PERI Auto-Tuning

Jacqueline Chame[4], Chun Chen[4], Jack Dongarra[5], Mary Hall[4], Jeffrey K. Hollingsworth[3], Paul Hovland[1], Shirley Moore[5], Keith Seymour[5], Jaewook Shin[1], Ananta Tiwari[3], Sam Williams[2], Haihang You[5], David H. Bailey[2]

[1]Argonne National Laboratory, Argonne, IL 60439
[2]Lawrence Berkeley National Laboratory, Berkeley, CA 94720
[3]University of Maryland, College Park, MD 20742
[4]USC/ISI, Marina del Rey, CA 90292
[5]University of Tennessee, Knoxville, TN 37996

E-mail: mhall@isi.edu

Abstract. The enormous and growing complexity of today's high-end systems has increased the already significant challenges of obtaining high performance on today's equally complex scientific applications. Application scientists are faced with a daunting challenge in tuning their codes to exploit performance-enhancing architectural features. The Performance Engineering Research Institute (PERI) is working towards the goal of automating portions of the performance tuning process. This paper describes PERI's overall strategy for auto-tuning tools, and recent progress in both building auto-tuning tools and demonstrating their success on kernels, some taken from large-scale applications.

1. Introduction

As we are enter the era of petascale systems, as with earlier high-end systems, there is likely to be a significant performance gap between peak and sustained performance on the petascale hardware. Historically, the burden of achieving high performance on new platforms has largely fallen on the application scientists. To relieve application scientists of this burden, we would like to provide performance tools that are (largely) automatic, a long-term goal commonly called *auto-tuning*. This goal encompasses tools that analyze a scientific application, both as source code and during execution, generate a space of tuning options, and search for a near-optimal performance solution. There are numerous challenges to fully realizing this vision, including enhancement of automatic code manipulation tools, automatic run-time parameter selection, automatic communication optimization, and intelligent heuristics to control the combinatorial explosion of tuning possibilities. On the other hand, we are encouraged by recent successful results such as ATLAS, which has automatically tuned components of the LAPACK linear algebra library[1]. We are also studying techniques used in the highly successful FFTW library[2] and several other related projects[3-6].

The Performance Engineering Research Institute (PERI) is formalizing a performance tuning methodology used by application developers and automating portions of this process. Auto-tuning is one of the three aspects of performance tuning that are the focus of PERI, in addition to performance modeling and application engagement. In the context of these three aspects, the goal of PERI is to migrate automatic and semi-automatic prototypes into practice for a set of important applications.

The remainder of this document focuses on the PERI auto-tuning strategy (Section 2), recent progress in developing common interfaces to auto-tuning tools (Section 3), and tool infrastructure and experimental results (Section 4), followed by a conclusion.

2. PERI Auto-tuning Conceptual Diagram

Figure 1 provides a conceptual diagram of auto-tuning in PERI. Several phases are shown, as described here, but this document focuses on Transformation, Code Generation and Off-line Search.

1. *Triage.* This step involves performance measurement, analysis and modeling to determine whether an application has opportunities for optimization.
2. *Semantic analysis.* This step involves analysis of program semantics to support safe transformation of the source code. The analyses include traditional compiler analyses to determine data and control dependences and can exploit semantic information provided by the user through annotations or information about domain-specific abstractions.
3. *Transformation*. Transformations include traditional optimizations such as loop optimizations and in-lining, as well as more aggressive data structure reorganizations and domain-specific optimizations. Transformations such as tiling may be parameterized to allow for input size and machine characteristic tuning.
4. *Code generation.* This phase produces a set of possible implementations to be considered.
5. *Offline search.*. Offline search entails running the generated code and searching for the best-performing implementation. The search process may be constrained by guidance from a performance model or user input.

