An Overview of the ECO Project

Jacqueline Chame, Chun Chen, Pedro Diniz, Mary Hall, Yoon-Ju Lee and Robert F. Lucas

University of Southern California/Information Sciences Institute
4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292
{jchame,chunchen,pedro,mhall,yoonju,rf lucas}@isi.edu

Abstract

In this paper, we describe a compilation system that automates much of the process of performance tuning that is currently done manually by application programmers interested in high performance. Our approach combines compiler models and heuristics with guided empirical search to take advantage of their complementary strengths. The models and heuristics limit the search to a small number of candidate implementations, and the empirical results provide the most accurate information to the compiler to select among candidates and tune optimization parameter values. The overall approach can be employed to alleviate some of the performance problems that lead to inefficiencies in key applications today: register pressure, cache conflict misses, and the trade-off between synchronization, parallelism and locality in SMPs.

The main focus of the paper is an algorithm for simultaneously optimizing across multiple levels of the memory hierarchy for dense-matrix computations. We have developed an initial compiler implementation, and present automatically-generated results on matrix multiply. Results on two architectures, SGI R10000 and Sun UltraSparc Ile, outperform the native compiler, and either outperform or achieve comparable performance as the ATLAS self-tuning library and the hand-tuned vendor BLAS library.

This paper describes other components of the ECO system, including supporting tools and experiments with programmer-guided performance tuning. This approach has provided a foundation for a general framework for systematic optimization of domain-specific applications. Specifically, we are developing an optimization system for signal and image processing that exploits signal properties, and we are using machine learning and a knowledge-rich representation can be exploited to optimize molecular dynamics simulation.

1 Introduction

The last decade has witnessed dramatic improvements in the effectiveness of compiler technology for high-performance computing. In spite of these achievements, there is plenty of anecdotal evidence that important science and engineering applications are nevertheless executing at only a small fraction of peak performance, often less than 10%, on systems ranging from PCs to ASCI supercomputers and computational grids. The sources of this inefficiency stem from trends in architecture, compilers and applications. On the architecture side, there is a growing gap between peak computation rate and memory and communication bandwidth across this whole range of systems. Thus, computational units are often stalled waiting on local memory accesses or communication with remote processors. Architectural and compiler solutions to combat this performance gap have resulted in systems of enormous complexity. As a result, statically predicting the impact of individual compiler optimizations and the aggregate impact of a collection of optimizations is becoming increasingly difficult for both compiler and application developers. Application complexity is also increasing dramatically. Scientific programs with several hundred thousand lines of code are now fairly common, problem sizes are scaling up dramatically, and applications are becoming increasingly irregular in terms of memory access behavior and parallelism.

The net effect of this increased hardware and software complexity is that programmers interested in high performance are spending more and more of their time on performance tuning. The process of manual performance tuning involves focusing in on a few key computational components of an application. For each component, the programmer derives a sequence of variants with different implementations for performing the computation. Each variant is first debugged, and then
its performance characteristics are evaluated. This lengthy process continues until the programmer either arrives at a suitable variant or, as is often the case, decides to give up on further performance tuning. New variants may be required for characteristically different input data sets, or as the application is ported to other platforms.

In this paper, we describe an application-compiler-runtime system we are developing that seeks to automate this performance tuning process as much as possible. Our approach is partially motivated by the successes of several recent self-tuning libraries and domain-specific systems that employ empirical techniques to systematically evaluate a collection of automatically-generated variants [7, 6, 3, 8]. Rather than trying to predict performance properties through analysis, variants are executed with representative input data sets so that performance can be measured and compared.

In our system, we will generalize the notion of empirical optimization so that it can be applied to full scientific applications. The key considerations in our system are how to make empirical optimization of full applications both practical and effective. Consider the ATLAS system, which automatically derives highly tuned implementations of matrix multiply; ATLAS takes a few hours to self-tune on a new platform. It is simply not feasible to spend several hours per loop nest tuning a several-hundred-thousand line program. Further, ATLAS uses a code generation strategy for deriving variants that is specifically tailored to matrix multiply, and is the result of decades of study of the performance properties of matrix multiply on modern architectures.

As part of a strategy for empirical optimization of applications, we must generalize the derivation and testing of code segment variants.

The remainder of the paper is organized as follows. Section 2 briefly describes our framework for model-guided empirical optimization of the memory hierarchy. Section 3 summarizes other research developments under the ECO project. Future research directions are discussed in Section 4 and Section 5 concludes the paper.

