Statistical Modeling of Feedback Data in an Automatic Tuning System

Richard Vuduc* Jeff Bilmes† James Demmel‡

Abstract

Achieving peak performance from library subroutines usually requires extensive, machine-dependent tuning by hand. Automatic tuning systems have been developed in response which 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. In this paper, we show how statistical modeling of the performance feedback data collected during the search phase can be used in two novel and important ways. 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 and compiler 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. Examples of such standards in science and engineering applications include the Basic Linear Algebra Subroutines (BLAS) [12, 6, 5], the Vector and Signal Image Processing Library API [14], 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, these systems contain parameterized code generators that encapsulate possible tuning strategies. Second, the systems tune for a particular hardware platform by *searching*, i.e., varying the generators' parameters, benchmarking the resulting routines, and selecting the fastest implementation.[1]

In this paper, we focus on the possible uses of feedback data (i.e., benchmark results) during the search task. 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 feedback 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 for matrix multiply [1, 2]. PHiPAC was the first system to propose the "generate and search" methodology. Its code generator produces matrix multiply implementations with various loop unrolling depths, varying register and L1- and L2-cache tile sizes [13], different software pipelining strategies, among other options. The output of the generator is C code, both to make the system portable and to allow the compiler to perform the final register allocation and instruction scheduling. The search phase benchmarks combinations of these generator options to select the best implementation.

There have been a number of other similar and important tuning systems. These include FFTW for discrete Fourier transforms [7], ATLAS [18] for the BLAS, Sparsity [9] for sparse matrix-vector multiply, and SPIRAL [8, 15] for signal and image processing. Vadhiyar, et al. [16], explore automatically tuning MPI collective operations. These sys-

*CS Division, U.C. Berkeley, Berkeley, CA 94720 USA, richie@cs.berkeley.edu

†Department of Electrical Engineering, Univ. of Washington, Seattle, WA 98195 USA, bilmes@ee.washington.edu

‡Department of Mathematics and Department of EECS, CS Division, U.C. Berkeley, Berkeley, CA 94720 USA, demmel@cs.berkeley.edu

[1]The performance of routines targeted by this approach are assumed to be of paramount importance, and the search process need only be performed once for a particular platform.

tems 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. While this will always be a necessary and effective approach in practice, in this paper we consider complementary techniques for pruning the search spaces independently of the code generator.

The search task deserves particular attention not only because of its central role in specialized tuning systems, but also because of its potential utility in compilers. Researchers in the Espirit OCEANS compiler project [11] are integrating such an empirical search procedure into a general purpose compiler. Their techniques for searching the space differ from ours in that we attempt to model statistically the search space itself, in order to provide users with a more meaningful early stopping criterion.

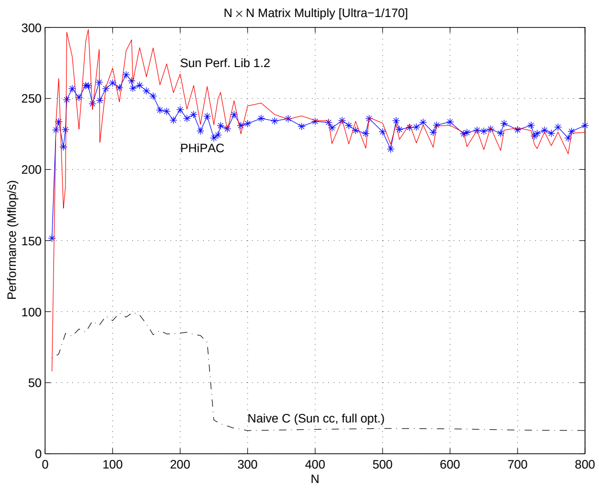
As far as we know, the methods for run-time selection that we discuss in Section 4, which are based on using feedback data in a statistical classification setting, have not been discussed previously.

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. We begin with an overview of PHiPAC, and then discuss data collected with the system to show the difficulty and necessity of searching.

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 values 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 limit tile sizes. In addition, the search first finds the best register tile size, then uses that to find an L1 tile size, and similarly with L2, etc. Still, searches generally can take hours to *weeks* depending on the user-selectable thoroughness of the search.

