**Automatic Tuning of Whole Applications:**

*A LACSI Symposium Workshop*

**Overview and Abstracts**

October 17, 2006

**Workshop Coordinator**

Ken Kennedy, ken@rice.edu

**Workshop Abstract**

For many years, retargeting of applications for new architectures has been a major headache for high performance computation. As new architectures have emerged at dizzying speed, we have moved from uniprocessors, to vector machines, symmetric multiprocessors, synchronous parallel arrays, distributed-memory parallel computers, and scalable clusters. Over the past year, clusters based on multicore chips (with partially shared cache hierarchies) have further added to the complexity of computing platforms. Each new architecture, and even each new model of a given architecture, has required retargeting and retuning every application, often at the cost of many person-months or years of effort.

Over the past several years, compiler and library developers have turned to strategies for automating the process of tuning applications to new architectures, using large amounts of computation time to explore a space of different variants of the program and running each variant on the target architecture. One example of this strategy is Atlas, which uses in-advance tuning to generate versions of a computational linear algebra kernel that achieve high performance on different machines. If this approach can be extended more generally to components and whole programs, it would help avoid the enormous human costs involved in retargeting applications to different machines. A major research question in this area remains: given that the space of variants can be enormous, how can we reduce tuning time to manageable levels?

A number of research groups have been pursuing work in this area. In 2005, we held a well-attended workshop on autotuning at the LACSI Symposium. The slides and abstracts from that workshop can be found [here](#). This follow-on workshop at the 2006 LACSI Symposium will provide a venue for reports on progress over the last year, and presentations of new research not covered last year. The goals of the workshop are:

- To provide a forum for research groups working on automatic tuning to report on the status of their efforts and their future plans;
- To foster collaborations among different autotuning research groups and the potential users of tools that they may develop; and
- To initiate an activity to develop a standard set of benchmarks for use in this research.
Presentation Abstracts

Performance-Related Aspects of LANL ASC Codes

Hank Alme (LANL X-3-PC)

This talk will focus on two design details of two of the ASC codes in use at Los Alamos National Laboratory. First, we present some details on the algorithms on the SAGE eulerian hydro code used to treat multiple material mixed in a single cell. The other topic is the memory access pattern implied by the data structure design for unstructured meshes in the FLAG code, a hydro code implementing ALE. This talk will attempt to present some areas and spark discussion on where the auto-tuning community could provide advice, and how the lab could be better applying the tools and ideas in its code design efforts.

Programming by Sketching

Rastislav Bodik (UC Berkeley)

Sketching is a software synthesis approach where the programmer develops a partial implementation --- a program with holes called a sketch. The synthesizer then completes the sketch to make it behave like a separately provided specification. Holes typically stand for error-prone code fragments such as index expressions, lookup tables, or bitmasks.

I will present SKETCH, a language for sketching finite programs. Finite programs include many high-performance kernels notorious for requiring orchestration of minute details. Sketching can often synthesize these details, allowing the programmer to focus on the higher-level implementation strategy. As a case study, I will describe our implementation of the AES cipher.

In contrast to prior synthesizers, which had to be equipped with domain-specific rules, SKETCH reduces synthesis to generalized Boolean satisfiability. Consequently, our synthesizer is complete for the class of finite programs: it is guaranteed to complete any sketch in theory, and in practice has scaled to realistic programming problems.

Slides: PPT slideshow
Combining Models and Guided Empirical Search for Memory Hierarchy Optimization

Chun Chen, Jacqueline Chame, and Mary Hall (USC ISI)

This talk will describe an algorithm for simultaneously optimizing across multiple levels of the memory hierarchy for dense-matrix computations. 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. We will present performance results and discuss future directions. Notably our results on Matrix Multiply achieve comparable performance with ATLAS and vendor BLAS libraries.

Self Adapting Numerical Software Overview

Jack Dongarra

This presentation will review a number of efforts currently being pursued in the area of self adaptive software.

