Statistical Models for Automatic Performance Tuning

Richard Vuduc[1], James W. Demmel[2], and Jeff Bilmes[3]

[1] Computer Science Division
University of California at Berkeley, Berkeley, CA 94720 USA
richie@cs.berkeley.edu
[2] Computer Science Division and Dept. of Mathematics
University of California at Berkeley, Berkeley, CA 94720 USA
demmel@cs.berkeley.edu
[3] Dept. of Electrical Engineering
University of Washington, Seattle, WA USA
bilmes@ee.washington.edu

Abstract. Achieving peak performance from library subroutines usually requires extensive, machine-dependent tuning by hand. Automatic tuning systems have emerged in response, and they typically operate, at compile-time, by (1) generating a large number of possible implementations of a subroutine, and (2) selecting a fast implementation by an exhaustive, empirical search. This paper applies statistical techniques to exploit the large amount of performance data collected during the search. First, we develop a heuristic for stopping an exhaustive compile-time search early if a near-optimal implementation is found. Second, we show how to construct run-time decision rules, based on run-time inputs, for selecting from among a subset of the best implementations. We apply our methods to actual performance data collected by the PHiPAC tuning system for matrix multiply on a variety of hardware platforms.

1 Introduction

Standard library interfaces have enabled the development of portable applications that can also achieve *portable performance*, provided that optimized libraries are available and affordable on all platforms of interest to users. Example libraries in scientific applications include the Basic Linear Algebra Subroutines (BLAS) [11,5], the Vector and Signal Image Processing Library API [12], and the Message Passing Interface (MPI) for distributed parallel communications.

However, both construction and machine-specific hand-tuning of these libraries can be tedious and time-consuming tasks. Thus, several recent research efforts are automating the process using the following two-step method. First, rather than code particular routines by hand for each computing platform of interest, these systems contain parameterized code generators that encapsulate possible tuning strategies. Second, the systems tune for a particular platform by *searching*, i.e., varying the generators' parameters, benchmarking the resulting routines, and selecting the fastest implementation.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 117–126, 2001.

In this paper, we focus on the possible uses of performance data collected during the search task.[1] Specifically, we first justify the need for exhaustive searches in Section 2, using actual data collected from an automatic tuning system. However, users of such systems cannot always afford to perform these searches. Therefore, we discuss a statistical model of the feedback data that allows users to stop the search early based on meaningful information about the search's progress in Section 3. Of course, a single implementation is not necessarily the fastest possible for all possible inputs. Thus, we discuss additional performance modeling techniques in Section 4 that allow us to select at run-time an implementation believed to perform best on a particular input. We apply these techniques to data collected from the PHiPAC system (see Section 2) which generates highly tuned matrix multiply implementations [1,2].

There are presently a number of other similar and important tuning systems. These include FFTW for discrete Fourier transforms [6], ATLAS [17] for the BLAS, Sparsity [8] for sparse matrix-vector multiply, and SPIRAL [7,13] for signal and image processing. Vadhiyar, et al. [14], explore automatically tuning MPI collective operations. These systems employ a variety of sophisticated code generators that use both the mathematical structure of the problems they solve and the characteristics of the underlying machine to generate high performance code. All match hand-tuned vendor libraries, when available, on a wide variety of platforms. Nevertheless, these systems also face the common problem of how to reduce the lengthy search process. Each uses properties specific to their code generators to prune the search spaces. Here, we present complementary techniques for pruning the search spaces independently of the code generator.

The search task deserves attention not only because of its central role in specialized tuning systems, but also because of its potential utility in compilers. Researchers in the OCEANS project [10] are integrating such an empirical search procedure into a general purpose compiler. Search-directed compilation should be valuable when performance models fail to charaterize source code adequately.