Figure 1 shows the performance on two platforms of PHiPAC-generated matrix multiply routines compared to hand-tuned vendor libraries and a “naive” C implementation (3-nested loops) on a square matrix multiply benchmark. PHiPAC routines compare fa-



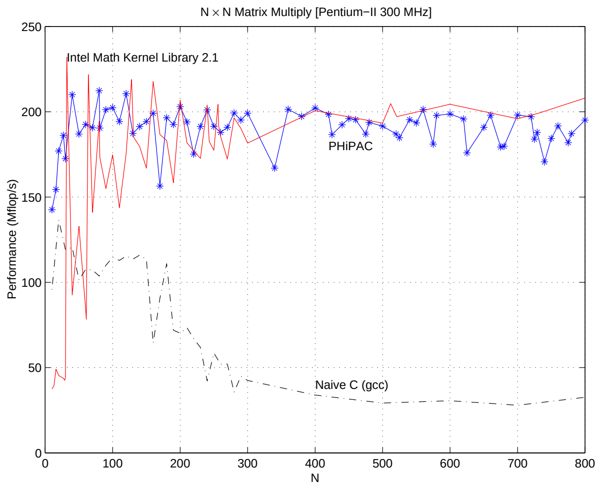


Figure 1: Performance (Mflop/s) on a square matrix multiply benchmark for the Sun Ultra 1/170 workstation (*top*) and a 300 MHz Pentium-II platform (*bottom*). The theoretical peaks are 333 Mflop/s and 300 Mflop/s, respectively.

vorably with the vendor routines, and are much faster than the naive versions which were compiled with full compiler optimizations enabled. Thus, although the PHiPAC generator optimizations are available in modern compilers, there is still a significant benefit to coding them explicitly.[2]

In addition, exhaustive searches are often necessary to find the very best implementations, although a partial search can find near-optimal implementations. Consider the case in which we fix a particular software pipelining strategy and explore the space of possible register tile sizes on six different platforms. This space is three-dimensional and we index it by

[2]Moreover, searches conducted on a number of platforms selected widely varying and often unanticipated combinations of generator options.

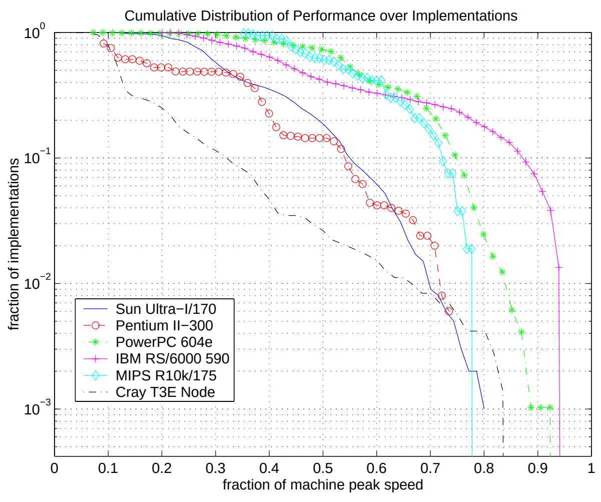


Figure 2: The fraction of register-tiled implementations (y-axis) attaining at least a given level of peak machine speed (x-axis) on six different machine platforms. Note the log scale on the y-axis.

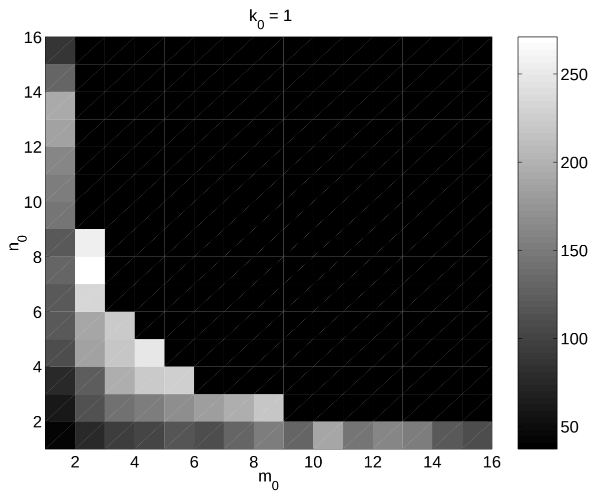


Figure 3: A 2-D slice of the 3-D register tile space on the Sun Ultra1/170 platform. The best implementation, shown in white at $m_0 = 2, n_0 = 8$, achieved 271 Mflop/s.

integer triplets (m_0, k_0, n_0).[3] Using heuristics based on the maximum number of registers available, this space was pruned to contain between 500 and 2500 reasonable implementations on each platform.

