A Case for Machine Learning to Optimize Multicore Performance

Conference Paper · January 2009

CITATIONS
39

READS
42

4 authors, including:

Kaushik Datta
Northrop Grumman
45 PUBLICATIONS **898** CITATIONS
SEE PROFILE

Armando Fox
University of California, Berkeley
198 PUBLICATIONS **18,763** CITATIONS
SEE PROFILE

A Case for Machine Learning to Optimize Multicore Performance

Archana Ganapathi, Kaushik Datta, Armando Fox and David Patterson
Computer Science Division, University of California at Berkeley
{archanag, kdatta, fox, pattrsn}@cs.berkeley.edu

Abstract

Multicore architectures have become so complex and diverse that there is no obvious path to achieving good performance. Hundreds of code transformations, compiler flags, architectural features and optimization parameters result in a search space that can take many machine-months to explore exhaustively. Inspired by successes in the systems community, we apply state-of-the-art machine learning techniques to explore this space more intelligently. On 7-point and 27-point stencil code, our technique takes about two hours to discover a configuration whose performance is within 1% of and up to 18% better than that achieved by a human expert. This factor of 2000 speedup over manual exploration of the auto-tuning parameter space enables us to explore optimizations that were previously off-limits. We believe the opportunity for using machine learning in multicore autotuning is even more promising than the successes to date in the systems literature.

1 Introduction

Multicore architectures are becoming increasingly complex due to higher core counts, varying degrees of multithreading, and memory hierarchy complexity. This rapidly evolving landscape has made it difficult for compilers to keep pace. As a result, compilers are unable to fully utilize system resources and hand-tuning is neither scalable nor portable. Auto-tuning has emerged as a solution that can be applied across platforms for performance optimization. Auto-tuning first identifies a set of useful optimizations and acceptable parameter ranges using hardware expertise and application-domain knowledge. Then it searches the parameter space to find the best performing configuration. In the last decade, auto-tuning has been successfully used to tune scientific kernels for both serial [9, 11] and multicore processors [8, 12].

Auto-tuning is scalable, automatic, and can produce high-quality code that is several times faster than a naïve implementation [6]. Unfortunately it suffers from two major drawbacks. The first is the size of the parameter space to explore: state-of-the-art auto-tuners that consider only a single application, a single compiler, a specific set of compiler flags, and homogeneous cores may explore a search space of over 40 million configurations [8]. This search would take about 180 days to complete on a single machine. If the auto-tuner considers alternative compilers, multichip NUMA systems, or heterogeneous hardware, the search becomes prohibitively expensive. Even parallel exploration of multiple configurations (e.g. in a supercomputing environment) achieves only linear speedup in the search, so most auto-tuners prune the space by using heuristics of varying effectiveness.

The second drawback is that most auto-tuners are only trying to minimize overall running time. Given that power consumption is a proximate cause of the multicore revolution, and a vital metric for tuning embedded devices, it is important to tune for energy efficiency as well. A performance-optimal configuration is not necessarily energy-optimal and vice versa.

We propose to address both these drawbacks using statistical machine learning (SML). SML algorithms allow us to draw inferences from automatically constructed models of large quantities of data [10]. The advantage of SML-based methodologies is that they do not rely on application or micro-architecture domain knowledge. Additionally, several SML methodologies allow us to simultaneously tune for multiple metrics of success [3]. We are able to reduce the half-year long search to two hours while achieving performance at least within 1% of and up to 18% better than a that achieved by a human expert.

We are inspired by the success of the systems community in using SML techniques to detect and diagnose failures and to predict performance [3, 14, 15]. The systems community has had to overcome hurdles to applying SML [10] that may be absent in the multicore/architecture domain. In contrast to the difficulty of collecting systems data, the architecture community has the advantage that real data can be obtained quickly using commodity hardware. Furthermore, the architecture community has a long-established culture of designing for testability and measurability via programmer-visible performance counters, making it is easier to find "ground truth". Lastly, since the architecture community tends to optimize applications with exclusive access to hardware, the SML mod-

els need not adapt to varying machine usage patterns and externally-imposed load conditions. As a result of these advantages, we see a great opportunity for using SML.

