprays, radiation, and soot.

S3D uses DNS to study the micro-physics of turbulent reacting flows. DNS provides full access to time resolved fields and physical insight into chemistry turbulence interactions. An aim of simulations with S3D is to develop and validate reduced model descriptions that can be used in macro-scale simulations of engineering-level systems. The S3D code is important to the Department of Energy's Office of Science: it was selected for an FY07 INCITE award of 6 million CPU hours on the Cray XT3/4 at the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. It has also been selected as a Tier 1 pioneering application for the 250TF system being installed at NCCS.

The S3D code decomposes a 3D data volume among a collection of processes organized as a 3D logical mesh. All processes have same number of grid points and same computational load. Processes communicate using MPI message passing, specifically non-blocking sends and receives of coarse-grain messages. The bulk of inter-processor communication during execution occurs between processes that are nearest neighbors in the 3D logical mesh. All-to-all communication is only required for monitoring and synchronization ahead of I/O. The communication to computation ratio in S3D is small since computation is proportional to each processor's 3D volume of data, whereas communication is proportional to its surface area.

In February 2007, the S3D application team provided a version of the code to a "Tiger Team" that is part of the outreach activities in the SciDAC-2 Performance Engineering Research Institute. To investigate the node performance of S3D, we studied a single-processor execution of a pressure wave test on a 50^3 domain. We performed our experiments on a Cray XD1, whose nodes are populated with 2.2 GHz Opteron 275 processors and DDR 400 memory, which provides them with 6.4 GB/s of memory bandwidth. The nodes of the Cray XD1 serve as a proxy for the dual-core Opteron nodes of the NCCS Cray XT3/4. For the model problem, the code achieved .305 FLOPs/cycle, which represents 15% of peak. The question the Tiger Team was charged with answering is whether this level of performance can be improved, and if so, how. In the rest of this section, we demonstrate the utility of the enabling technologies being developed under CScADS for analysis and tuning of scientific programs. We explain how we applied HPCTOOLKIT to the application to understand its performance and how we applied LoopTool to improve application performance.

To understand the node performance of S3D in more detail, we collected both flat and call path profiles using HPCTOOLKIT. Figure 2 shows a view of a timer-based call path profile of a single-processor execution of S3D on a Cray XD1 presented using HPCTOOLKIT's `hpcviewer`. The source pane at the top of the display shows the source code for a loop over a 5-dimensional data structure. This loop nest was the most costly one in the application. Two loops over the direction and the number of species appear explicitly in the source code. Other 3-dimensional loops are implicit in the Fortran 90 array notation. The lower part of the display shows the attribution of costs to the loops in the loop nest in the enclosing `computespeciesdiffflux` routine. The 5D loop nest beginning on line 735 accounts for 11.3% of the total execution time of the program. The lower part of the display also explicitly shows the loops implicit in the Fortran 90 vector statements; the presence of these loops was recovered by HPCTOOLKIT's `bloop` in its analysis of the S3D executable. HPCTOOLKIT is unique in its ability to provide an accurate breakdown of costs among nested loops, including compiler-generated loops such as those here that represent the implementation of Fortran 90 vector statements.

