Statistical Models for Empirical Search-Based Performance Tuning

Richard Vuduc* James W. Demmel† Jeff Bilmes‡

Abstract

Achieving peak performance from the computational kernels that dominate application performance often 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, reasonable implementations of a kernel, and (2) selecting the fastest implementation by a combination of heuristic modeling, heuristic pruning, and empirical search (*i.e.*, actually running the code). This paper presents quantitative data that motivates the development of such a search-based system, using dense matrix multiply as a case study. The statistical distributions of performance within spaces of reasonable implementations, when observed on a variety of hardware platforms, lead us to pose and address two general problems which arise during the search process. 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 when the space of inputs can be described by continuously varying features. We address both problems by using statistical modeling techniques that exploit the large amount of performance data collected during the search. We demonstrate these methods on actual performance data collected by the PHiPAC tuning system for dense matrix multiply.

We close with a survey of recent projects that use or otherwise advocate an empirical search-based approach to code generation and algorithm selection, whether at the level of computational kernels, compiler and run-time systems, or problem-solving environments. Collectively, these efforts suggest a number of possible software architectures for constructing platform-adapted libraries and applications.

*Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA, richie@cs.berkeley.edu

†Computer Science Division, Department of Electrical Engineering and Computer Sciences, and Department of Mathematics, University of California at Berkeley, Berkeley, CA 94720 USA, demmel@cs.berkeley.edu

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

1 Introduction

This paper presents quantitative data that motivates the use of the platform-specific, empirical search-based approach to code generation being adopted by a number of current *automatic performance tuning systems*. Such systems address the problem of how to generate highly efficient code for a number of basic computational kernels that dominate the performance of applications in science, engineering, and information retrieval, among others. The driving idea behind these systems is the use of *empirical search* techniques, *i.e.*, actually running and timing candidate implementations. Familiar examples of kernels include dense and sparse linear algebra routines, the fast Fourier transform and related signal processing kernels, and sorting, to name a few. Using dense matrix multiply as a case study, we show that performance can be surprisingly difficult to model on modern cache-based superscalar architectures, suggesting the use of an empirical approach to code generation. Moreover, we use this data to show how statistical techniques can be applied to help solve common, general problems arising in search-based tuning systems.

The construction and machine-specific hand-tuning of computational kernels can be tedious and time-consuming tasks because performance is a complex function of many factors, including the platform (*i.e.*, machine and compiler), kernel, and input data which may be known only at run-time. First, note that 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 either to perform particular optimizations or even to know which transformations will be most profitable on a particular kernel. 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, performance may depend strongly on the program inputs. If such inputs are known only at run-time, purely static techniques to code generation may be limited. To make matters worse, the process of performance tuning—and hence, of understanding the complex interactions among these various factors—must be repeated for every new platform.

Fast implementations of some kernels are available via hand-tuned libraries provided by hardware vendors. A number of standard and widely used library interfaces exist, including the Basic Linear Algebra Subroutines (BLAS) [27], and the Message Passing Interface (MPI) for distributed parallel communications [35]. However, the number of such interfaces is growing to cover additional application areas, such as the Sparse BLAS standard for operations on sparse matrices [13], and the Vector and Signal Image Processing Library API [76]. The cost of providing hand-tuned implementations of all these kernels for all hardware platforms, given the potential complexity of tuning each kernel, is

likely only to increase with time.

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) in order to leverage existing compiler instruction-scheduling technology. By "tuning strategies" we mean that the generators can output implementations which vary by low-level machine-specific 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* the space of implementations defined by the generator. Typical tuning systems search using a combination of heuristic performance modeling and empirical evaluation (*i.e.*, actually running code for particular implementations). 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.

This paper focuses on the search task itself and argues that searching is an effective means by which to achieve near-peak performance, and is therefore an important area for research. Indeed, search-based methods have proliferated in a variety of computing contexts, including applications, compilers, and run-time systems, as we discuss in Section 5. We begin by showing empirically the difficulty of identifying the best implementation, even within a space of reasonable implementations for the well-studied kernel, dense matrix multiply (Section 2). While a variety of sophisticated transformations and static models have been developed for matrix multiply, the data we present motivate *searching*, and furthermore suggest the necessity of exhaustive search to *guarantee* the best possible implementation.

However, exhaustive searches are frequently infeasible, and moreover, performance can depend critically on input data that may only be known at run-time. We address these two search-related problems in this paper using statistical modeling techniques. Specifically, we propose solutions to the *early stopping problem* and the problem of *run-time implementation selection*. Our techniques are designed to complement existing methods developed for these problems.

The early stopping problem arises when it is not possible to perform an exhaustive search (Section 3). Existing tuning systems use a combination of kernel-specific, heuristic performance modeling and empirical search techniques to avoid exhaustive search. We present a complementary technique, based on a simple statistical analysis of the data gathered while a search is on-going. Our method allows us to perform only a partial search while still providing an estimate on the performance of the best implementation found.

The run-time selection problem for computational kernels was first posed by Rice [73], and again more recently by Brewer [14] (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.

Our analyses are based on data collected from an existing tuning system, PHiPAC [10, 11]. PHiPAC 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 the statistical methods of this paper. (For complete implementations of the BLAS, we recommend the use of either the ATLAS tuning system or any number of existing hand-tuned libraries when available. In particular, ATLAS improves on PHiPAC ideas, extending their applicability to the entire BLAS standard [93].)

The ideas and timing of this paper are indicative of a general trend in the use of search-based methods at various stages of application development. We review the diverse body of related research projects in Section 5. While the present study adopts the perspective of tuning specific computational kernels, for which it is possible to obtain a significant fraction of peak performance by exploiting all kernel properties, the larger body of related work seeks to apply the idea of searching more generally: within the compiler, within the run-time system, and within specialized applications or problem-solving environments. Collectively, these studies imply a variety of general software architectures for empirical search-based tuning.

