Statistical Models for Automatic Performance Tuning

Richard Vuduc* James W. Demmel† Jeff Bilmes‡

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 by (1) generating a large number of possible implementations of a subroutine, and (2) selecting the fastest implementation by an exhaustive, empirical search. This paper presents quantitative data that motivates the development of such a search-based system, and discusses two problems which arise in the context of 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 address both problems by using statistical techniques to exploit the large amount of performance data collected during the search. 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

For a wide variety of applications in science, engineering, and information retrieval, among others, overall performance often depends critically on the underlying efficiency of only a few basic operations, or *computational kernels*. Familiar examples of such kernels include dense and sparse linear algebra routines, the fast Fourier transform and related signal processing kernels, and sorting, to name a few. Developers can build portable applications that also have portable performance, provided highly-tuned kernels are available on the platforms of interest to their users. Thus far, developers have relied on hardware vendors to provide these fast kernels, usually through some standard library interface. A wealth of popular libraries have been developed in this spirit, including the

*Computer Science Division, University of California at Berkeley, Berkeley, CA 94720 USA, `richie@cs.berkeley.edu`

†Computer Science Division and Dept. of Mathematics, University of California at Berkeley, Berkeley, CA 94720 USA, `demmel@cs.berkeley.edu`

‡Dept. of Electrical Engineering, University of Washington, Seattle, WA USA, `bilmes@ee.washington.edu`

Basic Linear Algebra Subroutines (BLAS) [17, 8, 7], the Vector and Signal Image Processing Library API [21], and the Message Passing Interface (MPI) for distributed parallel communications [1].

However, both construction and machine-specific hand-tuning of these libraries can be tedious and time-consuming tasks. First, program execution speed is sensitive to the detailed structure of the underlying machine. Modern machines employ deep memory hierarchies and microprocessors having long pipelines and intricate out-of-order execution schemes—understanding and accurately modeling performance on these machines can be extremely difficult. Second, modern compilers vary widely in the range and scope of analyses, transformations, and optimizations that they can perform, further complicating the task of developers who might wish to rely on the compiler to perform particular optimizations. Third, any given kernel could have many possible "reasonable" implementations which (a) are difficult to model because of the complexities of machine architecture and compiler, and (b) correspond to algorithmic variations that are beyond the scope of transformations compilers can presently apply. Fourth, the tuning process must be repeated for every platform.[1] Finally, performance may depend strongly on the program inputs, and this is typically known only at run-time. Thus, kernel performance could even depend on conditions not known when the library is initially built.

In reponse, several recent research efforts have sought to automate the tuning process using the following two-step methodology. First, rather than code a given kernel by hand for each computing platform of interest, these systems contain parameterized code generators that (a) encapsulate possible tuning strategies for the kernel, and (b) output an implementation, usually in a high-level language (like C or Fortran) for portability. By "tuning strategies" we mean that the generators can output implementations which vary by machine characteristics (e.g., different instruction mixes and schedules), optimization techniques (e.g., loop unrolling, cache blocking, the use of alternative data structures), run-time data (e.g., problem size), and kernel-specific transformations and algorithmic variants. Second, the systems tune for a particular platform by *searching*, i.e., varying the generators' parameters, benchmarking the resulting routines, and finally selecting the fastest implementation. In many cases it is possible to perform the potentially lengthy search process only once per platform. However, even the cost of more frequent compile-time, run-time, or hybrid compile-time/run-time searches can often be amortized over many uses of the kernel.

In this paper, we focus on the search task itself and argue that searching is an important and interesting area for research. We begin by showing empirically the difficulty of identifying the best implementation, even within a space of reasonable implementations (Section 2). This motivates searching and suggests the necessity of exhaustive search to *guarantee* the best possible implementation. Our argument also leads us to two search-related problems, both of which we treat in this paper using statistical modeling ideas. We refer to these problems

[1]By *platform* we mean a particular machine architecture and compiler.

as the *early stopping problem* and the problem of *run-time selection*.

The early stopping problem arises when it is not possible to perform an exhaustive search (Section 3). We ask whether we can perform a partial search and still provide some guarantees on the performance of the best implementation found. We answer the question by developing an informative, probabilistic stopping criteria that exploits performance data as it is collected during a search.

The run-time selection problem was first addressed by Brewer [5] (Section 4). Informally, suppose we are given a small number of implementations, each of which is fastest on some class of inputs. We assume we do not know the classes precisely ahead of time, but that we are allowed to collect a sample of performance of the implementations on a subset of all possible inputs. We then address the problem of automatically constructing a set of decision rules which can be applied at run-time to select the best implementation on any given input. We formulate the problem as a statistical classification task and illustrate the variety of models and techniques that can be applied within our framework.