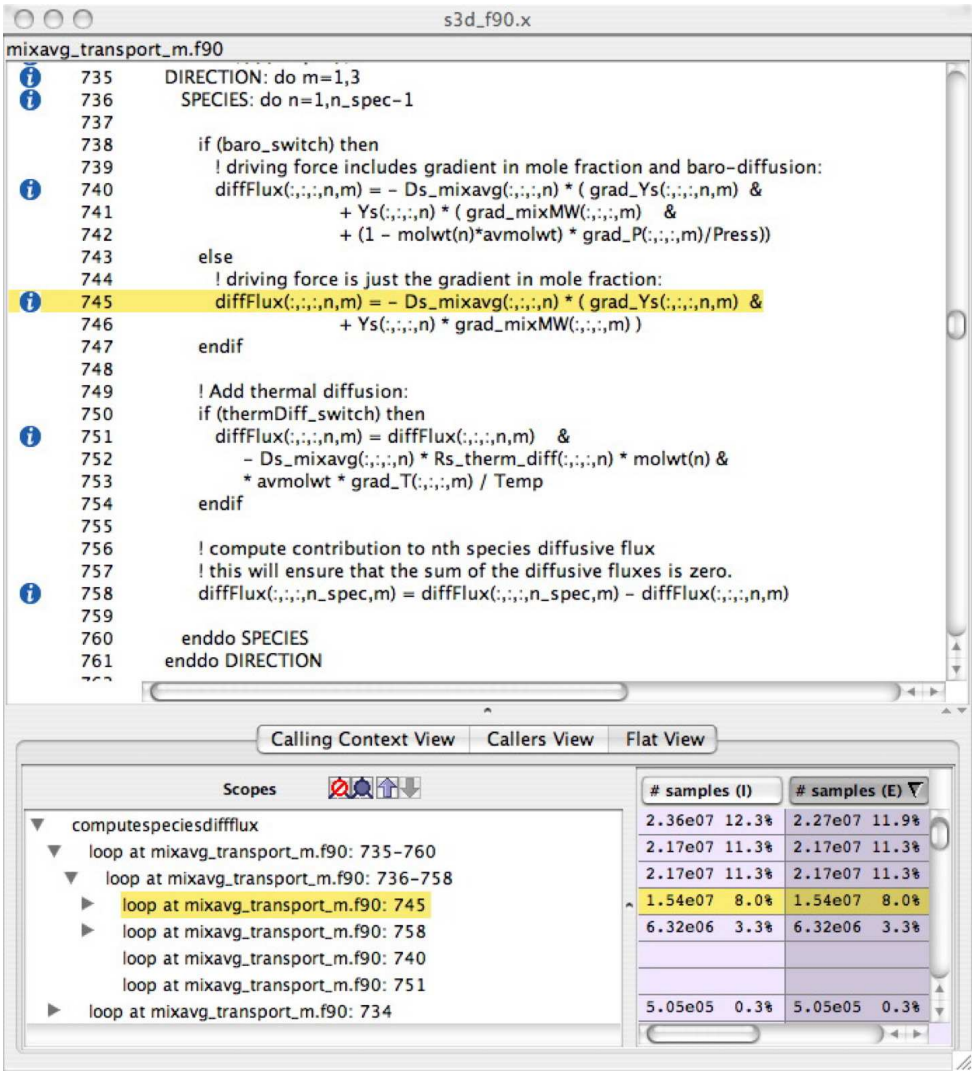

Figure 2. `hpcviewer` displaying a flat view of a timer-based call path profile for `S3D`.

Understanding how much opportunity exists for improving the performance of the loop nest shown in Figure 2 requires a more detailed analysis based on information available from the Opteron processor's hardware performance counters. We used HPCTOOLKIT to collect additional information to gain some more detailed insight into the performance of this loop nest. The metric pane in Figure 3 displays both measured metrics (cycles, instructions, FP instructions, and L1 cache accesses) and a derived wast metric for the loop nest. A relative waste metric was also computed, but its column is not shown in the figure. In this display, the cycles displayed for the loop at line 735 account for 10.3% of the execution time; this differs slightly from the 11.4% of samples attributed to the loop in the previous figure because the cycles metric shown here only accounts for time spent executing, whereas the samples metric in Figure 2 accounts for stall time as well. The waste column in the metric pane represents the difference between the theoretical peak number of FLOPs possible on an Opteron 275 core and the number of actual FLOPs executed, i.e., $(2 \times \text{CYCLES}) - \text{FLOPs}$. This metric tells us how much unrealized opportunity for floating point computation is associated with each context. Here, program scopes are rank ordered by the waste column in the metric pane. The