2 The Case for Searching

In this section, we present data to motivate the need for search methods in automated tuning systems, using PHiPAC as a case study. PHiPAC searches a combinatorially large space defined by possible optimizations in building its implementation. Among the most important optimizations are (1) register, L1, and L2 cache tile sizes where non-square shapes are allowed, (2) loop unrolling, and (3) a choice of six software pipelining strategies. To limit search time, machine parameters (such as the number of registers available and cache sizes) are used to restrict tile sizes. In spite of this and other pruning heuristics, searches generally can take hours to *weeks* depending on the user-selectable thoroughness of the search. Nevertheless, Figure 1 shows two examples in which the performance of PHiPAC-generated routines compares well with (a) hand-tuned vendor libraries and (b) "naive" C code (3-nested loops) compiled with full optimizations.

[1] An extended version of this paper has appeared elsewhere [16].

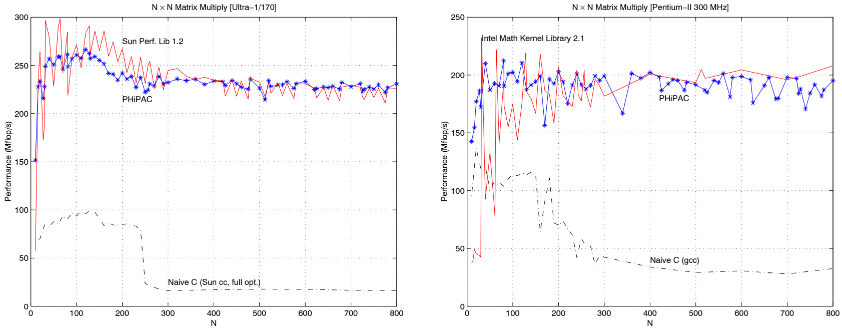

Fig. 1. Performance (Mflop/s) on a square matrix multiply benchmark for the Sun Ultra 1/170 workstation (*left*) and a 300 MHz Pentium-II platform (*right*). The theoretical peaks are 333 Mflop/s and 300 Mflop/s, respectively.

Exhaustive searches are often necessary to find the very best implementations, although a partial search can find near-optimal implementations. In an experiment we fixed a particular software pipelining strategy and explored the space of possible register tile sizes on six different platforms. This space is three-dimensional and we index it by integer triplets (m_0, k_0, n_0).[2] Using heuristics, this space was pruned to contain between 500 and 2500 reasonable implementations per platform. Figure 2 (*left*) shows what fraction of implementations (y-axis) achieved what fraction of machine peak (x-axis). On the IBM RS/6000, 5% of the implementations achieved at least 90% of the machine peak. By contrast, only 1.7% on a uniprocessor Cray T3E node, 4% on a Pentium-II, and 6.5% on a Sun Ultra1/170 achieved more than 60% of machine peak. And on a majority of the platforms, fewer than 1% of implemenations were within 5% of the best; 80% on the Cray T3E ran at less than 15% of machine peak. Two important ideas emerge: (1) different machines can display widely different characteristics, making generalization of search properties across them difficult, and (2) finding the very best implementations is akin to finding a "needle in a haystack."

The latter difficulty is illustrated in Figure 2 (*right*), which shows a 2-D slice $(k_0 = 1)$ of the 3-D tile space on the Ultra. The plot is color coded from black=50 Mflop/s to white=270 Mflop/s. The lone white square at $(m_0 = 2, n_0 = 8)$ was the fastest. The black region to the upper-right was pruned (i.e., not searched) based on the number of registers. We see that performance is not a smooth function of algorithmic details, making accurate sampling and interpolation of the space difficult. Like Figure 2 (*left*), this motivates an exhaustive search.

3 Early Stopping Criteria

Unfortunately, exhaustive searches can be demanding, requiring dedicated machine time for long periods. Thus, tuning systems prune the search spaces using

[2] The specifics of why the space is three dimensional are, for the moment, unimportant.