Throughout we demonstrate our techniques on actual data collected from an existing tuning system: PHiPAC, the first system to propose the "generate and search" methodology [2, 3]. The PHiPAC system generates highly-tuned, BLAS compatible dense matrix multiply implementations, and a more detailed overview of PHiPAC appears in Section 2. Our use of PHiPAC is primarily to supply sample performance data on which we can demonstrate our methods. The ideas apply more generally, and in Section 5 we offer a brief survey of a number of other important automatic tuning systems and related compiler work.

2 The Case for Searching

In this section, we 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 can generally 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.

To see the necessity of search, consider the following experiment in which 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

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

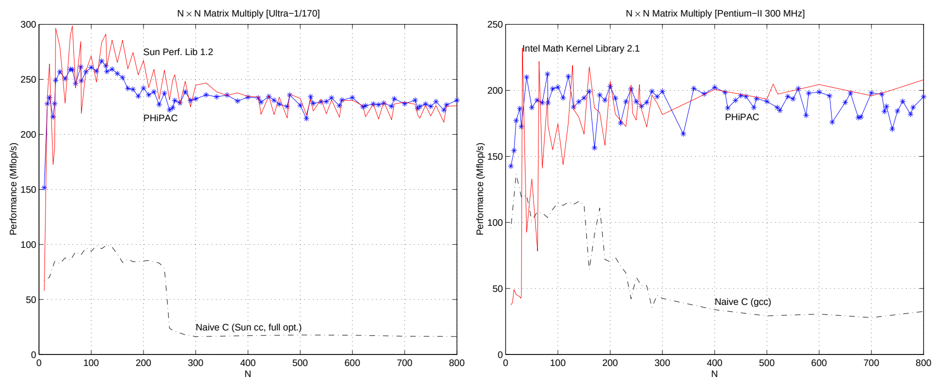


Figure 1: Performance (Mflop/s) on a square matrix multiply benchmark for the Sun Ultra 1/170 workstation (*left*) and a 300 MHz Pentium-II platform (*right*). The theoretical peaks are 333 Mflop/s and 300 Mflop/s, respectively.

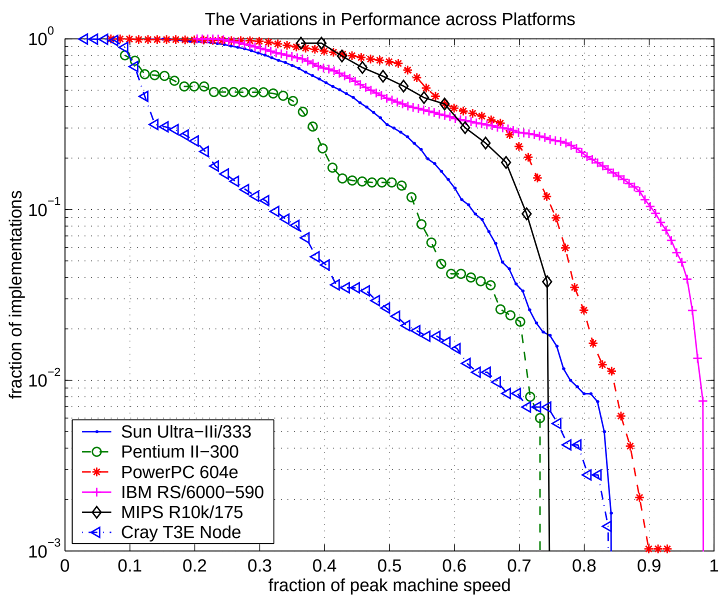
pruned to contain between 500 and 2500 reasonable implementations per platform. Figure 2 (*top*) shows what fraction of implementations (y-axis) achieved what fraction of machine peak (x-axis). On the IBM RS/6000, a machine with generous memory bandwidth, 5% of the implementations achieved at least 90% of the machine peak. By contrast, only 1.7% on a uniprocessor Cray T3E node, 4% on a Pentium-II, and fewer than 4% on a Sun Ultra 10/333 achieved more than 70% 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 more clearly in Figure 2 (*bottom*), which shows a 2-D slice ($k_0 = 1$) of the 3-D tile space on the Ultra 10. The plot is color coded from black=50 Mflop/s to white=620 Mflop/s. The lone white square at ($m_0 = 2, n_0 = 3$) 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, interpolation, or other modeling of the space difficult. Like Figure 2 (*top*), this motivates an exhaustive search.

3 An Early Stopping Criterion

While an exhaustive search can guarantee finding the best implementation,[3] such searches can be demanding, requiring dedicated machine time for long periods. If we assume that search will be performed only once per platform, then an

[3]By "guarantee" we mean within the space of implementations considered.