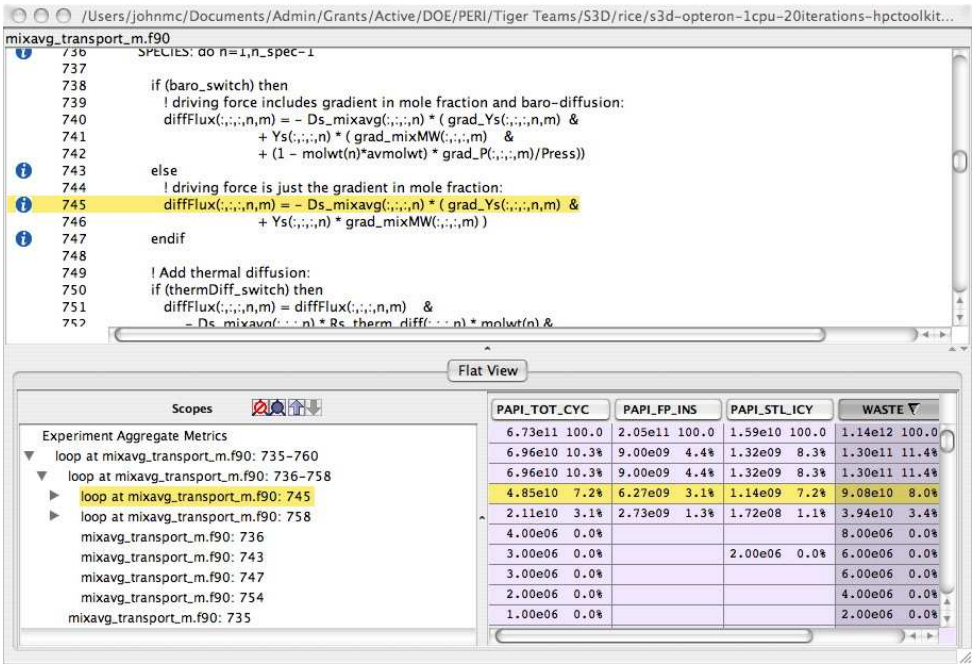

Figure 3. `hpcviewer` displaying a loop-level view of a flat hardware performance counter profile for `S3D`.

relative waste metric (not shown) indicates that the untuned loop nest achieves only 4% of the theoretical peak performance.

Study of the loop nest in Figure 3 reveals several unexploited opportunities for data reuse. The `diffFlux` array slice being computed in the first vector statement is reused in subsequent statements. Within individual vector statements there are opportunities for reuse across different iterations of the SPECIES and DIRECTION loops since many array references lack `m` and `n` subscripts. With careful application of loop transformations, these opportunities for data reuse can be exploited.

Figure 4 shows the loop nest of interest decorated with a LoopTool code transformation recipe. The recipe indicates that (a) two conditionals should be unswitched out of the SPECIES and DIRECTION loops to create four customized loop kernels (one for each switch setting), (b) after scalarization, loop nests for all of the 3D vector computations should be fused at all levels, (c) the DIRECTION loop should be unrolled completely and jammed inside the innermost loop of the 5D loop nest, and (d) the SPECIES loop should be unrolled by two, with pairs of its iterations jammed into the innermost loop. Based on these directives, LoopTool produces four customized loop nests.

Figure 5 shows the effect of having LoopTool apply the code transformation recipe shown in Figure 4. The left side of the figure shows a diagram that represents the structure of the original code. The green and brown arcs represent the direction and species loops. The two conditionals are shown explicitly, though the predicate `baro_switch` and `thermDiff_switch` have been abbreviated. The thick red, purple, blue, and black lines represent the Fortran 90 array statements in the original code.

The right side of the figure shows a diagram that represents the structure of the code after LoopTool applied the transformation recipe shown in Figure 4. At the highest level, the transformed code now consists of four loop nests. Unswitching the two conditionals out of the loop original loop nest creates four loop nests, each customized for a particular setting of the

```
!dir$ uj 3
  do m=1,3 ! DIRECTION
!dir$ uj 2
    do n=1,n_spec-1 !SPECIES
!dir$ unswitch 2
      if (baro_switch) then
        ! driving force includes gradient in mole fraction and baro-diffusion:
!dir$   fuse 1 1 1
        diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
                              + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m)    &
                              + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Press))
      else
        ! driving force is just the gradient in mole fraction:
!dir$   fuse 1 1 1
        diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
                              + Ys(:,:,:,n) * grad_mixMW(:,:,:,m))
      endif
      ! Add thermal diffusion:
!dir$ unswitch 2
      if (thermDiff_switch) then
!dir$   fuse 1 1 1
        diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m) - Ds_mixavg(:,:,:,n) *
            Rs_therm_diff(:,:,:,n) * molwt(n) * avmolwt * grad_T(:,:,:,m) / Temp
      endif
       ! compute contribution to nth species diffusive flux
       ! this will ensure that the sum of the diffusive fluxes is zero.
!dir$       fuse 1 1 1
      diffFlux(:,:,:,n_spec,m) = diffFlux(:,:,:,n_spec,m) - diffFlux(:,:,:,n,m)
    enddo ! SPECIES
  enddo ! DIRECTION
```