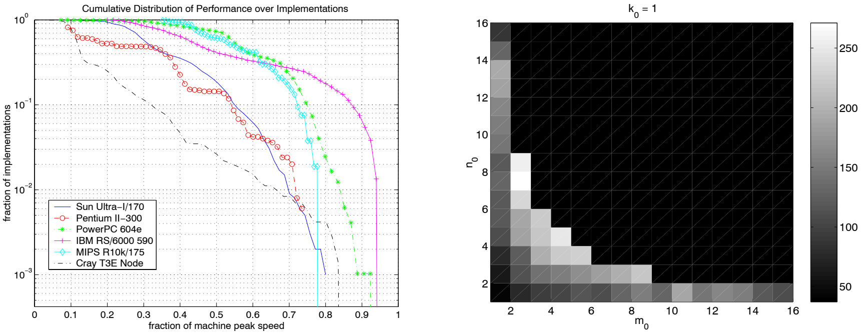

Fig. 2. (*Left*) The fraction of implementations (y-axis) attaining at least a given level of peak machine speed (x-axis) on six platforms. (*Right*) A 2-D slice of the 3-D register tile space on the Sun Ultra1/170 platform. The best implementation ($m_0 = 2, n_0 = 8$) achieved 271 Mflop/s.

application-specific heuristics. We consider a complementary method for stopping a search early based only on performance data gathered during the search.

More formally, suppose there are N possible implementations. When we generate implementation i, we measure its performance x_i. Assume that each x_i is normalized to lie between 0 (slowest) and 1 (fastest). Define the space of implementations as $S = \{x_1, \ldots, x_N\}$. Let X be a random variable corresponding to the value of an element drawn uniformly at random from S, and let $n(x)$ be the number of elements of S less than or equal to x. Then X has a cumulative distribution function (cdf) $F(x) = Pr[X \leq x] = n(x)/N$. At time t, where t is between 1 and N inclusive, suppose we generate an implementation at random *without* replacement. Let X_t be a random variable corresponding to the observed performance. Letting $M_t = \max_{1 \leq i \leq t} X_i$ be the maximum observed performance at t, we can ask about the chance that M_t is less than some threshold:

$$Pr[M_t \leq 1 - \epsilon] < \alpha, \tag{1}$$

where ϵ is the proximity to the best performance, and α is an upper-bound on the probability that the observed maximum at time t is below $1 - \epsilon$. Note that

$$Pr[M_t \leq x] = Pr[X_1 \leq x, X_2 \leq x, \ldots, X_t \leq x] = p_1(x)p_2(x)\cdots p_t(x) \tag{2}$$

where, assuming no replacement,

$$\begin{aligned} p_r(x) &= Pr[X_r \leq x | X_1 \leq x, \ldots, X_{r-1} \leq x] \\ &= \begin{cases} 0 & n(x) < r \\ \frac{n(x)-r+1}{N-r+1} & n(x) \geq r \end{cases} \end{aligned} \tag{3}$$

Since $n(x) = N \cdot F(x)$, we cannot know its true value since we do not know the true distribution $F(x)$. However we can use the t observed samples to approximate $F(x)$ using, say, the empirical cdf (ecdf) $\hat{F}_t(x)$ based on the t samples:

$$\hat{F}_t(x) = \hat{n}_t(x)/t \tag{4}$$

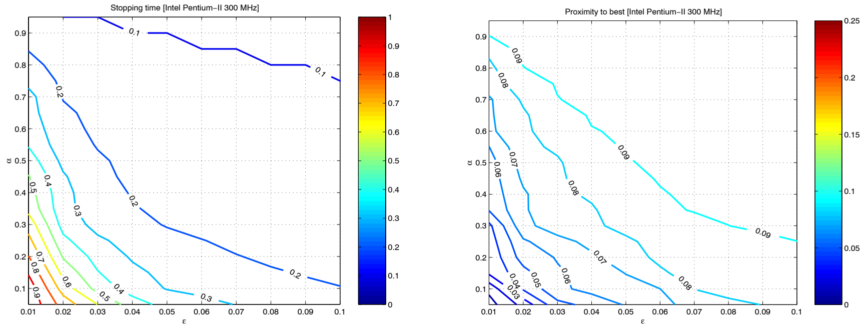

Fig. 3. Average stopping time (left), as a fraction of the total search space, and proximity to the best performance (right), as the difference between normalized performance scores, on the 300 MHz Pentium-II class workstation as functions of the tolerance parameters ϵ (x-axis) and α (y-axis). Note that the values shown are mean *plus* standard deviation, to give an approximate upper-bound on the average case.