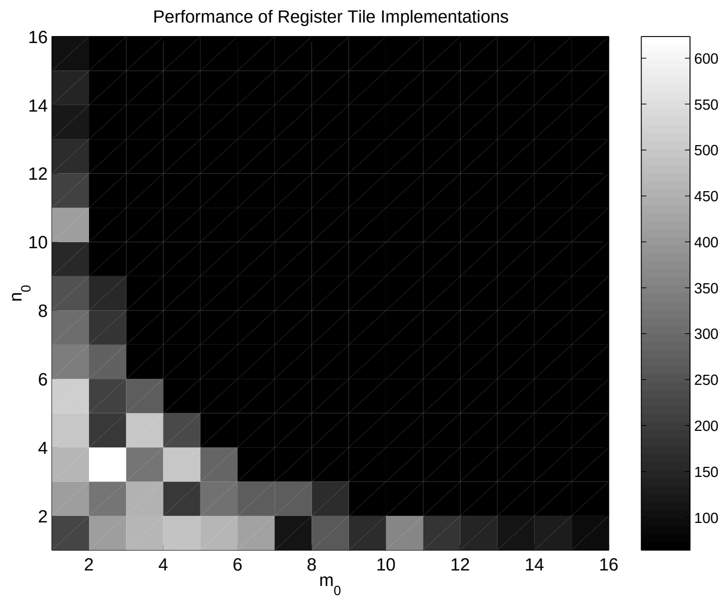


Figure 2: (*Top*) The fraction of implementations (y-axis) attaining at least a given level of peak machine speed (x-axis) on six platforms. (*Bottom*) A 2-D slice of the 3-D register tile space on the Sun Ultra 10/333 platform. Each square represents a particular implementation ($m_0 \times 1 \times n_0$ register tile), and is color-coded by performance (scale values are Mflop/s). The best implementation ($m_0 = 2, n_0 = 3$) achieved 620 Mflop/s out of a 667 Mflop/s peak performance. Note that the dark region in the upper-right corner of this plot has been pruned from the search space.

exhaustive search is justified. However, users today are more frequently running tuning systems themselves, or may wish to build kernels that are customized for their particular application or non-standard hardware configuration. Furthermore, we could imagine limited forms of search being employed at run-time, for instance, in the context of dynamic compilation systems.

Thus far, tuning systems have sought to prune the search spaces using heuristics and performance models specific to their code generators. Here, we consider a complementary method for stopping a search early based only on performance data gathered during the search. In particular, Figure 2 (*top*), described in the previous section, suggests that even when we cannot otherwise model the space, we can have access to some information as characterized by the statistical distribution of performance. Estimation of this distribution is the key idea behind our early stopping criterion.

3.1 A formal model

The following is a formal model of the search. 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:

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

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] 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.

3.2 Results with PHiPAC data

We apply the above model to the register tile space data for the platforms shown in Figure 2 (*top*). Specifically, we performed several hundred searches, measuring (1) the average stopping time, and (2) the average proximity in performance of the implementation found to the best found in an exhaustive search.

The results appear in Figures 3 and 4 for the Pentium and Cray T3E platforms, respectively. The top plots show the stopping time, and the bottom plots show the average proximity to the best. On the Pentium (Figure 3), setting $\epsilon = .05$ and $\alpha = .1$ (i.e., "find an implementation within 5% of the best with 10% uncertainty"), we see that, on average, the search ends after sampling 27% of the full space (top plot), having found an implementation within about 3.5% of the best (bottom plot).[5] By contrast, on the Cray T3E (Figure 4) where the best is difficult to find, the same tolerance values produce an implementation within about 4% of the best while still requiring exploration of 70% of the search space.[6] 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 [2], we experimented with search methods including random, ordered, best-first, simulated annealing. The OCEANS project [16] 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 random method while preserving the simplicity of its implementation. Furthermore, 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 exists. For some applications, however, several implementations may be "optimal" depending on the run-time inputs. In this section, we consider the run-time selection

[4]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 [4].

[5]The standard deviation on the stopping time at $\epsilon = .05$ and $\alpha = .1$ is 3.5%, and 3% for the proximity.

[6]The standard deviation of the stopping time is 10%, and 4% for the proximity.

problem: how can we automatically build decision rules to select the best implementation for a given input? Below, we formalize the problem and present three techniques to illustrate the variety of solutions possible.

4.1 A formal framework

We can pose the run-time selection problem as the following classification task. 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, as shown in Figure 5 (*left*). 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 to consider. Among them is the cost and complexity of building the predictor f. Another is the cost of evaluating $f(s)$—this cost 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, i.e., the fraction of test samples mispredicted. 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 will also consider the slow-down of the selected variant relative to the best, $\frac{t_{\text{selected}}}{t_{\text{best}}} - 1$, where t_{selected} and t_{best} are the execution times of the selected and best algorithms for a given input, respectively.

To illustrate the framework, we will refer to the following example in the remainder of this section. 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 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.[7] The inputs to each are M, K, and N, making the input space S three-dimensional. This is illustrated in Figure 5 (*right*).