Figure 4. An S3D loop nest annotated with a transformation recipe for CScADS's LoopTool.

switches. For instance, the first loop nest in the transformed code only contains lines representing the red, blue, and black statements that will be executed when the two conditionals are true. The other loops have similarly been customized based on the settings of the enclosing conditionals. The gray arcs in the transformed code represent the loops over the first three dimensions of the `diffFlux` array that arise when LoopTool scalarizes the Fortran 90 array notation. Within each of the four customized loop nests, all of the colored lines representing statement instances have been fused into a single set of triply nested loops as specified by the fusion directives. The arc for the green `m` loop is not present in the transformed code. Instead, the code has been unrolled three times. Replication of the body of the inner loop two extra times is indicated by the green shading underneath the extra copies of the statement instances in the inner loop. Similarly, the brown `n` loop has been unrolled by two, which causes an orthogonal duplication of each of the three copies of the innermost loop body present after unrolling the `m` loop. This additional duplication is shown by the left-to-right duplication of statements shaded in brown. This figure conveys some of the complexity of the resulting code following LoopTool's transformations. Not shown are loops for handling the case when the value of `nspec` is even. The LoopTool-generated code runs 2.94 times faster than the original. This change alone reduced the entire programs execution time by 6.8% for a 50^3 problem size.

Further analysis of S3D's node performance with HPCTOOLKIT, tuning loops with LoopTool,