2 Statistical Machine Learning for Performance Optimization

Statistical machine learning has been used by the high-performance computing (HPC) community in the past to address performance optimization for simpler problems: Brewer [2] used linear regression over three parameters to select the best data partitioning scheme for parallelization; Vuduc [16] used support vector machines to select the best of three optimization algorithms for a given problem size; and Cavazos et al [5] used a logistic regression model to predict the optimal set of compiler flags. All three examples carve out a small subset of the overall tuning space, leaving much of it unexplored. In addition, the above research was conducted prior to the multicore revolution, thus ignoring metrics of merit like energy efficiency.

Auto-tuning in the context that we are proposing explores a much larger search space than the previous work, thereby exercising the full potential of newer SML algorithms. Recast from a SML perspective, auto-tuning leverages relationships between a set of optimization parameters and a set of resultant performance metrics to explore the search space. Kernel Canonical Correlation Analysis (KCCA) [4] is a recent SML algorithm that effectively identifies these relationships. Specifically, KCCA finds multivariate correlations between optimization parameters and performance metrics on a training set of data. We can then use these statistical relationships to optimize for performance and energy efficiency.

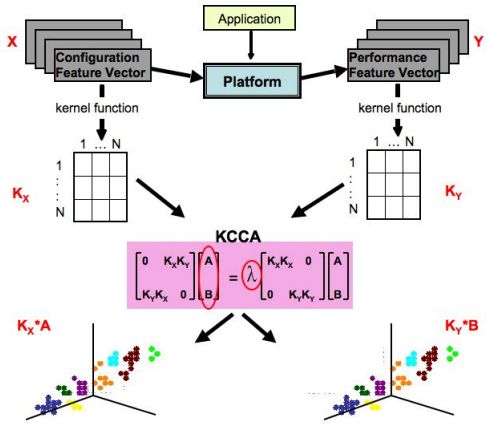

Figure 1: **The KCCA methodology discovers relationships between configurations and performance.**

For the training set data, KCCA first takes the vector representations of the configuration parameters (X) and the corresponding performance metrics (Y). Next, we construct a matrix K_x that captures the similarity of each configuration vector to every other configuration vector, and a matrix K_y that compares each performance vector to every other performance vector. Given K_x and K_y, KCCA produces a matrix A of basis vectors onto which K_x can be projected, and a matrix B of basis vectors onto which K_y can be projected, such that K_xA and K_yB are maximally correlated. That is, neighborhoods in K_xA correspond to neighborhoods in K_yB. We compute matrices A and B by solving the generalized eigenvector equation shown in Figure 1.

There are several design decisions to be considered when customizing KCCA for auto-tuning:

1. Representing optimization parameters as *configuration feature vectors* and performance metrics as *performance feature vectors*
2. Quantifying the similarity between any pair of configurations (using *kernel functions)*
3. Using the output of the KCCA algorithm to identify optimal configurations

In the following section, we present an example of optimizing a scientific code using a customized version of the KCCA algorithm.

3 Case Study: Stencil Code Optimization

Stencil (nearest-neigbor) computations are used extensively in partial differential equation (PDE) solvers involving structured grids [1]. In a typical stencil code, each grid point updates its value based on the values of its nearest neighbors. These neighbors define the stencil shape, which will be constant for every point in the grid.

3.1 Experimental Setup

System	Intel Clovertown	AMD Barcelona
Clock (GHz)	2.66	2.30
# Sockets × # Cores per Socket	2 × 4	2 × 4
Peak DP GFLOPs/second	10.7	9.2
Peak DRAM BW (GB/s)	85.3	73.6
Compiler	icc 10.1	gcc 4.1.2

Table 1: **Architectural comparison.**

Our work utilizes 3D 7-point and 27-point stencils arising from finite difference calculations. For each stencil

run, we perform a single sweep over a 256^3 grid where the read and write arrays are distinct. The 27-point stencil performs significantly more FLOPs per byte transferred than the 7-point stencil (1.25 vs. 0.33 FLOPs/byte) and thus is more likely to be compute-bound.

We conduct our experimental runs on two superscalar out-of-order machines: the Intel Clovertown and the AMD Barcelona, which Table 1 details. For each platform, we compile appropriate code variants for the 7-point and 27-point stencils. See [8] for details on each optimization and its corresponding parameters.