2 The Case for Searching

We motivate the need for empirical search methods using matrix multiply performance data as a case study. We show that, within a particular space of performance optimization (tuning) parameters, (1) performance can be a surprisingly complex function of the parameters, (2) performance behavior in these spaces varies markedly from architecture to architecture, and (3) the very best implementation in this space can be hard to find. Taken together, these observations suggest that a purely static modeling approach will be insufficient to find the best choice of parameters.

2.1 Factors influencing matrix multiply performance

We briefly review the classical optimization strategies for matrix multiply, and make a number of observations that justify some of the assumptions of this paper (Section 2.2 in particular). Roughly speaking, the optimization techniques fall into two broad categories: (1) cache- and TLB-level optimizations, such

as cache tiling (blocking) and copy optimization (*e.g.*, as described by Lam [53] or by Goto with respect to TLB considerations [37]), and (2) register-level and instruction-level optimizations, such as register-level tiling, loop unrolling, software pipelining, and prefetching. Our argument motivating search is based on the surprisingly complex performance behavior observed within the space of register- and instruction-level optimizations, so it is important to understand what role such optimizations play in overall performance.

For cache optimizations, a variety of sophisticated static models have been developed for kernels like matrix multiply to help understand cache behavior, to predict optimal tile sizes, and to transform loops to improve temporal locality [29, 36, 53, 95, 18, 59, 16]. Some of these models are expensive to evaluate due to the complexity of accurately modeling interactions between the processor and various levels of the memory hierarchy [62].[1] Moreover, the pay-off due to tiling, though significant, may ultimately account for only a fraction of performance improvement in a well-tuned code. Recently, Parello, *et al.*, showed that cache-level optimizations accounted for 12–20% of the possible performance improvement in a well-tuned dense matrix multiply implementation on an Alpha 21264 processor based machine, and the remainder of the performance improvement came from register- and instruction-level optimizations [66].

To give some additional intuition for how these two classes of optimizations contribute to overall performance, consider the following experiment comparing matrix multiply performance for a sequence of $n \times n$ matrices. Figure 1 shows examples of the cumulative contribution to performance (Mflop/s) for matrix multiply implementations in which (1) only cache tiling and copy optimization have been applied, shown by solid squares, and (2) applying the register-level tiling, software pipelining, and prefetching have been applied in conjunction with these cache optimizations, shown by triangles. These implementations were generated with PHiPAC, discussed below in more detail (Section 2.2). In addition, we show the performance of a reference implementation consisting of 3 nested loops coded in C and compiled with full optimizations using a vendor compiler (solid line), and a hand-tuned implementation provided by the hardware vendor (solid circles). The platform used in Figure 1 (*top*) is a workstation based on a 333 MHz Sun Ultra 2i processor with a 2 MB L2 cache and the Sun v6 C compiler, and in Figure 1 (*bottom*) is an 800 MHz Intel Mobile Pentium III processor with a 256 KB L2 cache and the Intel C compiler. On the Pentium III, we also show the performance of the hand-tuned, assembly-coded library by Goto [37], shown by asterisks.

On the Ultra 2i, the cache-only implementation is 17× faster than the reference implementation for large n, but only 42% as fast as the automatically generated implementation with both cache- and register-level optimizations. On the Pentium III, the cache-only implementation is 3.9× faster than the reference, and about 55–60% of of the register and cache optimized code. Furthermore,

[1] Indeed, in general it is even hard to approximate the optimal placement of data in memory so as to minimize cache misses. Recently, Petrank and Rawitz have shown the problem of optimal cache-conscious data placement to be in the same hardness class as the minimum coloring and maximum clique problems [67].

the PHiPAC-generated code matches or closely approaches that of the hand-tuned codes. On the Pentium III, the PHiPAC routine is within 5–10% of the performance of the assembly-coded routine by Goto at large n [37]. Thus, while cache-level optimizations significantly increase performance over the reference implementation, applying them together with register- and instruction-level optimizations is critical to approaching the performance of hand-tuned code.

These observations are an important part of our argument below (Section 2.2) motivating empirical search-based methods. First, we focus exclusively on performance in the space of register- and instruction-level optimizations on in-cache matrix workloads. The justification is that this class of optimizations is essential to achieving high-performance. Even if we extend the estimate by Parello, *et al.*—specifically, from the observation that 12–20% of overall performance is due to cache-level optimizations, to 12–60% based on Figure 1—there is still a considerable margin for further performance improvements from register- and instruction-level optimizations. Second, we explore this space using the PHiPAC generator. Since PHiPAC-generated code can achieve good performance in practice, we claim this generator is a reasonable one to use.

2.2 A needle in a haystack: the need for search

To demonstrate the necessity of search-based methods, we next examine performance within the space of register-tiled implementations. The automatically generated implementations of Figure 1 were created using the parameterized code generator provided by the PHiPAC matrix multiply tuning system [10, 11]. (Although PHiPAC is no longer actively maintained, for this paper the PHiPAC generator has been modified to include some software pipelining styles and prefetching options developed for the ATLAS system [93].) This generator implements register- and instruction-level optimizations including (1) register tiling where non-square tile sizes are allowed, (2) loop unrolling, and (3) a choice of software pipelining strategies and insertion of prefetch instructions. The output of the generator is an implementation in either C or Fortran in which the register-tiled code fragment is fully unrolled; thus, the system relies on an existing compiler to perform the instruction scheduling.

PHiPAC searches the combinatorially large space defined by possible optimizations in building its implementation. 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 search-space pruning heuristics, searches can generally take many hours or even a day depending on the user-selectable thoroughness of the search. Nevertheless, as we suggest in Figure 1, performance can be comparable to hand-tuned implementations.