- Lapack For Clusters
- Dynamic applications choice depending on the data
- Fault tolerant linear algebra approaches
- Optimization techniques applied to highly parallel systems
- Collective communication optimization

Slides: PDF

Procedure-Level Performance Tuning of Whole Programs

Rudolf Eigenmann (Purdue University)

I will present the architecture and performance of an automatic performance tuning system. The system partitions a program into a number of tuning sections and finds the best combination of compiler optimizations for each section.

The key techniques include (i) a partitioning algorithms that splits a given program into appropriate tuning sections, (ii) a search method for the best optimization combination in a large space of possibilities, and (iii) techniques that can fairly compare two runs of a subroutine under different execution contexts.
The performance tuning process includes several pre-tuning steps that partition and instrument the program, followed by the actual tuning and the post-tuning assembly of the individually optimized parts. The system, called PEAK, achieves fast tuning speed by measuring a small number of invocations of each code section, instead of the whole-program execution time, as in previous solutions. Compared to these solutions, PEAK reduces tuning time from 2.19 hours to 5.85 minutes on average, while achieving similar program performance. PEAK improves the performance of the SPEC CPU2000 FP benchmarks by an average 12% over GCC O3, the highest optimization level, on a Pentium IV machine.

The current system implementation tunes programs during a training run, then freezes the optimization combination for the production runs. I will discuss opportunities to perform the optimizations fully dynamically. Also, the only optimizations being tuned are currently those exposed in the form of compiler options. In ongoing work, we are exposing to the automatic tuning capability additional, compiler-internal optimization parameters as well as optimization parameters of library routines.

Slides: [PDF](#)

**Performance Analysis of Divide and Conquer Algorithms for the WHT and FFT**

Jeremy Johnson (Drexel University)

This talk presents a family of algorithms for computing the Walsh-Hadamard transform (WHT), an important computation in signal and image processing related to the fast Fourier transform (FFT). For input size $N=2^n$, there are $O(7^n)$ algorithms. These algorithms all compute the WHT using exactly $N\log(N)$ arithmetic operations; however, they do have different memory access patterns and use varying amounts of recursion, iteration, and unrolled code. Despite the fact that all of the algorithms use exactly the same number of arithmetic operations, they exhibit vastly different performance. Intelligent search methods (dynamic programming and an evolutionary algorithm) have been used to empirically find good implementations. This works well in practice; however, it does not provide an understanding of why certain algorithms are better than others, nor does it give information about the search space.

Two performance models, one based on instruction counts and the other on cache misses, and introduced. The models by themselves do not accurately predict performance, however, they explain many aspects of the search space and can be used to guide the search. Moreover, each model can be described using parameterized recurrence relations, which lead to formulas for the number of instructions and cache misses. For each model,
it is possible to analytically determine the worst case, best case, and expected performance, along with the limiting distributions.

The techniques presented in the talk are directly applicable to the FFT and can be extended to other DSP transforms. Such a generalization, along with ideas for combining models and estimating runtime performance, will be discussed in the context of the SPIRAL (www.spiral.net) system for the automated generation and optimization of DSP transforms.

**In Search of Near-Optimal Optimization Phase Orderings**

Prasad Kulkarni, David Whalley (Florida State University)

The phase-ordering problem is a long standing issue for compiler writers. Varying the order of applying optimization phases can produce different code with potentially significant performance variation between the best and worst performing instances. However, there is no universal optimization phase order that will produce the best code for all applications since the optimal order depends on the function being compiled, the compiler, and the architectural characteristics. Moreover, finding an optimal optimization phase sequence appears to be impractical as the attempted optimization phase space can be huge for a even a single function.

A key insight to the phase ordering problem is that applying many different sequences of phases result in the same code being generated. In this presentation I will show that the actual optimization phase order space is not as large as earlier believed and can often be exhaustively enumerated for the optimization phases in our compiler. We analyze the performance of code generated by different phase orderings based on dynamic frequency measures of the function instances within the space, show that these dynamic frequency measures correlate well to simulator processor cycle counts, and that the optimal ordering for the optimization phases in our compiler with respect to simulated measurements can often be determined in a short period of time. We also analyze this space to capture probabilities regarding the relationships between phases and use these probabilities to significantly reduce the compilation time for a conventional compiler.