Figure 2 shows what fraction of implementations (y-axis) achieved what fraction of machine peak (x-axis). On the IBM RS/6000-590, a machine with generous memory bandwidth, 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 300 MHz 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 difficulty in finding the best implementation appears again in Figure 3. The plot shows a 2-D slice ($k_0 = 1$) of the 3-D tile space described above 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 can see that performance is not a smooth function of algorithmic details, making accurate sampling and interpolation of the space difficult. Like Figure 2, this motivates an exhaustive search.

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

3 Early Stopping Criteria

Unfortunately, exhaustive searches can be demanding, requiring dedicated machine time for extended periods. Thus, tuning systems prune the search spaces using heuristics based on properties of the code generator, machine, or routine being generated. We consider a complementary method for stopping a search early based soley on feedback data collected during the search. This provides a generic way to reduce search times when dedicated resources are limited or a near-optimal implementation is acceptable. Below, we describe one model and then demonstrate it on PHiPAC data.

3.1 A formal model

The search proceeds by first generating an implementation at random, then measuring its performance, and repeating these steps until the search space is exhausted. However, we would like to stop the search early if we find a near-optimal solution. The following formal model of the search captures this idea.

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 that we generate an implementation at random *without* replacement. Let X_t be a random variable corresponding to the observed performance. If we let M_t be the maximum observed performance at time t, i.e., $M_t = \max_{1 \leq i \leq t} X_i$, then we can ask about the chance that M_t is less than some threshold:

$$Pr[M_t \leq 1 - \epsilon] < \alpha, \qquad (1)$$

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 $1 - \epsilon$. A user could specify ϵ and α in hopes of ending a search at a time t smaller than N while still having some assurance of the quality of the implementation found.

To compute equation (1), observe that

$$\begin{aligned} Pr[M_t \leq x] &= Pr[X_1 \leq x, X_2 \leq x, \ldots, X_t \leq x] \\ &= \prod_{1 \leq r \leq t} p_r(x). \qquad (2) \end{aligned}$$

where $p_r(x) = Pr[X_r \leq x | X_1 \leq x, \ldots, X_{r-1} \leq x]$. We can compute $p_r(x)$ explicitly as follows, assuming no replacement:

$$p_r(x) = \begin{cases} 0 & n(x) < r \\ \frac{n(x)-r+1}{N-r+1} & n(x) \geq r \end{cases} \qquad (3)$$

Note that since $n(x) = N \cdot F(x)$, we cannot know its true value since we don't know the true distribution $F(x)$. However, we can use the t observed samples to approximate $F(x)$. One possible approximation is to use the empirical cdf (ecdf) based on the t samples:

$$\hat{F}_t(x) = \frac{\hat{n}_t(x)}{t} \qquad (4)$$

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.[4] Both this rescaling policy and, more generally, alternative forms of equation (4) are opportunities for further experimentation.

In summary, the procedure is as folows. A user or library designer specifies the search tolerance parameters ϵ and α. Then at each time t, the automated search system, using all observed samples so far, builds the ecdf in equation (4) to estimate equation (2). The search ends when the condition in equation (1) is satisfied.

[4]As we will show in the next section, this was a reasonable approximation on actual data. We are currently developing bounds on the quality of this approximation under rescaling, which we expect will be close to the known bounds on ecdf approximation developed by Kolmogorov and Smirnov [3].

3.2 Results with PHiPAC data

We apply the above model to the register tile space data on the platforms shown in Figure 2. Specifically, on each platform we simulate 200 searches following the model above, using the benchmark data collected during actual runs. We measure, as a function of the two tolerance values ϵ and α, statistics on (1) the stopping time, and (2) proximity of the performance found to the best.

Figures 4 and 5 show the results on 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 slightly pessimistic bound on the average value. The right plots show the average proximity of the implementation found to the best one (again, plus the standard deviation), as a fraction.

On the Pentium (Figure 4), setting $\epsilon = .05$ and $\alpha = .1$ (i.e., "find an implementation within 5% of the best with probability of error .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). By contrast, on the Cray T3E (Figure 5) 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.