2 Model-guided empirical optimization of the memory hierarchy

Ideally, automatic tuning should achieve the results of evaluating and selecting among a large set of possible high-quality implementations of a piece of code. The set of possibilities can be prohibitively large, so we must prune this set so that only the most promising solutions are considered. One approach to limiting search is model-guided empirical optimization which combines the complementary strengths of model-based optimization and empirical search. Models and compiler heuristics can limit the search to a small number of the most promising candidate implementations, while empirical measurements provide the most accurate information to the compiler to select among candidates and tune optimization parameter values.

We have developed an approach to model-guided empirical optimization whereby application kernels are represented by a set of implementation variants, or alternative implementations of the kernels. Associated with each variant is a set of optimization parameters, integer variables whose values are determined empirically. The concept is to explore a broader set of alternative solutions than conventional compilers do, by generating a set of parameterized variants rather than deriving a single fixed solution.

In [1] we primarily focused on the complex problem of simultaneously optimizing for all levels of the memory hierarchy (registers, L1 and L2 caches, and TLB). This section presents a brief summary of our framework along with experimental performance results and a brief discussion of compile-time costs as compared to other approaches.

2.1 Framework

Our framework for model-guided empirical optimization of the memory hierarchy is organized into two main phases (Figure 1). In the first phase the compiler generates a set of parameterized code variants, based on static analysis and models. The second phase is a search among parameter values for each code variant, guided by models and heuristics.

Generate Parameterized Variants. In this phase the compiler uses traditional static analyses,
models and code transformations to generate a set of parameterized code variants. Among others, dependence analysis is used to determine the legality of code transformations, locality analysis to evaluate data reuse and select specific locality optimizations and register reuse analysis to estimate register pressure. The models include register, cache and TLB models and also incorporate various heuristics for those optimizations. These heuristics are also used in the second phase to prune the parameter search space. The code transformations considered are well-known compiler optimizations: loop permutation, unroll-and-jam, scalar replacement, tiling, data copying and prefetching. Code transformations that depend on parameter values, such as unroll-and-jam, are applied during the second phase, after the parameter values have been selected. Along with each code variant, the first phase also generates a set of constraints for the optimization parameters. These constraints are used in the second phase to guide and prune the search.

Here we have a brief discussion of how the code generation algorithm interacts with transformation modules and generates appropriate code variants and related parameters. The details of the algorithm can be found in [1]. The code generation algorithm systematically applies individual transformations based on analysis and models. During this process the algorithm generates code variants with unbound parameters, based on the characteristics of transformations. Table 1 shows the transformations and parameters used by the algorithm. The second and third columns contain a brief description of each transformation and its goals. The fourth column indicates whether a transformation results in more than one code variant. For example, for loop permutation the algorithm may generate multiple code variants, each with a different loop order, if it cannot decide which order is best statically. Other transformations, such as loop tiling, do not increase the number of variants, but result in the generation of code variants with unbound parameters, as illustrated in the table’s last column. The values of these parameters are determined later at the search phase. By carefully separating variants and parameters, we reduce the number of code variants needed to be generated and can take advantage of the relationship between those parameters to have an efficient search for each code variant.

**Search for Parameter Values.** In the second phase, a guided empirical search performs a series of experiments to derive the best parameter values for each code variant. In addition, code transformations that depend on parameter values, such as unroll-and-jam, are applied during this phase. The resulting code variants are then compiled and run on the target machine.

The search engine uses performance metrics collected by performance monitoring hardware and tools to evaluate the quality of a code variant with a given set of parameter values. The performance metrics might also be used to determine the next set of parameter values to be evaluated, that is, guide the search. Currently we use PAPI to collect performance data, and use processor cycles as the performance evaluation function for the search.

**2.2 Performance comparison**

We have integrated our strategy for model-guided empirical optimization in the Stanford SUFI compiler infrastructure and applied it to matrix multiply and Jacobi relaxation. Figure 2 shows a performance comparison of the optimized matrix multiply generated by our implementation (ECO) with the versions generated by the native compiler (Native), the vendor-supplied BLAS and the self-tuning library ATLAS, on an SGI R10000 and an UltraSpac Hc. Results for Jacobi can be found in [1].