Consider the following experiment in which we fixed a particular software pipelining strategy and explored the space of possible register tile sizes on 11 different platforms. As it happens, this space is three-dimensional and we index it by integer triplets (m_0, k_0, n_0).[2] Using heuristics based on the maximum

[2] By dimensional constraints on the operation $C \leftarrow AB$, we choose an $m_0 \times k_0$ tile for the

number of registers available, this space was pruned to contain between 500 and 10000 reasonable implementations per platform.

Figure 2 (*top*) shows what fraction of implementations (y-axis) achieved at least a given fraction of machine peak (x-axis), on a workload in which all matrix operands fit within the largest available cache. On two machines, a relatively large fraction of implementations achieve close to machine peak: 10% of implementations on the Power2/133 and 3% on the Itanium 2/900 are within 90% of machine peak. By contrast, only 1.7% on a uniprocessor Cray T3E node, 0.2% on the Pentium III-M/800, and fewer than 4% on a Sun Ultra 2i/333 achieved more than 80% of machine peak. And on a majority of the platforms, fewer than 1% of implemenations were within 5% of the best. Worse still, nearly 30% of implementations on the Cray T3E ran at less than 15% of machine peak. Two important ideas emerge from these observations: (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 2i/333. The plot is color coded from dark blue=66 Mflop/s to red=615 Mflop/s, and the lone red square at ($m_0 = 2, n_0 = 3$) was the fastest. The black region in the upper-right of Figure 2 (*bottom*) was pruned (*i.e.*, not searched) based on the number of registers. We see that performance is not a smooth function of algorithmic details as we might have expected. Thus, accurate sampling, interpolation, or other modeling of this space is difficult. Like Figure 2 (*top*), this motivates empirical search.

3 A Statistical Early Stopping Criterion

While an exhaustive search can guarantee finding the best implementation within the space of implementations considered, 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 exhaustive search may be 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, the notion of run-time searching, as pursued in dynamic optimization systems (Section 5) demand extensive search-space pruning.

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 do have access to the statistical distribution of performance. On-line estimation of this distribution is the key idea behind the following early stopping

A operand, a $k_0 \times n_0$ tile for the B operand, and a $m_0 \times n_0$ tile for the C operand.

criterion. This criterion allows a user to specify that the search should stop when the probability that the performance of the best implementation observed is approximately within some fraction of the best possible within the space.

3.1 A formal model and stopping criterion

The following is a formal model of the search process. Suppose there are N possible implementations. When we generate implementation i, we measure its performance x_i. Assume that each x_i is normalized so that $\max_i x_i = 1$. (We discuss the issue of normalization further in Section 3.1.2.) 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 an integer 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, and furthermore let $M_t = \max_{1 \leq i \leq t} X_i$ be the random variable corresponding to the maximum observed performance up to t.

We can now ask the following question at each time t: what is the probability that M_t is at least $1 - \epsilon$, where ϵ is chosen by the user or library developer based on performance requirements? When this probability exceeds some desired threshold $1 - \alpha$, also specified by the user, then we stop the search. Formally, this stopping criterion can be expressed as follows:

$$Pr[M_t > 1 - \epsilon] \quad > \quad 1 - \alpha$$

or, equivalently,

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

Let $G_t(x) = Pr[M_t \leq x]$ be the cdf for M_t. We refer to $G_t(x)$ as the *max-distribution*. Given $F(x)$, the max-distribution—and thus the left-hand side of Equation (1)—can be computed exactly as we show below in Section 3.1.1. However, since $F(x)$ cannot be known until an entire search has been completed, we must approximate the max-distribution. We use the standard approximation for $F(x)$ based on the current sampled performance data up to time t—the so-called empirical cdf (ecdf) for X. Section 3.1.2 presents our early stopping procedure based on these ideas, and discusses the issues that arise in practice.

3.1.1 Computing the max-distribution exactly and approximately

We can explicitly compute the max-distribution as follows. First, observe that

$$G_t(x) = Pr[M_t \leq x] \quad = \quad Pr[X_1 \leq x, X_2 \leq x, \ldots, X_t \leq x].$$

Recall that the search proceeds by choosing implementations uniformly at random without replacement. We can look at the calculation of the max-distribution

as a counting problem. At time t, there are $\binom{N}{t}$ ways to have selected t implementations. Of these, the number of ways to choose t implementations, all with performance at most x, is $\binom{n(x)}{t}$, provided $n(x) \geq t$. To cover the case when $n(x) < t$, let $\binom{a}{b} = 0$ when $a < b$ for notational ease. Thus,

$$G_t(x) \quad = \quad \frac{\binom{n(x)}{t}}{\binom{N}{t}} = \frac{\binom{N \cdot F(x)}{t}}{\binom{N}{t}} \tag{2}$$

where the latter equality follows from the definition of $F(x)$.

We cannot evaluate the max-distribution after $t < N$ samples because of its dependence on $F(x)$. However, we can use the t observed samples to approximate $F(x)$ using the empirical cdf (ecdf) $\hat{F}_t(x)$ based on the t samples:

$$\hat{F}_t(x) = \frac{\hat{n}_t(x)}{t} \tag{3}$$

where $\hat{n}_t(x)$ is the number of observed samples that are at most x at time t. We can now approximate $G_t(x)$ by the following $\hat{G}_t(x)$:

$$\hat{G}_t(x) \quad = \quad \frac{\binom{\lceil N \cdot \hat{F}_t(x) \rceil}{t}}{\binom{N}{t}} \tag{4}$$

The ceiling ensures that we evaluate the binomial coefficient in the numerator using an integer. Thus, our empirical stopping criterion, which approximates the "true" stopping criterion shown in Equation (1), is

$$\hat{G}_t(x) \quad \leq \quad \alpha \qquad . \tag{5}$$

3.1.2 Implementing an early stopping procedure

A search with our early stopping criterion proceeds as follows. First, a user or library designer specifies the search tolerance parameters ϵ and α. Then at each time t, the automated search system carries out the following steps:

1. Compute $\hat{F}_t(1-\epsilon)$, Equation (3), using *rescaled* samples as described below.
2. Compute $\hat{G}_t(1-\epsilon)$, Equation (4).
3. If the empirical stopping criterion, Equation (5), is satisified, then terminate the search.