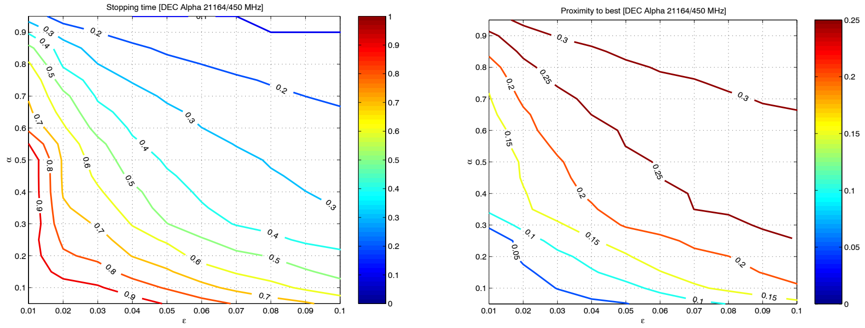

Fig. 4. Same as Figure 3 for a uniprocessor Cray T3E node.

where $\hat{n}_t(x)$ is the number of observed samples less than or equal to x. We *rescale* the samples so that the maximum is one, since we do not know the true maximum.[3] Other forms for equation (4) are opportunities for experimentation.

In summary, a user or library designer specifies the search tolerance parameters ϵ and α. Then at each time t, the automated search system builds the ecdf in equation (4) to estimate (2). The search ends when equation (1) is satisfied.

We apply the above model to the register tile space data to the platforms shown in Figure 2 (*left*). The results appear in Figures 3 and 4 for the Pentium and Cray T3E platforms, respectively. The left plots show the average stopping time *plus* the standard deviation as a function of ϵ and α; this gives a pessimistic bound on the average value. The right plots show the average proximity of the

[3] This was a reasonable approximation on actual data. We are developing theoretical bounds on the quality of this approximation, which we expect will be close to the known bounds on ecdf approximation due to Kolmogorov and Smirnov [3].

implementation found to the best one (again, plus the standard deviation), as a fraction. On the Pentium (Figure 3), setting $\epsilon = .05$ and $\alpha = .1$ we see that the search ends after sampling less than a third of the full space (left plot), having found an implementation within about 6.5% of the best (right plot). On the Cray T3E (Figure 4) where the best is difficult to find, the same tolerance values produce an implementation within about 8% of the best while still requiring exploration of 80% of the search space. Thus, the model adapts to the characteristics of the implementations and the underlying machine.

In prior work [1], we experimented with search methods including random, ordered, best-first, simulated annealing. The OCEANS project [10] has also reported on a quantitative comparison of these methods and others. In both, random search was comparable to and easier to implement than the others. Our technique adds user-interpretable bounds to the simple random method. Note that if the user wishes to specify a maximum search time (e.g., "stop searching after 3 hours"), the bounds could be computed and reported to the user.

4 Run-Time Selection Rules

The previous sections assume that a single, optimal implementation can be found. For some applications, however, several implementations may be "optimal" depending on the input parameters. Thus, we may wish to build decision rules to select an appropriate implementation based on the run-time inputs.