4.2 A cost minimization method

One geometric approach is to first assume that there are boundaries, described parametrically, that divide the implementations, and then optimize the param-

[7]Recent work by Gunnels, et al., considers more sophisticated examples of deriving algorithmic variants for matrix multiply based on matrix shapes [12]. These variants would be suitable implementations A in our formal model.

eters with respect to an appropriate cost. Formally, 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, $f(s) = \text{argmax}_{a \in A} \{w_{\theta_a}(s)\}$. We can compute the weights so as to minimize the average execution time over the training set, i.e., find $\theta_{a_1}, \ldots, \theta_{a_m}$ that minimize

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

Of the many possible choices for w_{θ_a}, we choose the *softmax function* [15], $w_{\theta_a}(s) = \exp\left(\theta_a^T s + \theta_{a,0}\right) / Z$ where θ_a has the same dimensions as s, $\theta_{a,0}$ is an additional parameter to estimate, and Z is a normalizing constant. 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 is cheap to evaluate at run-time: the linear form $\theta_a^T s + \theta_{a,0}$ costs $O(d)$ operations to evaluate, where d is the dimension of the space. However, the primary disadvantage of this approach is that the same linear form makes this formulation equivalent to asking for hyperplane boundaries to partition the space. Hyperplanes may not be a good way to separate the input space as we shall see below. Of course, other forms are certainly possible, but positing their precise form *a priori* might not be obvious, and more complicated forms could also complicate the numerical optimization.

4.3 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 [5]. For matrix multiply on matrices of size $N \times N$, we might guess that the running time of implementation a will have the form

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

where we can use standard regression techniques to determine the coefficients β_k, given the running times on some sample inputs S_0. 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. For example, 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. Also, no explicit 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.

4.4 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 [24]. 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.[8] The result is a decision function $f(s)$.

The SV method is 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. This extra factor exists because some fraction of the training set must be retained to compute the decision function. Thus, evaluation of $f(s)$ is possibly much more expensive to calculate at run-time than either of the other two methods.

4.5 Results with PHiPAC data

We offer a brief comparison of the three methods on the matrix multiply example described in Section 4.1, using PHiPAC to generate the implementations on a Sun Ultra 1/170 workstation.

To evaluate the prediction accuracy of the three run-time selection algorithms, we conducted the following experiment.[9] First, we considered a 2-D cross-section of the full 3-D input space in which $M = N$ and $1 \leq M, K, N \leq 800$. We selected 10 disjoint subsets of points in this space, where each subset contained 1500 points chosen at random.[10] Then we further divided each subset into 1000 training points and 500 testing points. We then trained and tested all three run-time selection algorithms, measuring the prediction accuracy on each subset. We report the overall misclassification rate for each selection algorithm as the average over all of the subsets.

In Figure 6, we show an example of a 500-point testing set where each point is color-coded by the implementation which ran fastest. We note that the implementation which was fastest on the majority of inputs is the default implementation generated by PHiPAC, and contains full tiling optimizations. Thus, a useful reference is a *baseline predictor* which always chooses this implementation. The misclassification rate of this predictor was 24%.

The predictions of the three methods on a sample test set are shown in Figures 7–9. We see qualitatively 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. Quantatively,

[8] Formally, this is the *optimal margin* criterion [24].

[9] The cost-minimization and regression models were implemented as described. For the support vector method, we used Platt's *sequential minimal optimization* algorithm with a Gaussian kernel [20]. We built multiclass classifiers from ensembles of binary classifiers, as described by [24].

[10] The points were chosen from a distribution with a bias toward small sizes.

the misclassification rates are 31% for the cost-based predictor, 34% for the regression predictor, and 12% for the support vector predictor. Only the support vector predictor outperformed the baseline predictor.

However, misclassification rates may seem to be too strict a measure of prediction performance, since we may be willing to tolerate some penalties to obtain a fast prediction. Figure 10 shows the distribution of slow-downs due to mispredictions. Observe that the baseline predictor performs fairly well in the sense that fewer than 1% of its mispredictions resulted in more than a 7% slow-down in execution time. The distribution of slow-downs for the baseline predictor indicates that the performance of the three tuned implementations is fairly similar. Thus, we do not expect to improve upon the baseline predictor much, and this is borne out by observing the slow-down distributions of the cost-minimization and regression predictors. Nevertheless, we also see that for slow-downs of up to 10%, the support vector predictor shows a significant improvement over the baseline predictor. Depending upon an application's performance requirements, this may justify the use of the more complex statistical model.

Although the baseline predictor is, on average, likely to be sufficient for this particular example, the distribution of slow-downs nevertheless show a number of interesting features. If we compare the regression and cost-minimization methods, we see a cross-over point: even though the overall misclassification rate for the cost-minimization predictor is lower than the regression predictor, the tail of the distribution for the regression predictor becomes much smaller. A similar cross-over exists between the baseline and support vector predictors.