There are many other possible combinatorial search algorithms. In prior work on PHiPAC [1], we experimented with search methods including random, ordered, best-first, simulated annealing. The OCEANS compiler project [11] has also reported on a quantitative comparison of these methods as well as others. In both projects, the random search was found to be comparable and easier to implement than the others. This is not surprising for the highly non-smooth spaces, littered with local minima, that we observe with PHiPAC data. The technique we are presently proposing adds user-interpretable bounds to the simple random method. This allows for an interesting possibility: if we allow the user 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

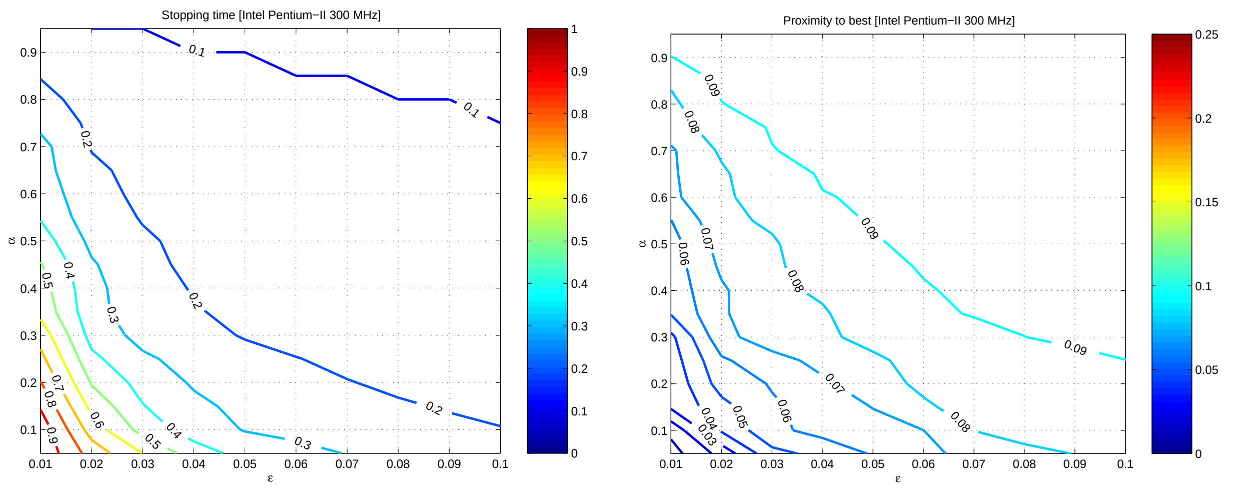


Figure 4: Average stopping time (left), as a fraction of the total search space, and proximity to the best performance (right), as the difference between normalized performance scores, on the 300 MHz Pentium-II class workstation as functions of the tolerance parameters ϵ (x-axis) and α (y-axis). Note that the values shown are mean *plus* standard deviation, to give an approximate upper-bound on the average case.

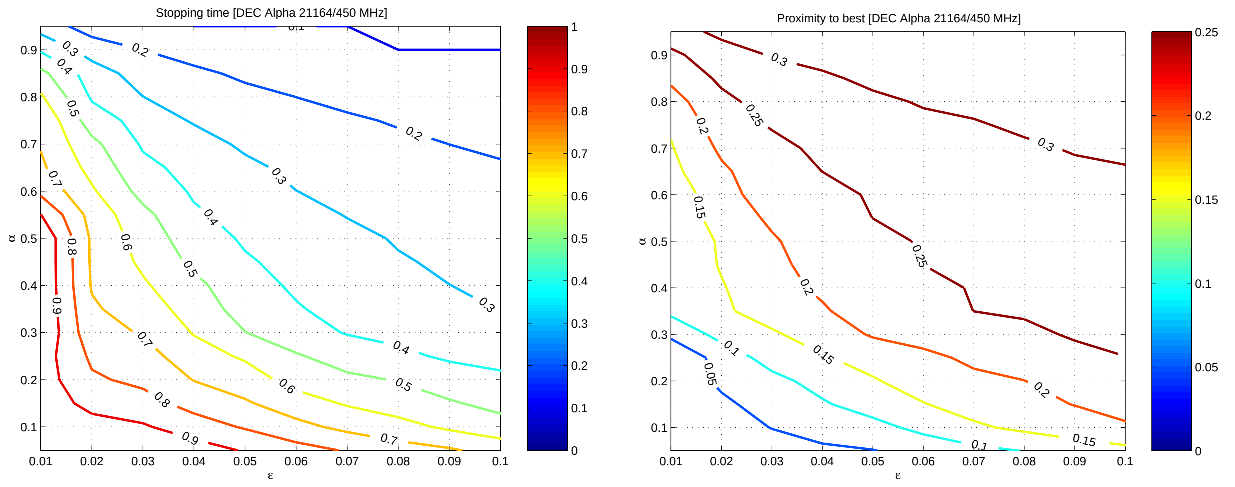


Figure 5: Same as Figure 4 for a uniprocessor Cray T3E node.