For each stencil, we randomly choose 1500 configurations to build our training data sets. We limit our sample size to 1500 datapoints as the KCCA algorithm is exponential with respect to the number of datapoints. We run the stencil code variant reflecting each chosen configuration and collect low-level performance counter data as detailed in Section 3.2. Then we use *kernel functions* (described in Section 3.3) to compare the similarity of every pair of stencil runs in our training set, generating two matrices that are input to our KCCA algorithm. KCCA projects these two matrices onto subspaces such that the projections are maximally correlated. However, by the nature of KCCA, it is difficult to reverse-engineer which features of the data contribute to a dimension of correlation. Thus we look up the projection of the best performing datapoint in our training set and use a genetic algorithm in that neighborhood, as described in Section 3.4. The result of this step is a new set of configurations that we run to find an optimal configuration.

Next, we consider several design decisions to customize KCCA for stencil code optimization.

3.2 Constructing Feature Vectors

For each stencil code run, we construct one configuration vector and one performance vector. The configuration feature vector is built using parameters for the optimizations listed in Table 2. See [8] for details. Similarly, the performance feature vector uses the counters listed in Table 3 as features and the measurements of these counters as values. We also measure Watts consumed using a power meter attached to our machines.

Using N=1500 runs per stencil, the N configuration vectors of K=16 features are combined into a $N \times K$ configuration matrix; the N performance vectors of L features (L=8 for Clovertown and L=6 for Barcelona) produce a $N \times L$ performance matrix. The corresponding rows in each of the two matrices describe the same stencil run.

3.3 Defining Kernel Functions

A useful aspect of the KCCA algorithm is that it produces neighborhoods of similar data with respect to configuration features as well as performance features. However, to achieve this result, KCCA uses *kernel functions*[*] to define the similarity of any two configuration vectors or any two performance vectors.

Optimization	Parameters	Total Configs
Thread Count	1	4
Domain Decomposition	4	36
Software Prefetching	2	18
Padding	1	32
Inner Loop	8	480
Total	**16**	$\mathbf{4 \times 10^7}$

Table 2: **Code optimization categories.**

Counter Description	Clovertown	Barcelona
Cycles per thread	✓	✓
L1 data cache misses	✓	✓
L2 data cache misses	✓	✓
L3 total cache misses		✓
TLB misses	✓	✓
Accesses to shared cache lines	✓	
Accesses to clean cache lines	✓	
Cache interventions	✓	
Power meter (Watts/sec)	✓	✓

Table 3: **Measured counters on the Clovertown architecture.**

Since our performance vector contains solely numeric values, we use the Gaussian kernel function [17] below:

$$k_{Gaussian}(y_i, y_j) = exp\{-\|y_i - y_j\|^2/\tau_y\}$$

where $\|y_i - y_j\|$ is the Euclidian distance and τ_y is calculated based on the variance of the norms of the data points. We derive a symmetric matrix K_y such that $K_y[i,j] = k_{Gaussian}(y_i, y_j)$. If y_i and y_j are identical, then $K_y[i,j] = 1$.

Since the configuration vectors contain both numeric and non-numeric values, we construct a kernel function using a combination of two other kernel functions. For numeric features, we use the Gaussian kernel function. For non-numeric features we define:

$$k_{binary}(x_i, x_j) = \begin{cases} 1 & \text{if } x_i = x_j, \\ 0 & \text{if } x_i \neq x_j \end{cases}$$

We define our symmetric matrix K_x such that

$$K_x[i,j] = average(k_{binary}(x_i, x_j) + k_{Gaussian}(x_i, x_j))$$

Thus, given the $N \times K$ configuration matrix and the $N \times L$ performance matrix, we form a $N \times N$ matrix K_x and a $N \times N$ matrix K_y, which are used as input to the KCCA algorithm.

[*] Our use of the term *kernel* refers to the SML *kernel function* and not HPC scientific *kernels*.

3.4 Identifying Optimal Configurations

Upon running KCCA on K_x and K_y, we obtain projections K_xA and K_yB that are maximally correlated. We leverage these projections to find an optimal configuration. We first identify the best performing point in our training set, called P_1. We look up its coordinates on the K_yB projection, and find its two nearest neighbors, called P_2 and P_3, on the projected space. We then construct new configuration vectors using all combinations of the optimizations in P_1, P_2, and P_3. We do not vary the parameters within each optimization. We run these new configurations to identify the best performing configuration.