In terms of prediction times (i.e., the time to evaluate $f(s)$), both the regression and cost-minimization methods lead to reasonably fast predictors. Prediction times were roughly equivalent to the execution time of a 3×3 matrix multiply. By contrast, the prediction cost of the SVM is about a 64×64 matrix multiply, which would 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, and we see that it does not stress all of the strengths and weaknesses of the three methods. Furthermore, a user or application might care about a region of the full input-space which is different from the one used in our example. Instead, our primary 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 of the role that statistical modeling of performance data can play.

5 Related Work: A Short Survey of Tuning Systems

To give a sense of the recent interest in automatic tuning systems, we mention briefly a number of important tuning systems in addition to PHiPAC. In signal processing, FFTW was the first tuning system for various flavors of the discrete Fourier transform [11]. FFTW is notable for its use of a high-level, symbolic representation of the FFT algorithm, as well as its run-time search which saves and uses performance history information. In dense linear algebra, ATLAS extends PHiPAC ideas and applicability to all of the BLAS [26].[11] The Sparsity system targets sparse linear algebra, and in particular produces tuned sparse matrix-vector multiply implementations specific not only to the machine architecture but also to a given sparse matrix [14]. There have also been additional efforts in signal processing which build on the FFTW ideas. The SPIRAL system is built on top of a symbolic algebra system, and uses a novel search method based on genetic algorithms [13, 22]. The UHFFT system is an alternative implementation of FFTW which includes a different implementation of the code generator which contains additional discrete Fourier transform variants [18]. The Pan system uses an FFTW-like specialized compiler to generate optimized implementations of kernels for image synthesis and manipulation [9]. In the area of parallel communications, Vadhiyar, et al., explore automatically tuning MPI collective operations [23].

All of these systems employ sophisticated code generators which 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. Thus, each uses heuristics, properties, and performance models specific to its code generator to prune the search spaces. This will always be a necessary and effective approach in practice, and we view our techniques as complementary methods 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 [16] are integrating an empirical search procedure into a compiler for embedded systems. They consider the use of models that predict implementation performance to identify which implementations to benchmark. For the run-time selection problem, a prototype compiler that performs run-time profiling and selection from among multiple implementations exists as part of the ADAPT project [25]. Mitchell considers static compile-time modeling methods for comparing and selecting from among various loop transformations [19]. Those models use empirical search data collected in a one-time profiling of the machine.

[11]It is worth noting that both FFTW and ATLAS are now included with Matlab.

6 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. This article uses statistical methods to address some of the challenges which arise.

One challenge is pruning the enormous implementation spaces. Existing tuning systems have shown the effectiveness of pruning these spaces using problem-specific heuristics and performance models; our statistical model for stopping a search early is a complementary technique. It has the nice properties of (1) making very few assumptions about the performance of the implementations, (2) incorporating performance feedback data, and (3) providing users with a meaningful way to control the search procedure (namely, via probabilistic thresholds).

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.

However, many other modeling techniques remain to be explored. For instance, the early stopping problem can be posed as a similar problem which has been treated extensively in the statistical literature under the theory of optimal stopping [6]. Problems treated in that theory can incorporate the cost of the search itself, which would be especially useful if we wished to perform searches not just at build-time, as we consider here, but at run-time—for instance, in the case of a just-in-time or other dynamic compilation system.

In the case of run-time selection, we make implicit geometric assumptions about inputs to the kernels being points in some continuous space. However, inputs could also be binary flags or other arbitrary discrete labels. This can be handled in the same way as in the traditional classification settings, namely, either by finding mappings from the discrete spaces into continuous (feature) spaces, or by using statistical models with discrete probability distributions (e.g., using graphical models [10]).

In short, this work connects high performance software engineering with statistical modeling ideas. Furthermore, as mentioned in Section 5, the idea of searching is being incorporated into more general purpose compilation systems. This indicates another future direction for the search process, and emphasizes the relevance of search beyond specialized tuning systems.

Acknowledgements

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

References

[1] The Message Passing Interface Standard. `www.mpi-forum.org`.

[2] 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.

[3] 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.

[4] 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.

[5] E. Brewer. High-level optimization via automated statistical modeling. In *Sym. Par. Alg. Arch., Santa Barbara, California*, July 1995.

[6] Y. S. Chow, H. Robbins, and D. Siegmund. *Great Expectations: The Theory of Optimal Stopping.* Houghton-Mifflin, 1971.

[7] 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.

[8] 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.

[9] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *LNCS Proc. of the Semantics, Application, and Implementation of Code Generators (SAIG) Workshop*, volume 1924. Springer-Verlag, 2000.

[10] B. Frey. *Graphical Models for Machine Learning and Digital Communications.* MIT Press, 1998.

[11] 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.

[12] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 51–60. Springer, May 2001.

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

[14] 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.

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

[16] 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.