Note that the ecdf $\hat{F}_t(x)$ models $F(x)$, making no assumptions about how performance varies with respect to the implementation tuning parameters. Thus, unlike gradient descent methods, this model can be used in situations where performance is an irregular function of tuning parameters, such as the example shown in Figure 2 (*bottom*).

There are two additional practical issues to address. First, due to inherent variance in the estimate $\hat{F}_t(x)$, it may be problematic to evaluate empirical stopping criterion, Equation (5), at every time t. Instead, we wait until t exceeds some minimum number of samples, $t_{\min}$, and then evaluate the stopping criterion at periodic intervals. For the experiments in this study, we use $t_{\min} = .02N$, and re-evaluate the stopping criterion at every $.01N$ samples, following a rule-of-thumb regarding ecdf approximation [8].

Second, we need a reasonable way to scale performance so that it lies between 0 and 1. Scaling by theoretical machine peak speed is not appropriate for all kernels, and a true upper bound on performance may be difficult to estimate. We choose to *rescale* the samples at each time t by the *current* maximum. That is, if $\{s_1, \ldots, s_t\}$ are the observed values of performance up to time t, and $m_t = \max_{1 \le k \le t} s_k$, then we construct the ecdf $\hat{F}_t(x)$ using the values $\{s_k/m_t\}$. This rescaling procedure tends to overestimate the fraction of samples near the maximum, meaning the stopping condition will be satisfied earlier than when it would have been satisfied had we known the true distribution $F(x)$. Furthermore, we would expect that by stopping earlier than the true condition indicates, we will tend to find implementations whose performance is less than $1-\epsilon$. Nevertheless, as we show in Section 3.2, in practice this rescaling procedure appears to be sufficient to characterize the shape of the distributions, meaning that for an appropriate range of α values, we still tend to find implementations with performance greater than $1 - \epsilon$.[3]

There are distributions for which we would not expect good results. For instance, consider a distribution in which 1 implementation has performance equal to 1, and the remaining $N - 1$ implementations have performance equal to $\frac{1}{2}$, where $N \gg 1$. After the first $t_{\min}$ samples, under our rescaling policy, all samples will be renormalized to 1 and the ecdf $\hat{F}_t(1 - \epsilon)$ will evaluate to zero for any $\epsilon > 0$. Thus, the stopping condition will be immediately satisfied, but the realized performance will be $\frac{1}{2}$. While this artificial example might seem unrepresentative of distributions arising in practice (as we verify in Section 3.2), it is important to note the potential pitfalls.

3.2 Results and discussion using PHiPAC data

We applied the above model to the register tile space data for the platforms shown in Figure 2 (*top*). Specifically, on each platform, we simulated 300 searches using a random permutation of the exhaustive search data collected for Figure 2 (*top*). For various values of ϵ and α, we measured (1) the average stopping time over all searches, and (2) the average proximity in performance of the implementation found to the best found by exhaustive search.

Figures 3–6 show the results for the Intel Itanium 2, Alpha 21164 (Cray T3E node), Sun Ultra 2i, and Intel Mobile Pentium III platforms, respectively. The

[3] We conjecture, based on some preliminary experimental evidence, that it may be possible to extend the known theoretical bounds on the quality of ecdf approximation due to Kolmogorov and Smirnov [12, 64] to the case where samples are rescaled in this way. Such an extension would provide theoretical grounds that this rescaling procedure is reasonable.

top half of Figures 3–6 show the average stopping time as a fraction of the search space size for various values of ϵ and α. That is, each plot shows at what value of t/N the empirical stopping criterion, Equation (5), was satisfied. Since our rescaling procedure will tend to overestimate the fraction of implementations near the maximum (as discussed in Section 3.1.2), we need to check that the performance of the implementation chosen is indeed close to (if not well within) the specified tolerance ϵ when α is "small," and moreover what constitutes a small α. Therefore, the bottom half of Figures 3–6 shows the average proximity to the best performance when the search stopped. More specifically, for each (ϵ, α) we show $1 - \bar{M}_t$, where $\bar{M}_t$ is the average observed maximum at the time t when Equation (5) was satisfied. (Note that $\bar{M}_t$ is the "true" performance where the maximum performance is taken to be 1.)

Suppose the user selects $\epsilon = .05$ and $\alpha = .1$, and then begins the search. These particular parameter values can be interpreted as the request, "stop the search when we find an implementation within 5% of the best with less than 10% uncertainty." Were this search conducted on the Itanium 2, for which many samples exhibit performance near the best within the space, we would observe that the search ends after sampling just under 10.2% of the full space on average (Figure 3 (*top*)), having found an implementation whose performance was within 2.55% of the best (Figure 3 (*bottom*)). Note that we requested an implementation within 5% ($\epsilon = .05$), and indeed the distribution of performance on the Itanium 2 is such that we could do even slightly better (2.55%) on average.

From the perspective of stopping a search as soon as possible, the Alpha 21164 T3E node presents a particularly challenging distribution. According to Figure 2 (*top*), the Alpha 21164 distribution has a relatively long tail, meaning very few implementations are fast. At $\epsilon = .05$ and $\alpha = .1$, Figure 4 (*top*) shows that indeed we must sample about 70% of the full space. Still, we do find an implementation within about 3% of the best on average. Indeed, for $\epsilon = .05$, we will find implementations within 5% of the best for all $\alpha \lesssim .15$.