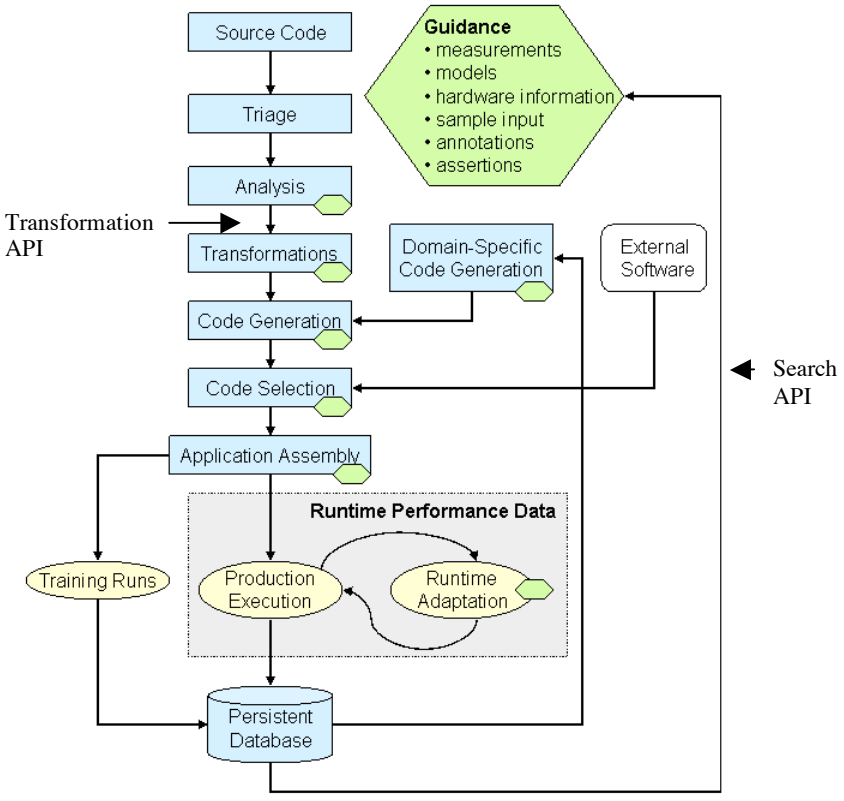


Figure 1: Flow diagram of the auto-tuning process

6. *Application assembly.* At this point, the optimized code components are integrated to produce an executable code, possible including instrumentation and support for dynamic tuning.
7. *Training runs*. Training runs involve a separate execution step designed mainly to produce performance data for feedback into the optimization process.
8. *Online adaptation.* Finally, optimizations may occur during production runs, especially for problems or machines whose optimal configuration changes during execution.

3. Evolving an Auto-tuning System through Common Interfaces

The challenges in automating the performance tuning process require that the following three issues be addressed: 1) The number of code variants for a complete application can be enormous. Strategies to avoid code explosion and to judiciously select what transformation techniques to apply to different sections of the application code are needed to keep the tuning time at manageable levels.
2) As the number of tuning parameters increases, the search space becomes high dimensional and exponential in size. Search algorithms that can cope with exponential spaces and deliver results within a few search iterations are needed. 3) A metric that measures and isolates the performance of a section of code being optimized within an application is needed to accurately guide search algorithms.

Within PERI, there are five different research groups working on developing auto-tuning *tools* to address these issues. These projects have complementary strengths and can, therefore, be brought together to develop an integrated auto-tuning *system*. Towards that end, we are working to develop a common framework to allow auto-tuning tools to share information and search strategies. Through common APIs, we can evolve an auto-tuning system that brings together the best capabilities of each of these tools, and also engage the broader community of tool developers beyond PERI researchers.

We have focused development of the interfaces on two portions of the auto-tuning process. Any compiler-based approach will apply code transformations to rewrite application code from its original form to one that more effectively exploits architectural features such as registers, caches, SIMD compute engines, and multiple cores. Commonly used code transformations include loop unrolling, blocking for cache, and software pipelining. Thus, we are designing a *transformation API* that will be input to the Transformation box in Figure 1. This API provides a *transformation recipe* that describes how to transform original source into an optimized source representation. By accepting a common transformation recipe, the auto-tuning system permits code transformation strategies derived by PERI compilers and tools (or users) to be implemented using any transformation and code generation tools, such as Chill (USC/ISI), LoopProcessor (LLNL) and POET (UTSA). The API supports the specification of unbound transformation parameters that are then tuned using search algorithms. The initial draft of the API includes a naming convention for specifying language constructs in source code and code transformations available in Chill and POET.

A *search API* provides input into the empirical optimization process of running a series of experiments on actual hardware to determine the best optimized implementation. The search API allows the auto-tuning tools to exchange information about their available tuning options and constraints on the search space, and to plug-in different search algorithms. The common framework will support both auto-tuning using training runs (and re-compilation) along with continuous optimization during production runs. For the search API, we are working on developing a simple and extensible language that standardizes the parameter space representation. Using the language, developers and researchers can expose tunable parameters to tuning frameworks. Relationships (ordering, dependencies, constraints and ranking) between tunable parameters can also be expressed.