On the SGI, the ECO version of matrix multiply shows stable behavior across the full range of problem sizes, as shown in Figure 2(a). Its performance ranges between 302 and 342 with an average of 333 MFLOPS, that is, 85% of the theoretical peak of 390 MFLOPS. The Native version suffers from severe cache conflict misses and has far more fluctuation in performance, ranging from 84 to 343 with an average of 308 MFLOPS. In addition, the performance of Native trails off for larger problem sizes due to very high TLB misses. ATLAS produces far more stable results than the SGI compiler, ranging from 246 to 316 with an average performance of 308 MFLOPS. PAPI results show that ATLAS has about 1.4 times L1 and L2 cache misses compared to those of ECO, but about three fourth load instructions and much less TLB misses. The performance of the hand-tuned Vendor BLAS is very close to that of ECO, ranging from 118 to 346 with and average of 334 MFLOPS, but it still has serious performance limitations for certain problem sizes. On the Sun UltraSpac Hc the average performance of the Native version is only 60 MFLOPS, as shown in Figure 2(b). ATLAS performs well in this architecture, ranging from 387 to 543 with an average of 517 MFLOPS. The performance of ECO ranges from 406 to 530 with an average of 506 MFLOPS. Again, the performance of Vendor BLAS is close to that of ECO, ranging from 368 to 512 with an average of 494 MFLOPS.
<table>
<thead>
<tr>
<th>Transformations</th>
<th>Definition</th>
<th>Goal</th>
<th>Variants</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop permutation</td>
<td>Change the loop order</td>
<td>Enable U&amp;J and Tiling + Reduce TLB misses</td>
<td>Different loop orders</td>
<td>-</td>
</tr>
<tr>
<td>Unroll and Jam</td>
<td>Unroll outer loops and fuse inner loops</td>
<td>Reuse in registers</td>
<td>-</td>
<td>Unroll factors</td>
</tr>
<tr>
<td>Scalar replacement</td>
<td>Replace array accesses with scalar variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiling</td>
<td>Divide iteration space into tiles</td>
<td>Reuse in cache</td>
<td>-</td>
<td>Tile sizes</td>
</tr>
<tr>
<td>Data copying (w/tiling)</td>
<td>Copy subarray into contiguous memory space</td>
<td>Avoid conflict misses + Avoid TLB thrashing</td>
<td>Yes/no on specific data structures</td>
<td>-</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Prefetch data into cache before actual references</td>
<td>Hide memory latency</td>
<td>-</td>
<td>Prefetch distances</td>
</tr>
</tbody>
</table>

Table 1. Transformation variants and parameters

2.3 Cost of empirical search

To put our approach into context with the other techniques we have compared against, we provide some measurements of the cost of empirical search.

Our current implementation searched 60 alternative implementations of Matrix Multiply on the SGI, which required roughly 8 minutes. On the Sun, it considered 44 points, which required 6 minutes. The Jacobi search was 94 points in 3 minutes on the SGI and 148 points in 5 minutes on the Sun. As compared to standard compile time, which was at most one or two seconds on both the SGI and Sun, this represents a fairly significant increase in cost, however it provides far better results that are comparable to the performance of hand-tuned codes on both architectures.

For Matrix Multiply, we can also compare against ATLAS search time for just the Matrix Multiply function. (This does not include the cost of a separate microbenchmarking search to determine architecturespecific properties, such as instruction-set features and usable register and cache capacity.) For a meaningful comparison with our compiler, we consider the ATLAS search time without the use of architectural defaults, which would have bypassed portions of the search. The ATLAS search for a Matrix Multiply implementation requires 35 minutes on the SGI and 14 minutes on the UltraSparc IIe.

We should also point out that the derivation of the hand-tuned vendor BLAS libraries can be considered a manual empirical search, which we assume requires on the order of days of a programmer’s time to try alternative versions and arrive at a highly tuned result.

3 Research summary

3.1 LS-DYNA

To demonstrate how a compiler can collaborate with the application programmer in model-guided empirical optimization, we presented a case study illustrating how software tools can be used to greatly accelerate the process of performance tuning, leading to better application performance as well as increasing productivity of programmers of high-end systems. To examine application-specific memory hierarchy behavior, we have experimented with unroll factors and tile sizes on the sparse solver from LS-DYNA, a widely used engi-
neering code, producing measurements on the SGI Origin. The programmer specified that these were parameters whose values should be determined by the compiler. In a previous publication, we presented a set of results where parameterized variants are generated and compared for the sparse solver in LS-DYNA [4]. The results demonstrate the sensitivity of the application’s performance to optimization parameters. Through this case study, we demonstrated the importance of developing automatic performance tuning support for performance-sensitive applications. In recent work, we have experimented with scheduling of tasks in OpenMP for key computations in LS-DYNA. We discovered that different scheduling algorithms lead to the best performance of LS-DYNA, depending on the phase of the computation. In some cases, the load is fairly balanced and static block scheduling of iterations leads to the best performance. In other cases, the workload is unbalanced, and as a result, dynamic scheduling leads to the best performance. Both these LS-DYNA experiments demonstrate the merit of using a code isolator in conjunction with empirical optimization of large application codes.