On the Ultra 2i (Figure 5), the search ends after sampling about 14% of the space, having found an implementation between 3–3.5% of the best, again at $\epsilon = .05, \alpha = .1$. On the Pentium III (Figure 6), the search ends after just under 20%, having found an implementation within 5.25% of the best.

The differing stopping times across all four platforms show that the model does indeed adapt to the characteristics of the implementations and the underlying machine. Furthermore, the size of the space searched can be reduced considerably, without requiring any assumptions about how performance varies within the space. Moreover, these examples suggest that the approximation $\hat{F}_t(x)$ to the true distribution $F(x)$ is a reasonable one in practice, judging by the proximity of the performance of the implementation selected compared to $1 - \epsilon$ when $\alpha \lesssim .15$.

There are many other possible combinatorial search algorithms, including simulated annealing and the use of genetic algorithms, among others. We review the application of these techniques to related search-based systems in Section 5. In prior work, we have experimented with search methods including random, ordered, best-first, and simulated annealing [10]. The OCEANS project [49]

has also reported on a quantitative comparison of these methods and others applied to a search-based compilation system. In these two instances, random search was comparable to and easier to implement than competing techniques. Our stopping condition adds user-interpretable bounds (ϵ and α) to the random method, while preserving the simplicity of the random method's implementation.

In addition, the idea of user-interpretable bounds allows a search system to provide feedback to the user in other search contexts. For example, if the user wishes to specify a maximum search time (*e.g.*, "stop searching after 1 hour"), the estimate of the probability $Pr[M_t > 1 - \epsilon]$ could be computed for various values of ϵ at the end of the search and reported to the user. A user could stop and resume searches, using these estimates to gauge the likely difficulty of tuning on her particular architecture.

Finally, the stopping condition as we have presented complements existing pruning techniques: a random search with our stopping criterion can always be applied to any space after pruning by other heuristics or methods.

4 Statistical Classifiers for Run-time Selection

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 implementation selection problem [73, 14]: how can we automatically build decision rules to select the best implementation for a given input? Below, we treat this problem as a statistical classification task. We show how the problem might be tackled from this perspective by applying three types of statistical models to a matrix multiply example. In this example, given the dimensions of the input matrices, we must choose at run-time one implementation from among three, where each of the three implementations has been tuned for matrices that fit in different levels of cache.

4.1 A formal framework

We can pose the 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 n 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, and
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.*, $f : S \rightarrow A$. The idea is to construct $f(s)$ using the

performance of the implementations in A on a sample of the inputs S_0. We refer to S_0 as the *training set*, and we refer to the execution time data $T(a, s)$ for $a \in A, s \in S_0$ as the *training data*. In geometric terms, we would like to partition the input space by implementation, as shown in Figure 7 (*left*). This partitioning 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 an implementation.

The decision function f models the relative performance of the implementations in A. Here, we consider three types of statistical models that trade-off classification accuracy against the cost of building f and the cost of executing f at run-time. Roughly speaking, we can summarize these models as follows:

1. *Parametric data modeling*: We can build a parametric statistical model of the execution time data directly. For each implementation, we posit a parameterized model of execution time and use the training data to estimate the parameters of this model (*e.g.*, by linear regression for a linear model). At run-time, we simply evaluate the models to predict the execution time of each implementation. This method has been explored in prior work on run-time selection by Brewer [14]. Because we choose the model of execution time, we can control the cost of evaluating f by varying the complexity of the model (*i.e.*, the number of model parameters).

2. *Parametric geometric modeling*: Rather than model the execution time directly, we can also model the shape of the partitions in the input space parametrically, by, say, assuming that the *boundaries* between partitions can be described concisely by parameterized functions. For example, if the input space is two-dimensional, we might posit that each boundary is a straight line which can of course be described concisely by specifying its slope and intercept. Our task is to estimate the parameters (*e.g.*, slope and intercept) of all boundaries using the training data. Such a model might be appropriate if a sufficiently accurate model of execution time is not known but the boundaries can be modeled. Like parametric data modeling methods, we can control the cost of evaluating f by our choice of functions that represent the boundaries.

3. *Nonparametric geometric modeling*: Rather than assume that the partition boundaries have a particular shape, we can also construct implicit models of the boundaries in terms of the actual data points. In statistical terms, this type of representation of the boundaries is called nonparametric. In the example of this paper, we use the *support vector method* to construct just such a nonparametric model [88]. The advantage of the nonparametric approach is that we do not have to make any explicit assumptions about the input distributions, running times, or geometry of the partitions. However, we will need to store at least some subset of the data points which make up the implicit boundary representation. Thus, the reduction in assumptions comes at the price of more expensive evaluation and storage of f compared to a parametric method.

(Note that this categorization of models implies a fourth method: nonparametric data modeling. Such models are certainly possible, for example, by the use of support vector regression to construct a nonparametric model of the data [79]. We do not consider these models in this paper.)

To illustrate the classification framework, we apply the above three models to a matrix multiply example. Specifically, consider the operation $C \leftarrow C + AB$, 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 7 (*right*). Note that these three parameters make the input space S three-dimensional. In PHiPAC, it is possible to generate different implementations tuned on different matrix workloads [11]. Essentially, this involves conducting a search where the size of the matrices on which the implementations are benchmarked is specified so that the matrices fit within a particular cache level. For instance, we could have three implementations, one tuned for the matrix sizes that fit approximately within L1 cache, those that fit within L2, and all larger sizes.

We compare the accuracy of the above modeling methods using two metrics. First, we use the average misclassification rate, *i.e.*, the fraction of test samples mispredicted. Note that 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, our second comparison metric is the slow-down of the predicted implementation relative to the true best. That is, for each point in the test set, we compute the relative slow-down $\frac{t_{\text{selected}}}{t_{\text{best}}} - 1$, where t_{selected} and t_{best} are the execution times of the predicted and best algorithms for a given input, respectively. For a given modeling technique, we consider the distribution of slow-downs for points in the test set.