wish to build decision rules to select an appropriate implementation based on the run-time inputs.[5]

In this section, we describe a possible general framework for addressing this problem based on the modeling of feedback data, and highlight the key issues involved. We then discuss three methods for solving the problem in the context of the framework. We then present some preliminary results on PHiPAC data to illustrate how these issues interact.

4.1 A formal framework

Formally, we want to solve the following problem. **Given**:

- 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.

- A set of samples $S_0 = \{s_1, s_2, \ldots, s_n\}$ from the space S of all possible inputs (i.e., $S_0 \subseteq S$). We will treat each s_i as a d-dimensional real vector, where each element is some input parameter.

- The execution time $T(a, s)$ of algorithm a on input s, where $a \in A$ and $s \in S$.

Find:

- A decision function $f(s)$ that maps an input to the best implementation in A, i.e., $f : S \to A$.

The idea is to construct $f(s)$ using the performance data of the good implementations A on a sample of the inputs S_0. We will refer to S_0 as the *training set*. A geometric interpretation is that we would like to partition the input space by implementation, as illustrated in Figure 6 (left), using the performance on samples of the space. 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. One 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. We will consider two metrics. The first is the average misclassification rate, Δ_{miss}:

$$\Delta_{\text{miss}} = 1 - \frac{1}{|S'|} \sum_{s \in S'} \delta\left(f(s), \text{argmin}_{a \in A} T(a, s)\right) \quad (5)$$

where $\delta(a, a')$ is 1 if $a = a'$, and 0 otherwise. The sum is over some subset S' of the input space. 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. Therefore we will also use the following evaluation metric which is the average slow-down of the selected implementation relative to the best, Δ_{err}:

$$\Delta_{\text{err}} = \frac{1}{|S'|} \sum_{s \in S'} \left\{ \frac{T\left(f(s), s\right)}{\min_{a \in A} T(a, s)} - 1 \right\} \quad (6)$$

There are a number of additional issues that are worth mentioning but which we will not directly address in this paper. The first is how to choose A and S_0. The selection of implementations must be addressed by the library builder and in an automatic tuning system will be largely a function of the code generator. S_0 could be chosen randomly from S, or specified by the library builder according to some workload of interest. In addition, the accuracy of the prediction model f will depend on the size of S_0. We will assume that both A and S_0 are given. Another issue is the cost of evaluating $T(a, s)$. Here, we implicitly assume that this is feasible to do repeatedly, but of course this will depend upon the application.

As a concrete 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, as shown in Figure 6 (right). In 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 cache, and all larger sizes. The inputs to the implementation are M, K, and N, making the input space S three dimensional. We will refer to this example in the following sections.

4.2 A cost minimization method

Suppose that we 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. Let our decision function select the algorithm with the highest weight on input s:

$$f(s) = \text{argmax}_{a \in A} \left\{ w_{\theta_a}(s) \right\}. \quad (7)$$

One intuitive approach is to compute the weights so as to minimize the average execution time over the training set, expressed as the following cost function:

$$C(\theta_{a_1}, \ldots, \theta_{a_m}) = \sum_{a \in A} \sum_{s \in S_0} w_{\theta_a}(s) \cdot T(a, s). \quad (8)$$

Our goal is to minimize equation (8).

[5]This is similar in spirit to run-time specialization.