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

[18] D. Mirkovic, R. Mahasoom, and L. Johnsson. An adaptive software library for fast fourier transforms. In *Proceedings of the International Conference on Supercomputing*, pages 215–224, May 2000.

[19] N. Mitchell. *Guiding Program Transformations with Modal Performance Models.* Ph.D. dissertation, University of California, San Diego, 2000.

[20] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods — Support Vector Learning*, Jan 1999.

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

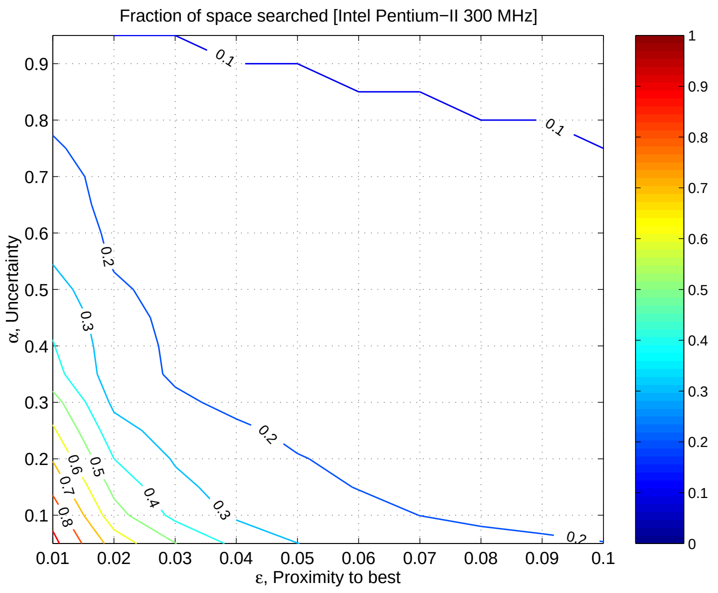
[22] B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proc. of the 17th Int'l Conf. on Mach. Learn.*, 2000.

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

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

[25] M. J. Voss and R. Eigenmann. ADAPT: Automated De-coupled Adaptive Program Transformation. In *Proceedings of the International Conference on Parallel Processing*, August 2000.

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



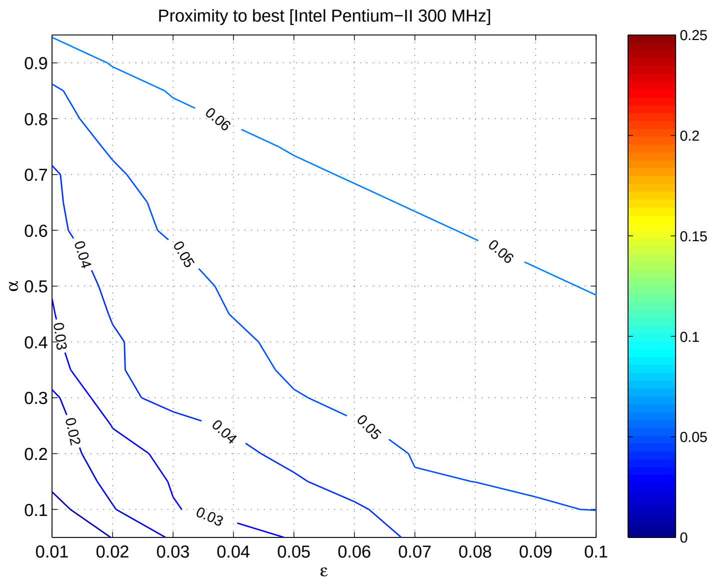
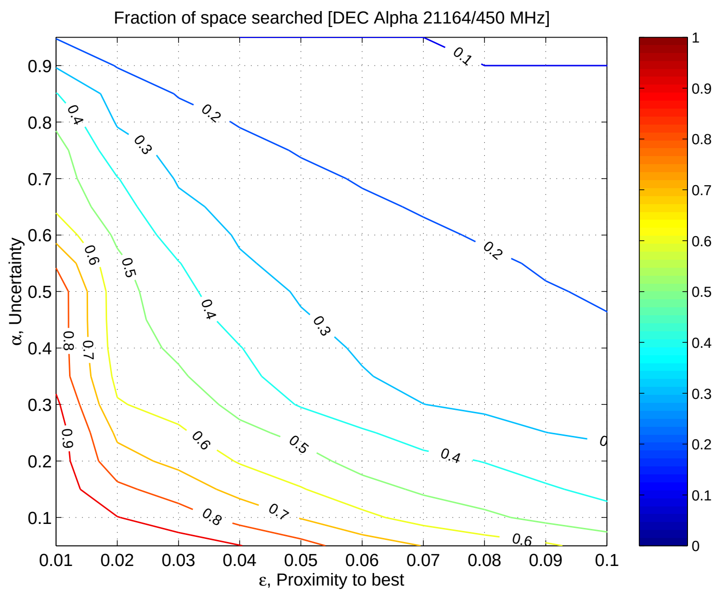


Figure 3: Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the 300 MHz Pentium-II class workstation as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).