4 Results

The first metric of success for our stencil code configurations is performance, measured in GFLOPs per second. Our performance counters inform us of cycles used for each run; we convert this number to GFLOPs/sec using the following equation:

$$\text{GFLOPs/sec} = \frac{(\text{ClockRate} \times 256^3\text{pts} \times \text{FLOPs/pt})}{\text{Cycles}}$$

Note that the clock rate is 2.66 GHz for Clovertown and 2.30 GHz for Barcelona. Furthermore, the number of FLOPs per point is 8 for the 7-point stencil and 30 for the 27-point stencil. Figures 2(a) and 2(b) show a breakdown of the GFLOP rate our SML methodology achieves for Clovertown and Barcelona respectively. They also compare our results to the no-optimization configuration and an expert optimized configuration [8].

On Clovertown, our technique provides performance within .02 GFLOPs/sec (1%) of expert optimized for the 7-point stencil and 1.5 GFLOPs/sec (18%) better for the 27-point stencil. Because the 7-point stencil is bandwidth bound on the Clovertown, none of the techniques show significant performance gains. The significant performance gain for the 27-point can be attributed to two factors: (i) our domain decomposition parameters more efficiently exploit the 27-point stencil's data locality; (ii) we likely use registers and functional units more efficiently as a result of our inner loop parameter values.

On Barcelona, our performance is 0.6 GFLOPs/sec (16%) better than that achieved by expert optimized for the 7-point stencil and within 0.35 GFLOPs/sec (2%) for the 27-point stencil. For the 7-point stencil, our inner loop optimization parameters unroll along a different dimension than expert optimized. We also prefetch further ahead than the expert optimized configuration. For the 27-point stencil on Barcelona, the dominant factor causing the performance gap is the smaller padding size in the unit stride dimension used by the expert optimized configuration. Furthermore, the expert's configuration use more efficient domain decomposition.

The second metric of success is energy efficiency. Based on power meter wattage readings, we calculate energy efficiency with the following formula:

$$\text{Energy Efficiency} = \frac{\text{MFLOPs/sec}}{\text{Watts}}$$

As seen in Figure 2(c), on Clovertown we achieve within 1% of the expert optimized energy efficiency for the 7-point stencil and 13% better than expert optimized for the 27-point stencil. For both our stencil codes on Clovertown and the 27-point stencil code on Barcelona, the best performing configuration is also the most energy efficient. For the 7-point stencil on Barcelona, the energy efficiency of our fastest run differs from the most energy efficient run by a mere 0.3%. As a result, we have omitted the Barcelona energy efficiency graph.

Figure 2(d) compares performance against energy efficiency on the Clovertown. The slope of the graph represents Watts consumed, and since marker shape/color denote thread count, we see that power consumption is dictated by the number of threads used. We observe configurations with identical performance but differing energy efficiency and vice versa, as highlighted by the red oval. In environments with real-time constraints (e.g., portable devices), there is no benefit to completing well before the real-time deadline; but there is significant benefit to conserving battery power. In such environments, performance can be sacrificed for energy efficiency, and thus we expect a wider gap between the two metrics.

5 Discussion

5.1 How long does our methodology take?

The last row in Table 2 shows the total number of configurations in the exhaustive search space. At 5 trials per configuration and about 0.08 seconds to run each trial, the total time amounts to 180 days. Our case study only requires running 1500 randomly chosen configurations. Given our previous assumptions, our runs would complete in 400 seconds; however, we must add the time it takes to build the model (approximately 2 hours for the 1500 data points) and the time to compute the heuristic and run the suggested configurations (under one minute) - adding up to just over two hours! Obviously, domain knowledge would help eliminate areas of the search space, which is reflected by our expert-optimized results. However, our methodology is easier to scale to other architectures as well as other optimization problems, such as FFTs and sparse matrix multiplication [12].