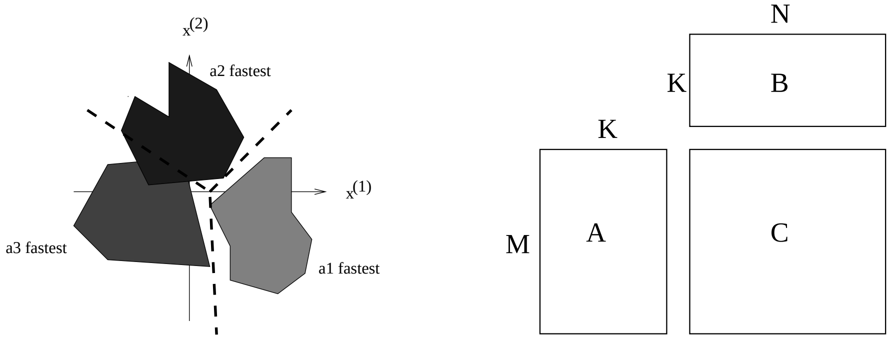


Figure 6: (*Left*) A hypothetical two-dimensional input space, partitioned by the algorithm. (*Right*) A matrix multiply operation $C = C + AB$ is specified by three dimensions, M, K, and N.

Of the many possible choices for w_{θ_a}, we choose the *softmax function* [10],

$$w_{\theta_a}(s) = \frac{e^{\theta_a^T s + \theta_{a,0}}}{\sum_{\theta_{a'}} e^{\theta_{a'}^T s + \theta_{a',0}}} \qquad (9)$$

where θ_a has the same dimensions as s, and $\theta_{a,0}$ is an additional parameter to estimate. Note that the weights vary smoothly with s, and that for fixed s the sum of the weights over all implementations is 1. Since the derivatives of the weights are easy to compute, we can estimate θ_a and $\theta_{a,0}$ by minimizing equation (8) numerically using Newton's method. A nice property of the weight function is that it makes f cheap to compute at run-time. Specifically, we only need to evaluate the argument of the exponent of the numerator in equation (9), which is linear in the dimension d of the input.

One drawback to this approach is that Newton's method can be sensitive to the initial guess making multiple training runs necessary. Another disadvantage is that the linear form $\theta_a^T s + \theta_{a,0}$ makes this formulation equivalent to asking for hyperplane boundaries to partition the space. However, hyperplanes may not be a good way to separate the input space as we shall see below. Other forms are certainly possible, but positing a form *a priori* can be difficult and could also complicate the numerical optimization.

4.3 A regression model

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, which we describe below, was originally proposed by Brewer [4].[6]

[6]A similar technique is also used in SPIRAL [15] to prune search spaces of FFT implementations.

Consider the case of conventional matrix multiply on square matrices of size $N \times N$. Since the time complexity is $O(N^3)$, we might postulate the running time of implementation a to be of the form

$$T_a(N) = \beta_3 N^3 + \beta_2 N^2 + \beta_1 N + \beta_0. \qquad (10)$$

Given sample running times on some inputs S_0, we can use standard regression techniques (i.e., least-squares fitting) to determine the β_k coefficients. The decision function is just

$$f(s) = \text{argmin}_{a \in A} T_a(s) \qquad (11)$$

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. For matrices of more general sizes, (M, K, N), we might hypothesize a model $T_a(M, K, N)$ with linear coefficients and the terms MKN, MK, KN, MN, M, K, N, and 1. We can even eliminate terms whose coefficients are "small" to reduce the run-time prediction costs. Another advantage is that training the model is likely to be faster than the cost-minimization method, depending on the model. Furthermore, no assumptions are being made about the geometry of the input space, as with the cost-minimization technique. However, a difficult disadvantage is that it may not be easy or obvious to postulate a run-time model with enough terms to capture the potentially complicated behavior of implementation running time.

4.4 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 [17].

Suppose that there are only two implementations and that we assign each point s_i in the training set a *class label* $y_i \in \{+1, -1\}$. The SV method constructs a classifier $L(s) \in \mathbb{R}$ whose sign is the predicted class label for input s. The specific form of the classifier is

$$L(s) = -b + \sum_{s_i \in S_0} \beta_i y_i K(s_i, s). \qquad (12)$$

The SV method determines the coefficients $\{\beta_i\}$ and b to maximize the minimum distance between the two classes[7] using the labeled training data. This requires solving a quadratic programming problem. The function $K(s_i, s)$ is any symmetric positive definite function, and is related to the shape of the boundary separating the classes [17].

Note that the classifier makes explicit use of the training data. This means that if there are many training points, then evaluating the predictor at run-time will be expensive. It turns out that for most applications, many of the β_i coeffients are zero which helps reduce the prediction cost.