Slides: [PDF](#)
Using Machine Learning to Focus Iterative Optimization

Michael O’Boyle (University of Edinburgh)

Iterative compiler optimization has been shown to outperform static approaches. This, however, is at the cost of large numbers of evaluations of the program. This paper develops a new methodology to reduce this number and hence speed up iterative optimization. It uses predictive modeling from the domain of machine learning to automatically focus search on those areas likely to give greatest performance. This approach is independent of search algorithm, search space or compiler infrastructure and scales gracefully with the compiler optimization space size. Off-line, a training set of programs is iteratively evaluated and the shape of the spaces and program features are modeled. These models are learnt and used to focus the iterative optimization of a new program. We evaluate two learnt models, an independent and Markov model, and evaluate their worth on two embedded platforms, the Texas Instrument C6713 and the AMD Au1500. We show that such learnt models can speed up iterative search on large spaces by an order of magnitude. This translates into an average speedup of 1.26 on the TI C6713 and 1.27 on the AMD Au1500 in just 2 evaluations.

The Price of Cache-obliviousness

Keshav Pingali (University of Texas, Austin)

Cache-oblivious algorithms and data structures have been proposed as a methodology for writing portable, high-level, machine-independent programs whose performance is nevertheless competitive with that of cache-aware programs customized to particular machines. This methodology was instrumental in the development of the highly successful FFTW code, but FFT has little algorithmic reuse. How does the cache-oblivious methodology perform for problems with a lot of algorithmic reuse such as dense linear algebra computations?

In this talk, we describe an experimental study of cache-oblivious and cache-aware programs for dense linear algebra problems. Our results show that substantial effort may be required to produce cache-oblivious codes that perform well on modern machines, and that even highly optimized versions may not perform as well as cache-aware codes for the same problem. We conclude with some research directions that may help narrow the gap.

This is joint work with Kamen Yotov, Tom Roeder, Fred Gustavson and John Gunnels.
Generating Parallel Transforms Using Spiral

Markus Pueschel (CMU)

Spiral (www.spiral.net) is a program generation system for linear transforms such as the discrete Fourier transform, discrete cosine transforms, filters, and others. For a user-selected transform, Spiral autonomously generates different algorithms, represented in a declarative form as mathematical formulas, and implementation choices to find the best match to the given target platform. Besides the search, Spiral performs deterministic optimizations on the formula level effectively restructuring the code in ways unpractical at the code level.

In this talk we explain how Spiral generates efficient programs for parallel platforms focusing on shared-memory and, in particular, chip-multiprocessor systems (CMPs). As other optimizations, also parallelization is performed by a rewriting system that systematically modifies a formula to achieve load balancing and avoids false sharing. For example, for the DFT, the application of these rules produce a novel variant of the famous Cooley-Tukey algorithm particularly suited for CMPs. Benchmarks with Spiral generated DFT code on CMPs show a parallelization speed-up already for sizes that fit into L1 cache and compare favorably to other DFT libraries across all small and midsize DFTs and considered platforms.

Finally, we briefly show that the same rewriting methodology is used in Spiral for all relevant types of parallelism: SIMD vector instructions, shared and distributed memory platforms, and field-programmable gate arrays (FPGAs).

Slides: PDF

Model-guided Empirical Tuning of Loop Fusion and Tiling

Apan Qasem, Ken Kennedy (Rice)

This talk will describe a model-guided empirical tuning strategy for profitable application of loop fusion and tiling. The strategy consists of a detailed cost model that characterizes the interaction between the two transformations at different levels of the memory hierarchy. The novelty of our approach is in exposing key architectural parameters within the model for automatic tuning through empirical search. Our approach allows tuning of multiple transformations using a single parameter, which dramatically reduces the optimization search space. Preliminary experiments with a set of applications on four different platforms show that our strategy achieves significant performance improvement over fully optimized code generated by state-of-the-art commercial compilers. The time
spent in searching for the best parameters is considerably less than with other search strategies.