4.2 Parametric data model: linear regression modeling

In our first approach, we 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 adopted by Brewer [14]. For matrix multiply on matrices of size $N \times N$, we might guess that the running time of implementation a will have the form

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

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

One strength 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:

$$T_a(N) = \beta_7 MKN + \beta_6 MK + \beta_5 KN + \beta_4 MN + \beta_3 M + \beta_2 K + \beta_1 N + \beta_0. \quad (6)$$

We can even eliminate terms whose coefficients are "small" to reduce the runtime cost of generating a prediction. For matrix multiply, a simple model of this form could even be automatically derived by an analysis of the 3-nested loops structure. However, in general it might be difficult to determine a sufficiently precise parametric form that captures the interaction effects between the processor and all levels of the memory hierarchy. Moreover, for other more complicated kernels or algorithms—having, say, more complicated control flow like recursion or conditional branches—such a model may be more difficult to derive.

4.3 Parametric geometric model: separating hyperplanes

One geometric approach is to first assume that there are some number of boundaries, each described parametrically, that divide the implementations, and then find best-fit boundaries with respect to an appropriate cost function.

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. Furthermore, let the weights satisfy the property, $\sum_{a \in A} w_{\theta_a}(s) = 1$. 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 parameters $\theta_{a_1}, \ldots, \theta_{a_m}$ (and thus, the weights) so as to minimize the the following weighted execution time over the training set:

$$C(\theta_{a_1}, \ldots, \theta_{a_m}) = \frac{1}{|S_0|} \sum_{s \in S_0} \sum_{a \in A} w_{\theta_a}(s) \cdot T(a, s). \tag{7}$$

Intuitively, if we view $w_{\theta_a}(s)$ as a probability of selecting algorithm a on input s, then C is a measure of the expected execution time if we first choose an input uniformly at random from S_0, and then choose an implementation with the probabilities given by the weights on input s.

In this formulation, inputs s with large execution times $T(a, s)$ will tend to dominate the optimization. Thus, if all inputs are considered to be equally important, it may be desirable to use some form of normalized execution time. We defer a more detailed discussion of this issue to Section 4.5.

Of the many possible choices for $w_{\theta_a}(\cdot)$, we choose the *logistic* function,

$$w_{\theta_a}(s) = \frac{\exp\left(\theta_a^T s + \theta_{a,0}\right)}{\sum_{b \in A} \exp\left(\theta_b^T s + \theta_{b,0}\right)} \tag{8}$$

where θ_a has the same dimensions as s, $\theta_{a,0}$ is an additional parameter to estimate. Note that the denominator ensures that $\sum_{a \in A} w_{\theta_a}(s) = 1$. While there is some statistical motivation for choosing the logistic function [45], in this case it also turns out that the derivatives of the weights are particularly easy to compute. Thus, we can estimate θ_a and $\theta_{a,0}$ by minimizing Equation (7) numerically using Newton's method.

A nice property of the weight function is that f is cheap to evaluate at runtime: 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.

4.4 Nonparametric geometric model: support vectors

Techniques exist to model the partition boundaries nonparametrically. The support vector (SV) method is one way to construct just such a nonparametric model, given a labeled sample of points in the space [88].

Specifically, each training sample $s_i \in S_0$ is given a label $l_i \in A$ to indicate which implementation was fastest on input s_i. That is, the training points are assigned to classes by implementation. The SV method then computes a partitioning by selecting a subset of training points that best represents the location of the boundaries, where by "best" we mean that the minimum geometric distance between classes is maximized.[4] The resulting decision function $f(s)$ is essentially a linear combination of terms with the factor $K(s_i, s)$, where only s_i in the selected subset are used, and K is some symmetric positive definite function. Ideally, K is chosen to suit the data, but there are also a variety of "standard" choices for K as well. We refer the reader to the description by Vapnik for more details on the theory and implementation of the method [88].

The SV method is regarded as a state-of-the-art method for the task of statistical classification on many kinds of data, 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 up to a factor of $|S_0|$ greater than that of the other methods since some fraction of the training points must be retained to evaluate f. 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 and discussion 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 with a 16 KB L1 cache and a 512 KB L2 cache.

4.5.1 Experimental setup

To evaluate the prediction accuracy of the three run-time selection algorithms, we conducted the following experiment. First, we built three matrix multiply implementations using PHiPAC: (a) one with only register-level tiling, (b) one with register + L1 tiling, and (c) one with register, L1, and L2 tiling. We considered the performance of these implementations within 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

[4] Formally, this is known as the *optimal margin* criterion [88].

disjoint subsets of points in this space, where each subset contained 1936 points chosen at random.[5] Then we further divided each subset into 500 testing points and 1436 training points. We trained and tested the three statistical models (details below), measuring the prediction accuracy on each test set.

In Figure 8, we show an example of a 500-point testing set from this space where each point is color-coded by the implementation which ran fastest. The implementation which was fastest on the majority of inputs is the default implementation generated by PHiPAC containing full filing optimizations, and is shown by a blue "x". Thus, a useful reference is a *baseline predictor* which always chooses this implementation: the misclassification rate of this predictor was 24%. The implementation using only register-tiling makes up the central "banana-shaped" region in the center of Figure 8, shown by a red "o". The register and L1 tiled implementation, shown by a green asterisk (*), was fastest on a minority of points in the lower left-hand corner of the space. Observe that the space has complicated boundaries, and is not strictly cleanly separable.

The three statistical models were implemented as follows.

- We implemented the linear least squares regression method as described in Section 4.2, Equation (6). Since the least squares fit is based on choosing the fit parameters to minimize the total square error between the execution time data and the model predictions, errors in the larger problem sizes will contribute more significantly to the total squared error than smaller sizes, and therefore tend to dominate the fit. This could be adjusted by using weighted least squares methods, or by normalizing execution time differently. We do not pursue these variations here.