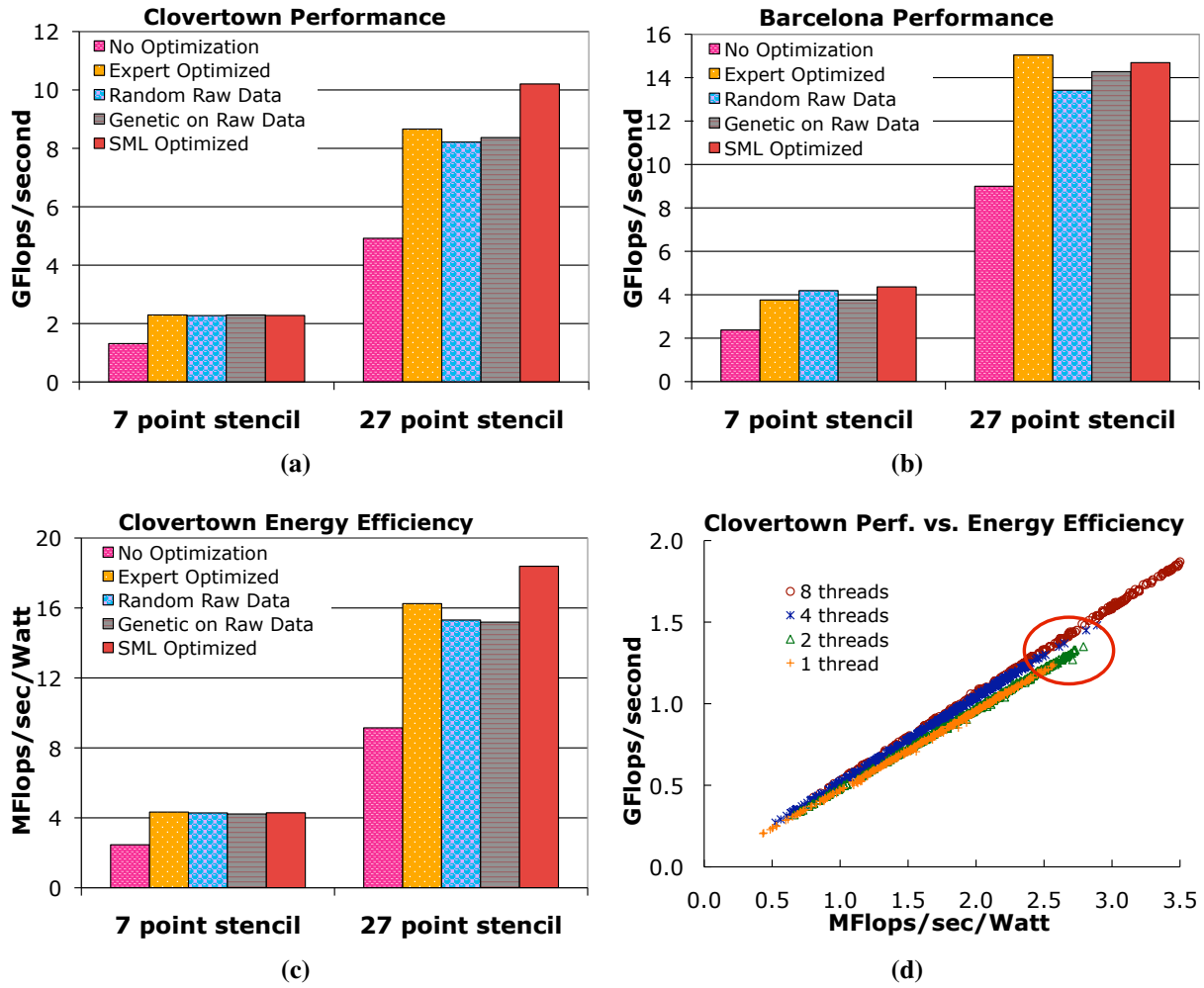

Figure 2: **(a) Stencil performance on Intel Clovertown (b) Stencil performance on AMD Barcelona (c) Energy efficiency on Clovertown (d) Performance vs. energy efficiency for the 7-point stencil training data on Clovertown. Note that the slope indicates Watts consumed. The red oval highlights configurations with similar performance but different energy efficiencies.**

5.2 Can we measure success without comparing to an expert?

Human experts incorporate architectural and application-specific domain knowledge to identify a good configuration. However, the expert's configuration may not reflect the system's true performance upper bound. The Roofline model [13] uses architectural specifications and microbenchmarks to calculate peak performance. We can use this model to gauge how close we are to fully exhausting system resources.