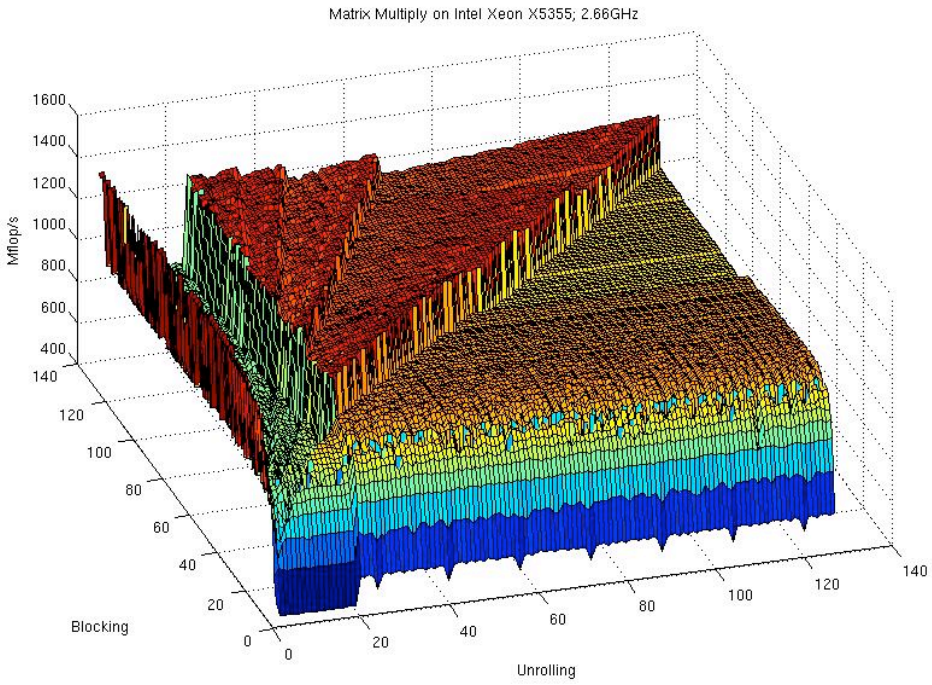
To understand the requirements of integrating auto-tuning tools developed independently by PERI researchers, we have been engaging in the integration of several PERI tools. Researchers at the University of Tennessee have integrated auto-tuning search (described below) with the ROSE LoopProcessor (LLNL) and the POET code generator (UTSA). Similarly, an initial integration of the Active Harmony system (UMD) and the Chill transformation framework (USC/ISI) is providing experience in how to effectively integrate these separate tools into an auto-tuning system.

4. Auto-tuning Infrastructure and Experimental Results

We now describe recent performance results from applying auto-tuning to dense and sparse linear algebra kernels. The dense linear algebra experiments use our prototype compiler and code-generation tools, while the more difficult-to-analyze sparse linear algebra experiments use a library-based auto-tuning approach. This section describes these results and the tool infrastructures used to derive them.

Code generation and empirical search. In conjunction with the previously-described APIs, a goal in PERI is to easily substitute different code generators and search engines in the auto-tuning process. To that end, researchers at the University of Tennessee have developed a simple syntax for specifying the way to generate code to perform the evaluation of the code variants. To permit switching search techniques, the application-specific aspects of the evaluation process are described by a simple specification of the search bounds and constraints. This system permits evaluation of various search techniques to determine which work best in an auto-tuning context. We have evaluated some classic techniques such as genetic algorithms, simulated annealing, and particle swarm optimization, as well as some more ad-hoc techniques such as a modified orthogonal search.

As a simple example of an optimization search space, Figure 2 shows optimization of square matrix-matrix multiplication (N=400), using an exhaustive search over two dimensions: block sizes up to 128 and unrolling up to 128. The x and y axes represent block size and unrolling amount, respectively, while the z axis represents the performance in Mflop/s of the generated code. In general, we see the best results along the blocking axis with a low unrolling amount as well as along the diagonal where blocking and unrolling are equal, but there are also peaks along areas where the block size is evenly divisible by the unrolling amount. The best performance was found with block size 80 and unrolling amount 2. This code variant ran at 1459 Mflop/s compared to 778 Mflop/s for the naive version compiled with gcc.