Slides: PDF

**An End-to-end Framework for Whole Program Tuning**

Dan Quinlan and Rich Vuduc (LLNL) and Qing Yi (UT San Antonio)

We describe on-going work that extends the ROSE source-to-source compiler infrastructure to support empirical tuning of whole applications. We rely on externally-developed tools for instrumentation and profiling (Rice's HPCToolkit), process checkpointing (U. Wisconsin's ckpt library), parameterized code generation (UTSA's POET), and a search engine (UTK). These tools complement existing components in ROSE relevant to end-to-end tuning, including an outliner, an extensive loop optimization infrastructure, and a whole program analysis capability. We present and discuss a proof-of-concept demonstration of tuning the SMG2000 benchmark within our framework using these components.

**Automatic Parallel Application Transformation**

Martin Swany (University of Delaware)

This talk will present ASPhALT, an Automatic System for Parallel AppLication Transformation. The goal of ASPhALT is transformation of MPI-based parallel applications to improve the overlapping of communication with computation. Overlapping is a key to improving performance of parallel applications, but designing applications in this way is difficult. In addition, due to differences in hardware and software, an application optimized in this way on one platform may not run with the best performance on another. Thus, an automatic transformation system, coupled with an empirical optimization framework can enable MPI programs to take advantage of advanced cluster environments with less programmer effort and can be retargeted easily.

Slides: PDF
**Automatic Tuning in the Performance Engineering Research Institute**

Kathy Yelick (U.C. Berkeley and LBNL)

The Performance Engineering Research Institute (PERI) is a recently-funded DOE SciDAC project that encompasses research in: (1) performance modeling and prediction; (2) automatic performance optimization; and (3) performance engineering of high profile applications. In this talk I will begin with an overview of the planned PERI activities, focusing on the compiler-based and library-based optimization efforts in that project.

I will then talk in more detail about the automatic tuning activities at Lawrence Berkeley National Laboratory and U.C. Berkeley, including an update on the Optimized Sparse Kernel Interface (OSKI) Library. OSKI contains a set of optimizations designed to tailor sparse matrix data structures and algorithms to a given sparsity structure and machine. OSKI has recently been integrated into the PETSc library and the resulting system demonstrated on matrices arising in large parallel applications. In addition, we are developing optimization for stencil operations that are very common in both adaptive and non-adaptive block-structured grid codes. I will show that performance models can reveal both software optimization opportunities and inherent system bottlenecks for these computations, and that careful loop reorganization can significantly improve performance. One of the optimization techniques in both sparse matrix and stencil computations is to merge computation across iterators to improve temporal locality. This reduces messages in a parallel computation and cache misses in a serial one. I will talk about some of the benefits and limitations of this approach.

PERI is a multi-institutional project led by Robert Lucas. The Berkeley/LBNL work is joint with James Demmel, Rich Vuduc, Leonid Oliker, John Shalf, Mark Hoemmen, Kaushik Datta, Hormozd Gahvari, Rajesh Nishtala, Shoaib Kamil, and Sam Williams.

**POET: Parameterized Optimizations for Empirical Tuning**

Qing Yi (University of Texas, San Antonio)

In this talk, we present POET (Parameterized Optimization for Empirical Tuning), an embedded scripting language for parameterizing optimized kernels of scientific applications. The POET language is designed to be an integral component of a general empirical tuning framework that accepts arbitrary code as input. Within such a framework, instead of producing a single version of code for each library or application, an independent optimizing compiler or a professional developer performs program analysis to identify profitable optimizations, parameterizes the code generation of relevant transformations, and then produces POET scripts as the transformation result. POET scripts can be embedded in code written in any other language (such as C, C++, or
FORTRAN), efficiently ported to different machine architectures, and dynamically configured by independent empirical search engines. We have used the POET language to parameterize and to empirically tune three loop optimizations—interchange, blocking, and unrolling—for two linear algebra kernels. We show experimentally that the time required to tune these optimizations using POET, which does not require any program analysis, is significantly shorter than that when using a full compiler-based source-code optimizer that performs sophisticated program analysis and transformations.