3.2 Code isolator

We developed a supporting tool for empirical optimization called a code isolator. Its purpose is to extract an executable code segment from a large application, so that a programmer or compiler tool can experiment with the code segment independent of the remainder of the computation. The code isolator not only separates the code segment from the program, but also derives representative inputs, values of parameters such as problem sizes, and machine state to be used in the empirical evaluation, such as the state of the memory hierarchy. In a previous publication, we focused on developing efficient techniques for capturing and setting machine state [5]. Recently, we have derived benchmark versions of the UMT2K application, and focused on compiler optimizations to reduce the input data for the isolated code and collect representative input data. While in earlier work we focused on isolating sequential code, we are now developing OpenMP benchmarks from OpenMP and mixed MPI/OpenMP codes.

3.3 Systematic empirical search

In [1] we demonstrate the promise of combining models and empirical search. Nevertheless, the heart of making such an approach general and practical involves gathering an understanding of how to perform such a search effectively, leading to a high-quality result while managing the execution time cost of the search. The field of Artificial Intelligence (AI) has developed various search techniques for solving complex, multi-parameter optimization problems, which are characterized by very large and rough parameter landscapes. In [2], we explore the search algorithm in an effort to develop a systematic and generalizable approach that goes beyond the specifics of our memory hierarchy optimization strategy. We discuss the suitability of several
AI search techniques to our optimization problem, and describe how to incorporate domain knowledge into a systematic search process.

4 Future research

4.1 Systematic application optimization with planning and machine learning

ECO has provided a foundation for optimization strategies that search among a set of possible application mappings for a high performance solution. Looking forward, we would like to develop a systematic approach for application optimization that enables future optimization goals to benefit from our approach. To do this, we require an optimization framework that decouples core technologies in programming environments, compilers and run-time tools that are a constant across different architectures, including specific optimizations, program analyses and mechanical code transformations. Applications can be viewed as workflows consisting of composable components to be mapped to machine resources. The remainder of the system will rely on the application programmer and tool developers to describe a range of alternative workflow implementations that are then searched automatically by tools to find a high-quality implementation. Incorporating cognitive search techniques and taking advantage of parallel resources, it is now feasible to explore a range of possible implementations—either offline or, in some cases, dynamically—and choose the most appropriate.

Porting applications to a new architecture will use the same core tools, but will repeat the automated search among possible implementations to find the solution that is optimal for the new environment in which the application will execute. Overall, such an approach will free the programmer from navigating a variety of complex tradeoffs in computation performance, reliability, area or memory requirements, and other demands of future applications. This initial project will focus on molecular dynamics simulation, for which there is a significant body of knowledge on the performance-precision tradeoff space.

4.2 Domain-specific, data-driven optimization of signal processing

A related project will develop a dynamic, data-driven application system for signal and image processing under severe resource constraints. We propose a multidisciplinary approach that optimizes from algorithm specification, to mathematical representation, to software and hardware (FPGA) implementation, based on properties of data and unique requirements of the environment and the target hardware device. The proposed approach will perform joint optimization across mathematical, software and hardware (system-on-a-chip FPGA) domains in a dynamic and data-driven fashion in that signal-processing transforms are tailored to algorithm requirements and input signals, for reduced distortion and increased compression. System implementations will be based on the best mathematical formulation of the problem coupled with automated selection of the best implementation among a space of alternatives, through the integration of models relating mathematical properties to implementation behavior. Both hardware and software optimization are treated in a unified way.

5 Conclusion

This paper has been an overview of the work on the ECO project over the last three years. Our work focuses on combining programmer input, models, empirical search and compiler analysis and code generation to automate the process of performance tuning on today’s complex hardware platforms. This work has provided a foundation for other optimization work on new projects. We are developing a broader optimization strategy that systematizes performance optimization of code components and application workflows using machine learning and a rich knowledge representation. We are also developing a domain-specific data-driven optimization system for signal and image processing, that performs optimization beginning with the mathematical representation of signals and their properties, and also co-optimizes software and hardware implementations.

References