Compiler-based infrastructure. Researchers at USC/ISI are developing the TUNE auto-tuning compiler and its constituent Chill transformation framework, which provide a compiler-based infrastructure for general application code. Chill takes source code and a transformation recipe with bound parameter values as input. Using a polyhedral framework, Chill extracts a mathematical

representation of the iteration space and array accesses, and composes the transformations in the recipe through rewriting rules, to generate optimized code. A parameter sweep interface takes a transformation recipe with unbound parameter values and ranges and constraints on parameter values to derive a series of alternative implementations.

TUNE has been used to achieve hand-tuned levels of performance on dense linear algebra kernels such as matrix-matrix multiply, matrix-vector multiply and LU factorization for older architectures such as the SGI R10K, Pentium M and Pentium D. Recent work to optimize matrix-matrix multiply for newer architectures is promising, but show gaps between compiler-optimized and hand-tuned performance. Performance results for the Jacquard system at LBNL are 40% below hand-tuned performance, and we attribute the gaps to code generation for SSE-3, instruction scheduling and control of prefetch. Collaborating researchers at Argonne have been experimenting with manually scheduled inner loop bodies have achieved performance within 83% of peak and are developing automatic approaches to close this gap. On the Intel Core2Duo, we have achieved performance that is 28% below near-peak hand-tuned results. The primary performance gap appears to be control of software prefetch at the source code level.

Auto-tuning sparse-matrix vector multiply for multi-core architectures. In a study of auto-tuning a sparse matrix-vector multiply (SpMV) kernel, PERI researchers at Lawrence Berkeley National Laboratory compared performance on a dual-socket Intel Core2 Xeon, single- and dual-core AMD processors, a quad-core Opteron, a Sun Niagara2, and an IBM Cell Broadband Engine. This study found that these processors differed markedly in the effectiveness of various autotuning schemes on the SpMV kernel [6]. In two subsequent studies [7,8], these researchers analyzed similar but more sophisticated automatic optimizations for SpMV, the explicit heat equation PDE on a regular grid (Stencil), and a lattice Boltzmann application (LBMHD). This auto-tuning approach employs a code generator that produces multiple versions of the computational kernels using a set of optimizations with varying parameter settings. The optimizations include: TLB and cache blocking, loop unrolling, code reordering, software prefetching, streaming stores, and use of SIMD instructions.

The impact of each optimization varied significantly across architecture and kernel, necessitating a machine-dependent approach to automatic tuning in this study. In addition, detailed analysis revealed performance bottlenecks for each computation on the various systems mentioned above. The overall performance results showed that the Cell processor offers the highest raw performance and power efficiency for these computations, despite having peak double-precision performance and memory bandwidth that is lower than many of the other platforms in our study. The key architectural advantage of Cell is explicit software control of data movement between the local store (cache) and main memory, which is a major departure from conventional programming. In any event, these studies make it clear that there is still considerable room for improvement in automatic tuning methods for all of the candidate architectures.

Acknowledgment

This work was supported in part by the U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

References

[1] Whaley, C., Petitet, A., Dongarra, J. "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, Vol. 27 (2001), no. 1, pg. 3-25.

[2] Frigo, M., Johnson, S. "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, May 1998.

[3] Bilmes, J., Asanovic, K., Chin, C. W., Demmel, J. "Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology," *Proceedings of the*

International Conference on Supercomputing, Vienna, Austria, ACM SIGARCH, July 1997.

[4] Vuduc, R., Demmel, J., Yelick, K. "OSKI: A Library of Automatically Tuned sparse Matrix Kernels," *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, June 2005.

[5] Chen, C., Chame, J., Hall, M. "Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy," *Proceedings of the Conference on Code Generation and Optimization*, March, 2005.

[6] S. Williams, J. Carter, L. Oliker, J. Shalf, K. Yelick, "Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms," International Parallel & Distributed Processing Symposium (IPDPS) (to appear), 2008. WINNER: Best paper, applications track.

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," Proceedings of SC07, ACM/IEEE, November 2007.

[8] Samuel Williams, Kaushik Datta, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick, David Bailey, "PERI - Auto-tuning Memory Intensive Kernels for Multicore," available at http://crd.lbl.gov/~dhbailey/dhbpapers/scidac08_peri.pdf.