For $m > 2$ implementations, we can separately build classifiers $L_{a_1}(s), \ldots, L_{a_m}(s)$ where $L_a(s)$ distinguishes between class a and all other classes. Each of these binary classifiers has its own set of $\{\beta_i\}, b$ parameters. The decision function becomes

$$f(s) = \text{argmax}_{a \in A} L_a(s). \qquad (13)$$

We include the SV method in our discussion because it is regarded as one of the most accurate prediction methods for a large class of practical problems. Thus, it provides a useful practical upper-bound on prediction accuracy. However, the primary disadvantages of this method are its high training costs (proportional to $|S_0|^2$) and prediction costs (proportional to $|S_0| \cdot d$), which may make it impractical for some applications.

4.5 Results with PHiPAC data

We offer a brief comparison of the three methods on the matrix multiply example described at the end of Section 4.1, which consists of three implementations tuned for different levels of cache. Recall that the input space is defined by all positive integer values of (M, K, N). We consider a 2-D cross-section of this 3-D space in which $M = N$ and $1 \le M, K, N \le 800$. We train each of the three methods on a subset of size 1000 points, and test the methods on a separate subset of size 500 points. We use the same training and test sets for each method. An example test set is shown in Figure 7 (left), color-coded by the fastest implementation (i.e., "truth" values). We can see that the space is divided in a complicated, non-linear fashion, although distinct regions are visible. Results are reported for the Sun Ultra-1/170.

[7]Formally, this is the *optimal margin* criterion [17].

			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%

Table 1: Comparison of the three prediction methods 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.

The predictions of the three methods on a sample test set are shown in Figures 7 (right) and 8. Qualitatively, we see that the cost-based method with its hyperplane boundaries is 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, though not perfect, 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. Note that the worst 20% and 50% numbers show that the regression method made slightly worse 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 SV method is about a 32x32 matrix multiply, which may make its use impractical when small sizes occur frequently.[8]

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; it does not stress all of the strengths and weaknesses of the three methods. Instead, our aim is simply to present the general framework and illustrate the issues on actual data. Moreover, there are many other possible models; our

[8]How to select and even combine run-time predictors automatically is another interesting area for exploration.

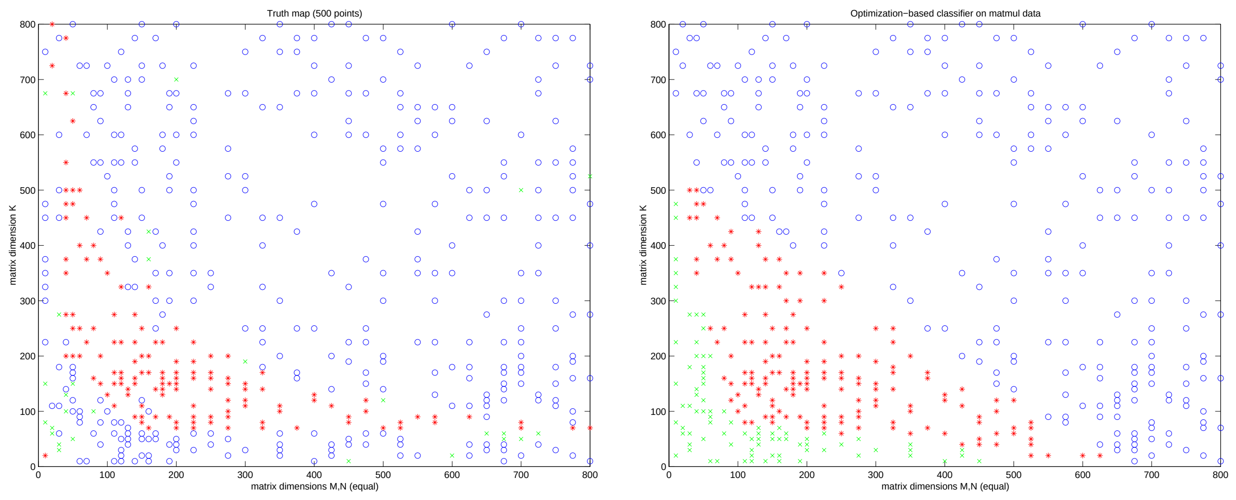


Figure 7: (*Left*) A "truth map" showing the regions in which particular implementations are fastest. A 500-point sample of a 2-D slice of the input space is shown. Red *'s correspond to an implementation with only register tiling, green x's have L1 cache tiling, and blue o's have L1 and L2 tiling. (*Right*) Prediction results for the cost-based method.