- For the separating hyperplane method outlined in Section 4.3, we built a model using 6 hyperplanes in order to try to better capture the central region in which the register-only implementation was fastest. Furthermore, we replaced the execution time $T(a, s)$ in Equation (7) by a "binary" execution time $\hat{T}(a, s)$ such that $\hat{T}(a, s) = 0$ if a was the fastest on input s, and otherwise $\hat{T}(a, s) = 1$. (We also compared this binary scheme to a variety of other notions of execution time, including normalizing each $T(a, s)$ by MKN to put all execution time data on a similar scale. However, we found the binary notion of time gave the best results in terms of the average misclassification rate on this particular data set.)

- For the support vector method of Section 4.4, we used Platt's *sequential minimal optimization* algorithm with a Gaussian kernel for the function $K(\cdot, \cdot)$ [69]. In Platt's algorithm, we set the tuning parameter $C = 100$ [69]. We built multiclass classifiers from ensembles of binary classifiers, as described by Vapnik [88].

Below, we report on the overall misclassification rate for each model as the average over all of the 10 test sets.

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

4.5.2 Results and discussion

Qualitative examples of the predictions made by the three models on a sample test set are shown in Figures 9–11. The regression method captures the boundaries roughly but does not correctly model one of the implementations (upper-left of Figure 9). The separating hyperplane method is a poor qualitative fit to the data. The SV method appears to produce the best predictions. Quantatively, the misclassification rates, averaged over the 10 test sets, were 34% for the regression predictor, 31% for the separating hyperplanes predictor, 12% for the support vector predictor. Only the support vector predictor significantly outperformed the baseline predictor.

However, misclassification rate seems too strict a measure of prediction performance, since we may be willing to tolerate some penalties to obtain a fast prediction. Therefore, we also show the distribution of slow-downs due to mispredictions in Figure 12. Each curve depicts this distribution for one of the four predictors. The distribution shown is for one of the 10 trials which yielded the lowest misclassification rate. Slow-down appears on the x-axis, and the fraction of predictions on all 1936 points (including both testing and training points) exceeding a given slow-down is shown on the y-axis.

Consider the baseline predictor (solid blue line with '+' markers). Note that only 5–6% of predictions led to slow-downs of more than 5%, and that only about 0.4% of predictions led to slow-downs of more than 10%. Noting the discretization, evidently only 1 out of the 1936 cases led to a slow-down of more than 47%, with no implementations being between 18–47% slower. These data indicate that the baseline predictor performs fairly well, and that furthermore the performance of the three tuned implementations is fairly similar. Therefore, we do not expect to improve upon the baseline predictor by much. This hypothesis is borne out by observing the slow-down distributions of the separating hyperplane and regression predictors (green circles and red 'x' markers, respectively), neither of which improves significantly (if at all) over the baseline.

However, we also see that for slow-downs of up to 5% (and, to a lesser extent, up to 10%), the support vector predictor (cyan '*' markers) shows a significant improvement over the baseline predictor. It is possible that this difference would be significant in some applications with very strict performance requirements, thereby justifying the use of the more complex statistical model. Furthermore, had the differences in execution time between implementations been larger, the support vector predictor would have appeared even more attractive.

Note that there are a number of cross-over points in Figure 12. For instance, comparing the regression and separating hyperplanes methods, we see that even though the overall misclassification rate for the separating hyperplanes 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. These cross-overs suggest the possibility of hybrid schemes that combine predictors or take different actions on inputs in the "tails" of these distributions, provided these inputs could somehow be identified or otherwise isolated.

In terms of prediction times (*i.e.*, the time to evaluate $f(s)$), both the regression and separating hyperplane 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. Again, it may be possible to reduce this run-time overhead by a simple conditional test of the input dimensions, or perhaps a hybrid predictor.

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.[6] 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 only a particular 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; the examples presented here offer a flavor of the role that statistical modeling of performance data can play.

5 Related Work: A Survey of Empirical Search-Based Tuning

There has been a flurry of research activity in the use of empirical search-based approaches to platform-specific code generation and tuning. The primary motivation, as this paper demonstrates for matrix multiply, is the difficulty of instantiating purely static models that predict performance with sufficient accuracy to decide among possible code and data structure transformations. Augmenting such models with observed performance appears to yield viable and promising ways to make these decisions.

In our review of the diverse body of related work, we note how each study or project addresses the following high-level questions:

1. **What is the unit of optimization?** In a recent position paper on *feedback-directed optimization*, Smith argues that a useful way to classify dynamic optimization methods is by the size and semantics of the piece of the program being optimized [78]. Traditional static compilation applies optimizations in "units" which following programming language conventions, *e.g.*, within a basic block, within a loop nest, within a procedure, or within a module. By contrast, dynamic (run-time) techniques optimize across units relevant to run-time behavior, *e.g.*, along a sequence of consecutively executed basic blocks (a *trace* or *path*).

   Following this classification, we divide the related work on empirical search-based tuning primarily into two high-level categories: *kernel-centric* tuning and *compiler-centric* tuning. This paper adopts the kernel-centric

[6] In particular, the hyperplane and regression methods were written in Matlab, while the SMO support vector training code was written in C.

perspective in which the unit of optimization is the kernel itself. The code generator—and hence, the implementation space—is specific to the kernel. One would expect that a generator specialized to a particular kernel might best exploit mathematical structure or other structure in the data (possibly known only at run-time) relevant to performance. As we discuss below, this approach has been very successful in the domains of linear algebra and signal processing, where understanding problem-specific structure leads to new, tunable algorithms and data structures.