For instance, the Roofline model considers memory bandwidth as a potential bottleneck. Considering only compulsory misses in our stencil codes, we achieve 95.4% of Stream bandwidth for the 7-point stencil and 13.8% more than the Stream bandwidth for the 27-point stencil on the Clovertown platform. We exceed Stream bandwidth for the 27-point stencil because the Stream benchmark is unoptimized. On Barcelona, we achieve 89.2% of Stream bandwidth for the 7-point stencil and 80.1% for the 27-point stencil. Since Barcelona has a lower FLOP to byte ratio than Clovertown, the stencils are more likely to be compute bound on this platform. As a result, we are unable to achieve a higher fraction of stream bandwidth.

5.3 Do simpler algorithms work?

One of the common criticisms of our methodology is that a simpler algorithm would have worked just as well. To address this concern, Figures 2(a)-(c) include results for two simpler alternatives. The *Random Raw Data* column shows the best performing point in our training data set. The *Genetic on Raw Data* column shows the best case performance achieved by using a genetic algorithm (combinations) on the top three best-performing points in our

training data set. We do not vary the parameters within each optimization. While these two techniques are building blocks of our methodology, individually they do not perform as well as our SML optimized results. In the future, we plan to explore conjugate gradient descent algorithms in the best-performing KCCA neighborhood as an alternative to the genetic algorithm we currently use.

6 Conclusions and Future Work

We have shown that SML can quickly identify configurations that simultaneously optimize running time and energy efficiency. SML based auto-tuning methodologies are agnostic to the underlying architecture as well as the code being optimized, resulting in a scalable alternative to human expert-optimization.

As an example, we optimized two stencil codes on two multicore architectures, either matching or outperforming a human expert by up to 18%. The optimization process took about two hours to explore a space that would take half a year to explore exhaustively on a single computer by conventional techniques. This result gives us reason to believe that SML effectively handles the combinatorial explosion posed by the optimization parameters, potentially allowing us to explore some previously-intractable research directions:

- **Fine-grained power profiling** We can improve energy efficiency by considering each component's individual power consumption rather than only considering whole-machine power consumption. Architectures with dynamic voltage and frequency scaling of either the whole chip or individual cores would also increase the number of tunable parameters.
- **Scientific motif composition** While we optimized a single scientific motif, complex problems often consist of a composition of these motifs; Asanovic et al [7] have identified thirteen motifs common in parallel computing, of which seven are fundamental to HPC. Optimal configurations for a particular motif may not correspond to optimal configurations for composing that motif with others. We can explore the composition space using the techniques in this paper by merely changing the training data set.
- **Tuning applications for multi-chip servers** We can tune applications for multi-chip servers, optimizing both computation on individual (possibly heterogeneous) nodes as well as communication efficiency across the network.

Our results to date, and the promise of mapping difficult research problems such as those above onto approaches similar to our own, give us confidence that SML will open up exciting new opportunities to advance the state-of-the-art in multicore performance optimization.

References

[1] Applied Numerical Algorithms Group, Lawrence Berkeley National Laboratory, Berkeley, CA. Chombo:. `http://seesar.lbl.gov/ANAG/software.html`.

[2] E. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Not.*, 30(8):80–91, 1995.

[3] A. Ganapathi et al. Predicting multiple performance metrics for queries: Better decisions enabled by machine learning. In *Proc International Conference on Data Engineering (ICDE)*, March 2009.

[4] F.R. Bach et al. Kernel independent component analysis. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Hong Kong, China, 2003.

[5] J. Cavazos et al. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2007.

[6] J. Demmel et al. Self adapting linear algebra algorithms and software. In *Proc IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, February 2005.

[7] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS, UC Berkeley, 2006.

[8] K. Datta et al. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proc SC2008: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008.

[9] M. Frigo et al. The design and implementation of FFTW3. *IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[10] Moises Goldszmidt et al. Three research challenges at the intersection of machine learning, statistical induction, and systems. In *HotOS 2005*, 2005.

[11] R. Vuduc et al. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc SciDAC 2005, J. of Physics: Conference Series*, June 2005.

[12] S. Williams et al. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc SC2007: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2007.

[13] S. Williams et al. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM (CACM) (to appear)*, April 2009.

[14] S. Zhang et al. Capturing, indexing, clustering, and retrieving system history. In *Proc 20th ACM Symposium on Operating Systems*, 2005.

[15] E. Kıcıman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks (special issue on Adaptive Systems)*, Spring 2005.

[16] R.Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, UC Berkeley, 2003.

[17] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.