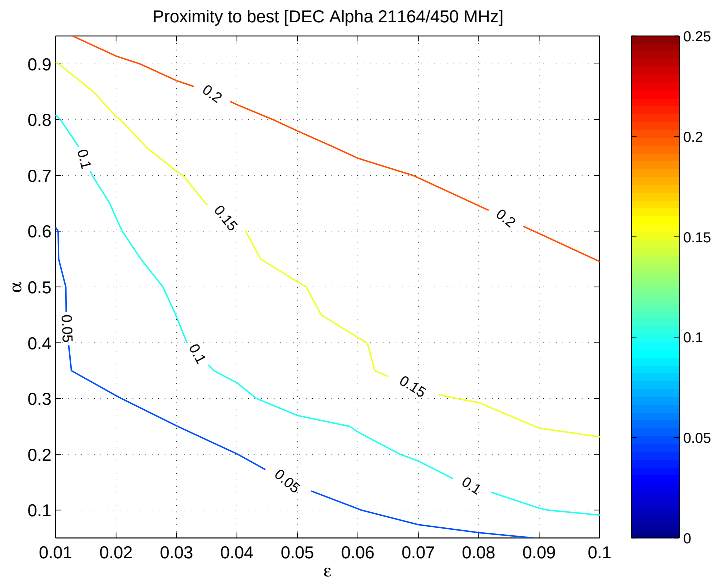
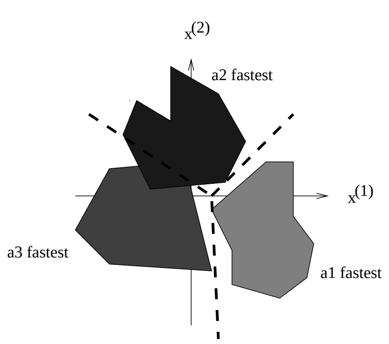


Figure 4: Same as Figure 3 for a uniprocessor Cray T3E node.



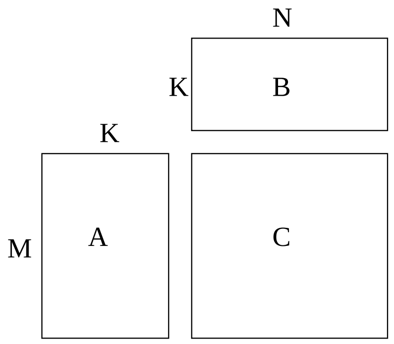


Figure 5: (*Left*) Geometric interpetation of the run-time selection problem: A hypothetical 2-D input space in which one of three algorithms runs fastest in some region of the space. Our goal is to partition the input space by algorithm. (*Right*) The matrix multiply operation $C = C + AB$ is specified by three dimensions, M, K, and N.

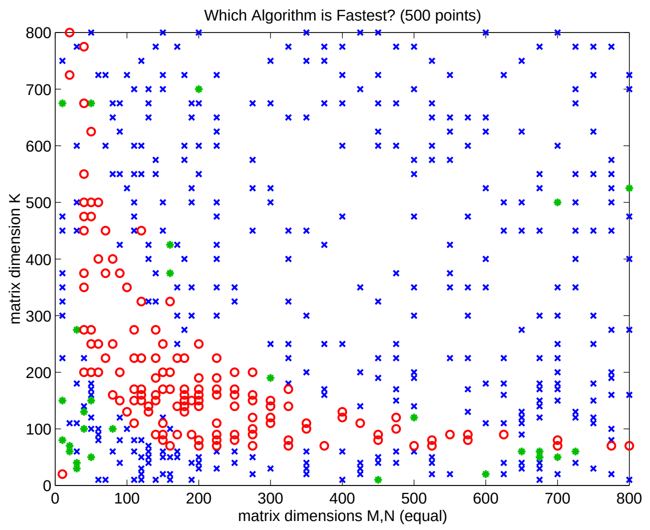


Figure 6: A "truth map" showing the regions in which particular implementations are fastest. The points shown represent a 500-point sample of a 2-D slice (specifically, $M = N$) of the input space. An implementation with only register tiling is shown with a red `O`; one with L1 and register tiling is shown with a green `*`; one with register, L1, and L2 tiling is shown with a blue `X`. The baseline predictor always chooses the blue algorithm. The average misclassification rate for this baseline predictor is 24.5%.