Formally, we want to solve the following problem. Suppose we are given (1) a set of m "good" implementations of an algorithm, $A = \{a_1, \ldots, a_m\}$ which all give the same output when presented with the same input; (2) a set of samples $S_0 = \{s_1, s_2, \ldots, s_n\}$ from the space S of all possible inputs (i.e., $S_0 \subseteq S$), where each s_i is a d-dimensional real vector; (3) the execution time $T(a, s)$ of algorithm a on input s, where $a \in A$ and $s \in S$. Our goal is to find a decision function $f(s)$ that maps an input s to the best implementation in A, i.e., $f : S \to A$. The idea is to construct $f(s)$ using the performance of the good implementations on a sample of the inputs S_0. We will refer to S_0 as the *training set.* In geometric terms, we would like to partition the input space by implementation. This would occur at compile (or "build") time. At run-time, the user calls a single routine which, when given an input s, evaluates $f(s)$ to select and execute an implementation.

There are a number of important issues. Among them is the cost and complexity of building f. Another is the cost of evaluating $f(s)$; this should be a fraction of the cost of executing the best implementation. A third issue is how to compare the prediction accuracy of different decision functions. One possible metric is the average misclassification rate, or fraction of test samples mispredicted (call it Δ_{miss}). We always choose the *test set* S' to exclude the training data S_0, that is, $S' \subseteq (S - S_0)$. However, if the performance difference between two implementations is small, a misprediction may still be acceptable. Thus, we also use the average slow-down of the selected variant relative to the best, Δ_{err}.

For example, consider the matrix multiply operation $C = C + AB$, where A, B, and C are dense matrices of size $M \times K$, $K \times N$, and $M \times N$, respectively. In

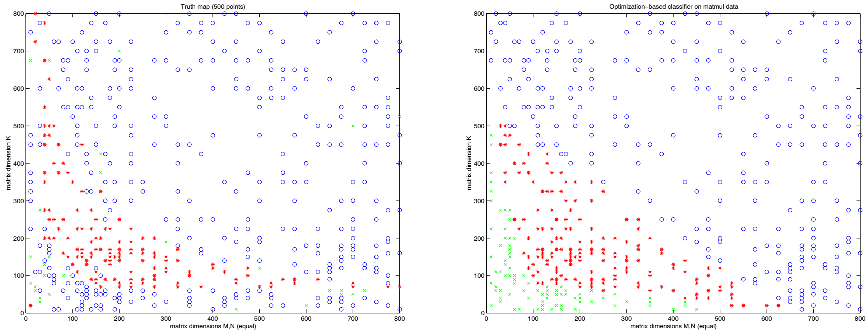

Fig. 5. (*Left*) A "truth map" showing the regions in which particular implementations are fastest. A 500-point sample of a 2-D slice of the input space is shown. Red *'s correspond to an implementation with only register tiling, green x's have L1 cache tiling, and blue o's have L1 and L2 tiling. (*Right*) Prediction results for the cost-based method.

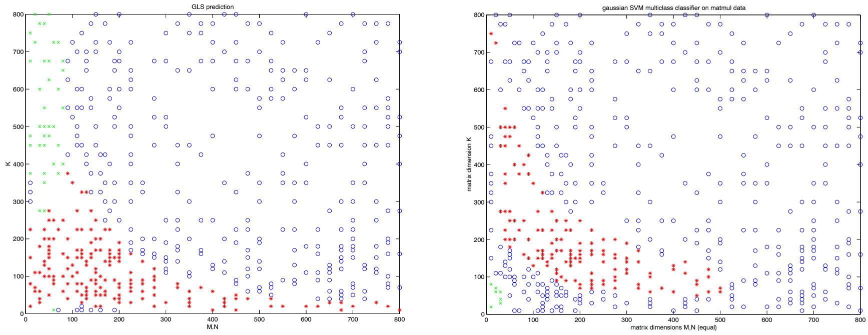

Fig. 6. Prediction results for the regression (*left*) and support-vector (*right*) methods.

PHiPAC, it is possible to generate different implementations tuned on different matrix workloads. For instance, we could have three implementations, tuned for matrix sizes that fit approximately within L1 cache, those that fit within L2, and all larger sizes. The inputs to each are M, K, and N, making the input space S three-dimensional. We will refer to this example in the following sections.