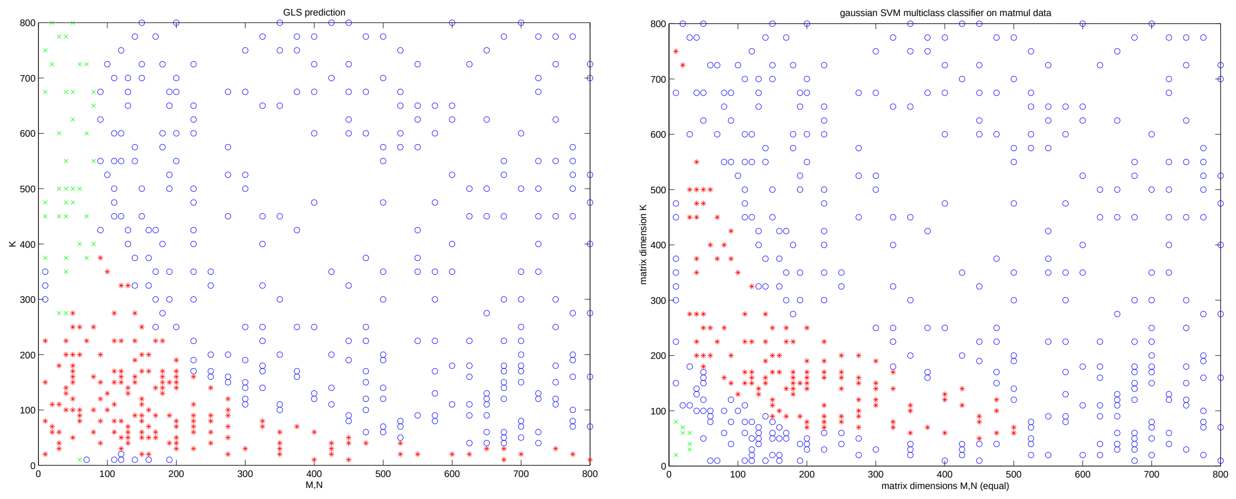


Figure 8: Prediction results (c.f. Figure 7)—regression method (*left*) and support-vector method (*right*).

examples offer a flavor for the role that statistical modeling of feedback 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 as an interesting and challenging problem in and of itself.

One of the challenges of the search problem is pruning the enormous possible implementation spaces. Even for a relatively simple but important operation such as matrix multiply, the design spaces are expansive. All of the existing tuning systems mentioned have shown the effectiveness of pruning the search spaces using problem-specific heuristics. Our black-box pruning method for stopping the search process early is another complementary technique. It has the nice properties of (1) incorporating feedback information in the form of benchmark data, and (2) providing users with a meaningful way (namely, via thresholds) to control the search procedure. The idea of empirical search to select optimizations has already found its way into a research compilation system [11]; this draws attention to the utility and importance of feedback-based search.

The other challenge is to find efficient ways to select implementations at run-time when several known implementations are available. Our aim in this paper has been to discuss a possible framework, using feedback-based sampling and classification, for attacking this problem in the context of automatic tuning systems. While we did not provide "the" solution, we did give several examples of parameterized black-box techniques. This brings together high performance software engineering with statistical modeling ideas. Other modeling techniques and software applications remain to be explored.

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. *Journal of the American Statistical Association*, 47:425–441, September 1952.

[4] E. Brewer. High-level optimization via automated statistical modeling. In *Symposium on Parallel Algorithms and Architectures, 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] J. Dongarra, J. D. Croz, I. Duff, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[7] 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.

[8] G. Haentjens. An investigation of recursive FFT implementations. Master's thesis, Carnegie Mellon University, 2000.

[9] 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.

[10] M. I. Jordan. Why the logistic function? Technical Report Report 9503, MIT, August 1995.

[11] 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.

[12] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[13] E. E. Rothberg, M. S. Lam, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS*, April 1991.

[14] D. A. Schwartz, R. R. Judd, W. J. Harrod, and D. P. Manley. VSIPL 1.0 API, March 2000. `www.vsipl.org`.

[15] 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.

[16] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective operations. In *Proceedings of Supercomputing 2000*, November 2000.

[17] V. N. Vapnik. *Statistical Learning Theory.* John Wiley and Sons, Inc., 1998.

[18] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, 1998.