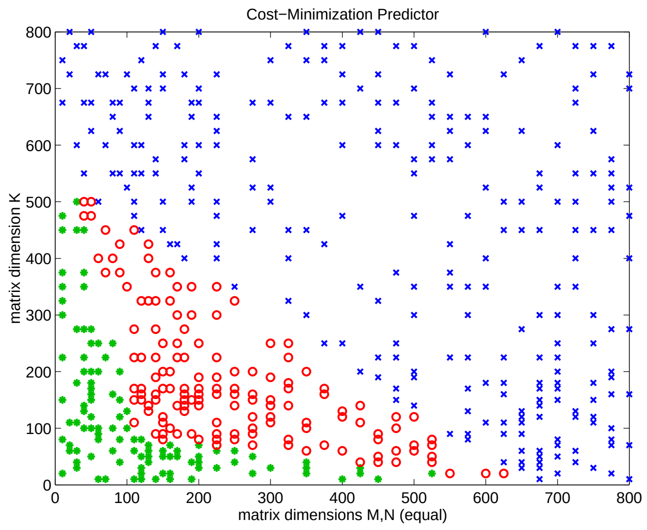


Figure 7: Example of the prediction results for the cost-based method on the same 500 point sample shown in Figure 6. The average misclassification rate for this predictor is 31.6%.

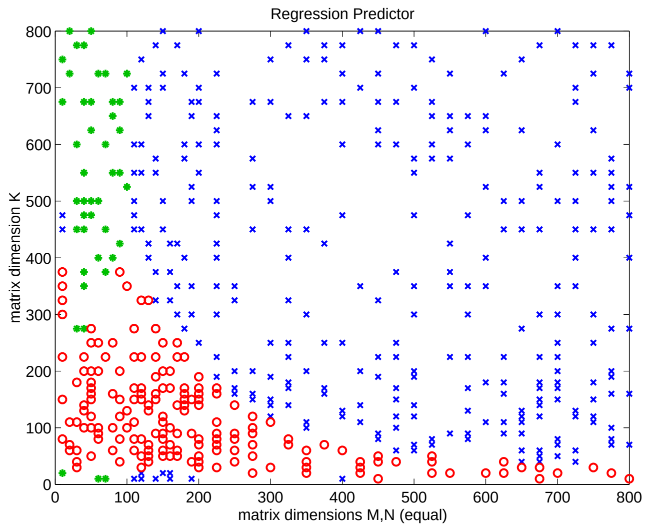


Figure 8: Example of the prediction results for the regression predictor. The average misclassification rate for this predictor was 34.5%.

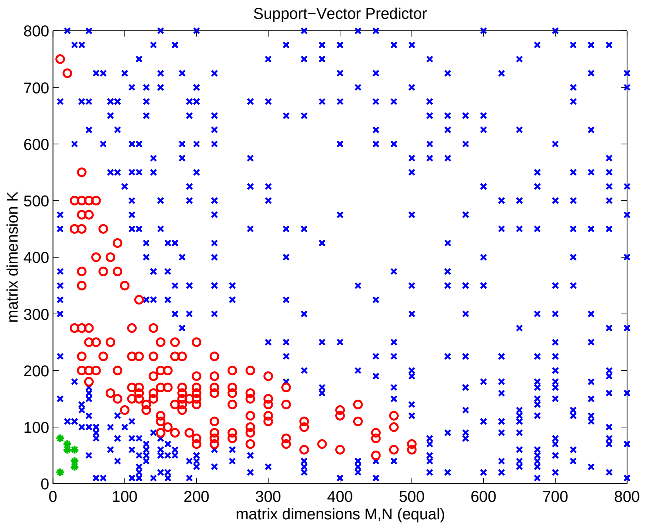


Figure 9: Prediction results support-vector predictor. The average misclassification rate for this predictor was 12%.

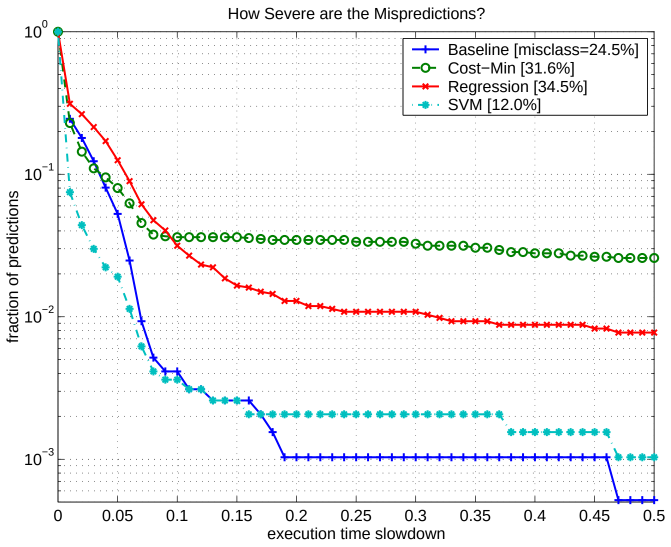


Figure 10: Each line corresponds to the distribution of slow-downs due to mispredictions for a particular predictor. A point on a given line indicates what fraction of predictions (y-axis) resulted in more than a particular slow-down (x-axis). Note the logarthmic scale on the y-axis.