4.1 A Cost Minimization Method

Associate with each implementation a a weight function $w_{\theta_a}(s)$, parameterized by θ_a, which returns a value between 0 and 1 for some input value s. Our decision function selects the algorithm with the highest weight on input s, $f(s) = \mathrm{argmax}_{a \in A} \{w_{\theta_a}(s)\}$. Compute the weights so as to minimize the average execution time over the training set, i.e., minimize

$$C(\theta_{a_1}, \ldots, \theta_{a_m}) = \sum_{a \in A} \sum_{s \in S_0} w_{\theta_a}(s) \cdot T(a, s). \quad (5)$$

Of the many possible choices for w_{θ_a}, we choose the *softmax function* [9], $w_{\theta_a}(s) = exp\left(\theta_a^T s + \theta_{a,0}\right)/Z$ where θ_a has the same dimensions as s, $\theta_{a,0}$ is an additional parameter to estimate, and Z is a normalizing constant. It turns out that the derivatives of the weights are easy to compute, so we can estimate θ_a and $\theta_{a,0}$ by minimizing equation (5) numerically using Newton's method. A nice property of the weight function is that f becomes cheap to evaluate at run-time.

4.2 Regression Models

Another natural idea is to postulate a parametric model for the running time of each implementation. Then at run-time, we can choose the fastest implementation based on the execution time predicted by the models. This approach was originally proposed by Brewer [4]. For matrix multiply on matrices of size $N \times N$, we might guess that the running time of implementation a will have the form

$$T_a(N) = \beta_3 N^3 + \beta_2 N^2 + \beta_1 N + \beta_0. \tag{6}$$

Given sample running times on some inputs S_0, we can use standard regression techniques to determine the β_k coefficients. The decision function is just $f(s) = \text{argmin}_{a \in A} T_a(s)$. An advantage of this approach is that the models, and thus the accuracy of prediction as well as the cost of making a prediction, can be as simple or as complicated as desired. Also, no assumptions are being made about the geometry of the input space, as with our cost-minimization technique. However, a difficult disadvantage is that it may not be easy to postulate an accurate run-time model.

4.3 The Support Vector Method

Another approach is to view the problem as a statistical classification task. One sophisticated and successful classification algorithm is known as the support vector (SV) method [15]. In this method, each sample $s_i \in S_0$ is given a label $l_i \in A$ to indicate which implementation was fastest for that input. The SV method then computes a partitioning that attempts to maximize the minimum distance between classes.[4] The result is a decision function $f(s)$. The SV method is reasonably well-grounded theoretically and potentially much more accurate than the previous two methods, and we include it in our discussion as a kind of practical upper-bound on prediction accuracy. However, the time to compute $f(s)$ is a factor of $|S_0|$ greater than that of the other methods and is thus possibly much more expensive to calculate at run-time.

4.4 Results with PHiPAC Data

We offer a brief comparison of the three methods on the matrix multiply example described previously. The predictions of the three methods on a sample test set

[4] Formally, this is the *optimal margin* criterion [15].

Table 1. The three predictors on matrix multiply. "Best 5%" is the fraction of predicted implementations whose execution times were within 5% of the best possible. "Worst 20%" and "50%" are the fraction less than 20% and 50% of optimal, respectively.

			Best	Worst	
Method	Δ_{miss}	Δ_{err}	*5%*	*20%*	*50%*
Regression	34.5%	2.6%	90.7%	1.2%	0.4%
Cost-Min	31.6%	2.2%	94.5%	2.8%	1.2%
SVM	12.0%	1.5%	99.0%	0.4%	0%

are shown in Figures 5 (right) and 6. Qualitatively, we see that the boundaries of the cost-based method are a poor fit to the data. The regression method captures the boundaries roughly but does not correctly model one of the implementations (upper-left of figure). The SV method appears to produce the best predictions.