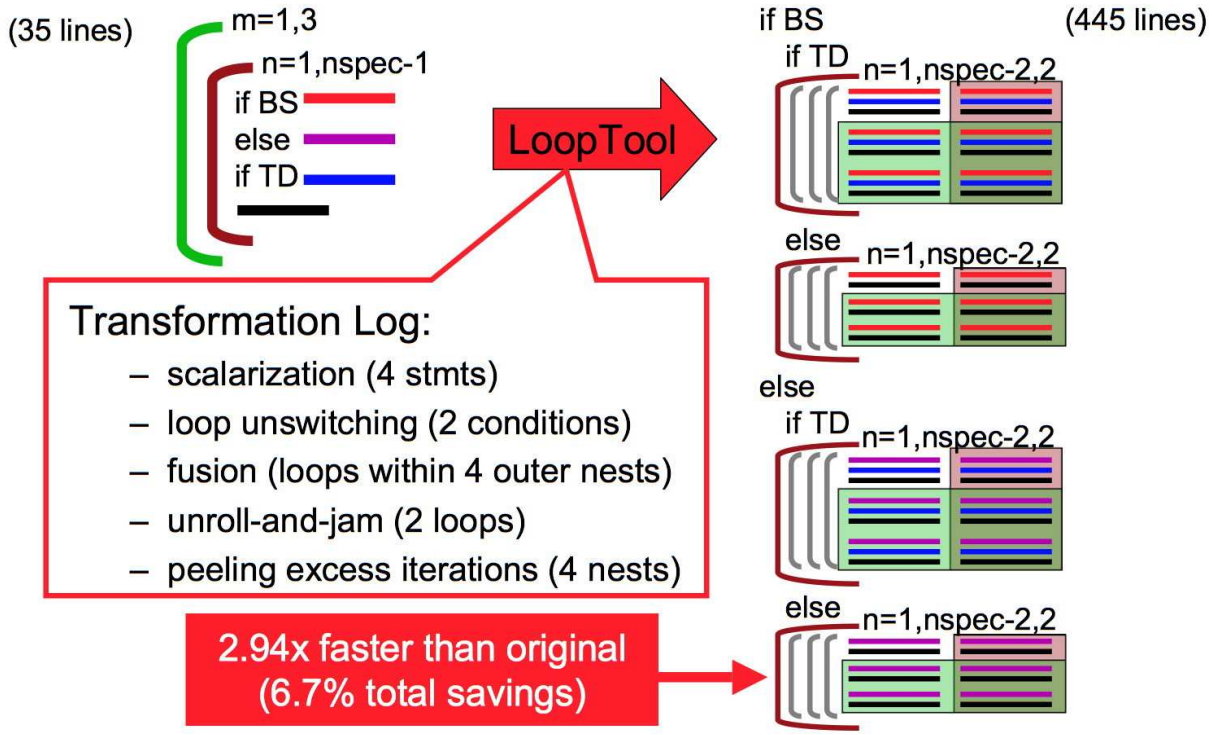

Figure 5. Using LoopTool to optimize S3D's diffusive flux computation on a 50^3 domain.

and some manual adjustment of procedure argument passing conventions yielded an aggregate improvement of roughly 12.7% for the 50^3 problem size.

4. Conclusions and future plans

The analysis and tuning of S3D described in this paper illustrates several points. First, analysis and tuning of data-intensive scientific applications for microprocessor-based parallel systems is difficult. Our example showed that one can significantly improve the performance of loop nests by improving memory hierarchy utilization. For the loop nest that was the focus of our tuning efforts, tuning nearly tripled performance. However, for complex scientific applications such as S3D, time is often spread out over many loops. Significantly improving node performance in such cases requires methodically applying analysis and tuning to each loop nest until no further improvements are possible. Second, achieving peak performance with data-intensive scientific codes on microprocessor-based systems is simply not possible; the best one can do is tailor one's code to avoid unnecessary performance losses due to missed opportunities for optimization. Third, our case study with S3D has shown that HPCTOOLKIT and LoopTool are powerful tools for analysis and tuning of scientific applications.

Plans for the Rice University team within CScADS call for enhancing and deploying performance measurement and analysis tools for the leadership class systems, improving compiler technology for scientific programs, and exploring compiler and runtime support for next-generation programming models for scalable parallel systems. As we move forward, we aim to maintain a close relationship with application teams both to better understand their needs for enabling technologies, and to assist them in tuning their codes for efficient execution on leadership-class platforms. For the near term, we will continue work with the S3D application team and also will work with several of the fusion application teams to improve the performance of their particle-in-cell simulations.

Acknowledgments
Jacqueline Chen of Sandia National Laboratory provided us with access to the S3D application. Michael Fagan, Mark Krentel, and Nathan Tallent have made recent contributions to HPCToolkit. Apan Qasem helped enhance LoopTool to support transformation of S3D. Yuan Zhao and Apan Qasem contributed to the performance studies of S3D. Guohua Jin and Gabriel Marin have begun analysis of GTC.

References
[1] Monroe D 2002 *SciDAC Review* URL `http://www.scidacreview.org/0602/html/combustion.html`
[2] Mellor-Crummey J, Tallent N, Fagan M and Odegard J 2007 *Proceedings of the Cray User's group meeting* (Seattle, WA) URL `http://www.cs.rice.edu/ johnmc/papers/hpctoolkit-cug-2007.pdf`
[3] Qasem A, Jin G and Mellor-Crummey J 2003 Improving performance with integrated program transformations Technical Report TR03-419 Department of Computer Science, Rice University
[4] Froyd N, Mellor-Crummey J and Fowler R 2005 *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing* (New York, NY, USA: ACM Press) pp 81–90 ISBN 1-59593-167-8
[5] Froyd N, Tallent N, Mellor-Crummey J and Fowler R 2006 *GCC Summit '06: Proceedings of the GCC Developers' Summit, 2006* pp 21–36
[6] Ammons G, Ball T and Larus J R 1997 *SIGPLAN Conference on Programming Language Design and Implementation* pp 85–96 URL `citeseer.ist.psu.edu/ammons97exploiting.html`
[7] Coarfa C, Mellor-Crummey J, Froyd N and Dotsenko Y 2007 *ICS '07: Proceedings of the 21st annual international conference on Supercomputing* (New York, NY, USA: ACM Press) pp 13–22 ISBN 978-1-59593-768-1