Table 1 compares the accuracy of the three methods by the two metrics Δ_{miss} and Δ_{err}; in addition we report the fraction of test points predicted *within* 5% of the best possible, and the fraction predicted that were 20% and 50% *below* optimal. These values are averaged over ten training and test sets. The values for Δ_{miss} confirm the qualitative results shown in the figures. However, the methods are largely comparable by the Δ_{err} metric, showing that a high misclassification rate did not necessarily lead to poor performance overall. Note that the worst 20% and 50% numbers show that the regression method made slightly better mispredictions on average than the cost-minimization method. In addition, both the regression and cost-minimization methods lead to reasonably fast predictors. Prediction times were roughly equivalent to the execution time of a 3x3 matrix multiply. By contrast, the prediction cost of the SVM is about a 64x64 matrix multiply, which may prohibit its use when small sizes occur often.

However, this analysis is not intended to be definitive. For instance, we cannot fairly report on specific training costs due to differences in the implementations in our experimental setting. Also, matrix multiply is only one possible application. Instead, our aim is simply to present the general framework and illustrate the issues on actual data. Moreover, there are many possible models; our examples offer a flavor for the role that statistical modeling of performance data can play.

5 Conclusions and Directions

While all of the existing automatic tuning systems implicitly follow the two-step "generate-and-search" methodology, one aim of this study is to draw attention to the process of searching itself as an interesting and challenging problem.

One challenge is pruning the enormous implementation spaces. Existing tuning systems have shown the effectiveness of pruning these spaces using problem-specific heuristics; our black-box pruning method for stopping the search process early is a complementary technique. It has the nice properties of (1) incorporating performance feedback data, and (2) providing users with a meaningful way (namely, via probabilistic thresholds) to control the search procedure.

The other challenge is to find efficient ways to select implementations at run-time when several known implementations are available. Our aim has been to discuss a possible framework, using sampling and statistical classification, for attacking this problem in the context of automatic tuning systems. This connects high performance software engineering with statistical modeling ideas. Other modeling techniques and applications remain to be explored.

Acknowledgements. We wish to thank Andrew Ng for his feedback on our statistical methodology.

References

1. J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proc. of the Int'l Conf. on Supercomputing, Vienna, Austria*, July 1997.
2. J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C. Chin. The PHiPAC v1.0 matrix-multiply distribution. Technical Report UCB/CSD-98-1020, University of California, Berkeley, October 1998.
3. Z. W. Birnbaum. Numerical tabulation of the distribution of Kolmogorov's statistic for finite sample size. *J. Am. Stat. Assoc.*, 47:425–441, September 1952.
4. E. Brewer. High-level optimization via automated statistical modeling. In *Sym. Par. Alg. Arch., Santa Barbara, California*, July 1995.
5. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
6. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the Int'l Conf. on Acoustics, Speech, and Signal Processing*, May 1998.
7. G. Haentjens. An investigation of recursive FFT implementations. Master's thesis, Carnegie Mellon University, 2000.
8. E.-J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proc. of the 9th SIAM Conf. on Parallel Processing for Sci. Comp.*, March 1999.
9. M. I. Jordan. Why the logistic function? Technical Report 9503, MIT, 1995.
10. T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the 8th International Workshop on Compilers for Parallel Computers*, pages 35–44, 2000.
11. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
12. D. A. Schwartz, R. R. Judd, W. J. Harrod, and D. P. Manley. VSIPL 1.0 API, March 2000. `www.vsipl.org`.
13. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proc. of the 17th Int'l Conf. on Mach. Learn.*, 2000.
14. S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective operations. In *Proceedings of Supercomputing 2000*, November 2000.
15. V. N. Vapnik. *Statistical Learning Theory.* John Wiley and Sons, Inc., 1998.
16. R. Vuduc, J. Demmel, and J. Bilmes. Statistical modeling of feedback data in an automatic tuning system. In *MICRO-33: Third ACM Workshop on Feedback-Directed Dynamic Optimization*, December 2000.
17. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, 1998.