In the compiler-centric view, the implementation space is defined by the space of possible compiler transformations that can be applied to any program expressed in a general-purpose programming language. In fact, the usual suite of optimizations for matrix multiply can all be expressed as compiler transformations on the standard 3-nested loop implementation, and thus it is possible in principle for a compiler to generate the same high-performance implementation that can be generated by hand. However, what makes a specialized generator useful in this instance is that the expert who writes the generator identifies the precise transformations which are hypothesized to be most relevant to improving performance. Moreover, we could not reasonably expect a general purpose compiler to know about all of the possible mathematical transformations or alternative algorithms and data structures for a given kernel—it is precisely these kinds of transformations that have yielded the highest performance for other important computational kernels like the discrete Fourier transform (DFT) or operations on sparse matrices.

We view these approaches as complementary, since hybrid approaches are also possible. For instance, in this paper we consider the use of a matrix multiply-specific generator that ouputs C or Fortran code, thus leaving aspects of the code generation task (namely, scheduling) to the compiler. What these approaches share is that their respective implementation spaces can be very large and difficult to model. It is the challenge of *choosing* an implementation that motivates empirical search-based tuning.

2. **How should the implementation space be searched?** Empirical search-based approaches typically choose implementations by some combination of *modeling* and *experimentation* (*i.e.*, actually running the code) to predict performance and thereby choose implementations. Section 2 argues that performance can be a complex function of algorithmic parameters, and therefore may be difficult to model accurately using only static models in practice. This paper explores the use of statistical models, constructed from empirical data, to model performance within the space of implementations. The related work demonstrates that a variety of additional kinds of models are possible. For instance, one idea that has been explored in several projects is the use of evolutionary (genetic) algorithms to model and search the space of implementations.

3. **When to search?** The process of searching an implementation space

could happen at any time, whether it be strictly off-line (*e.g.*, once per architecture or once per application), strictly at run-time, or in some combination. The cost of an off-line search can presumably be amortized over many uses, while a run-time search can maximally use the information only available at run-time. Again, hybrid approaches are common in practice.

The question of when to search has implications for software system support. For instance, a strictly off-line approach requires only that a user make calls to a special library or a special search-based compiler. Searching at run-time could also be hidden in a library call, but might also require changes to the run-time system to support dynamic code generation or dynamic instrumentation or trap-handling to support certain types of profiling. This survey mentions a number of examples.

Our survey summarizes how various projects and studies have approached these questions, with a primary emphasis on the kernel-centric vs. compiler-centric approaches, though again these we see these two viewpoints as complementary. Collectively, these questions imply a variety of possible software architectures for generating code adapted to a particular hardware platform and run-time workload.

5.1 Kernel-centric empirical search-based tuning

Typical kernel-centric tuning systems contain specialized code generators that exploit specific mathematical properties of the kernel or properties of the data. The target performance goal of these systems is to achieve the performance of hand-tuned code. Most research has focused on tuning in the domains of dense and sparse linear algebra, and signal processing. In these areas, there is a rich mathematical structure relevant to performance to exploit. We review recent developments in these and other areas below. (For alternative views of some of this work, we refer the reader to recent position papers on the notion of *active libraries* [90] and self-adapting numerical software [28].)

5.1.1 Dense and sparse linear algebra

Dense matrix multiply is among the most important of the computational kernels in dense linear algebra both because a large fraction (say, 75% or more) of peak speed can be achieved on most machines with proper tuning, and also because many other dense kernels can be expressed as calls to matrix multiply [47]. The prototype PHiPAC system was an early system for generating automatically tuned implementations of this kernel with cache tiling, register tiling, and a variety of unrolling and software pipelining options [10]. The notion of automatically generating tiled matrix multiply implementations from a concise specification with the possibility of searching the space of tile sizes for matrix multiply also appeared in early work by McCalpin and Smotherman [58]. The ATLAS project has since extended the applicability of the PHiPAC prototype to all of the other dense matrix kernels that constitute the BLAS [93]. These

systems contain specalized, kernel-specific code generators, as discussed in Section 2. Furthermore, most of the search process can be performed completely off-line, once per machine architecture. The final output of these systems is a library implementing the BLAS against which a user can link her application.

A recent and promising avenue of research relates to the construction of sophisticated new generators. Veldhuizen [89] and Siek and Lumsdaine [77] have developed C++ language-based techniques for cleanly expressing dense linear algebra kernels. More recent work by Gunnels, *et al.*, in the FLAME project demonstrates the feasibility of systematic derivation of algorithmic variations for a variety of dense matrix kernels [40]. These variants would be suitable implementations a_i in our run-time selection framework (Section 4). In addition, FLAME provides a new methodology by which one can cleanly generate implementations of kernels that exploit caches. However, these implementations still rely on highly-tuned "inner" matrix multiply code, which in turn requires register- and instruction-level tuning. Therefore, we view all of these approaches to code generation as complementing empirical search-based register- and instruction-level tuning.

Another complementary research area is the study of so-called *cache-oblivious* algorithms, which to claim eliminate the need for cache-level tuning to some extent for a number of computational kernels. Like traditional tiling techniques [41, 75], cache oblivious algorithms for matrix multiply and LU factorization have been shown to asymptotically minimize data movement among various levels of the memory hierarchy, under certain cache modeling assumptions [83, 33, 1, 30]. Unlike tiling, cache-oblivious algorithms do not make explicit reference to a "tile size" tuning parameter, and thus appear to eliminate the need to search for optimal cache tile sizes either by modeling or by empirical search. Furthermore, language-level support now exists both to convert loop-nests to recursion automatically [96] and also to convert linear array data structures and indexing to recursive formats [94]. However, we note in Section 2 that at least for matrix multiply, cache-level optimizations account for only a part (perhaps 12–60%, depending on the platform) of the total performance improvement possible, and therefore complements additional register- and instruction-level tuning. The nature of performance in these spaces, as shown in Figure 2, together with recent results showing that even carefully constructed models of the register- and instruction-level implementation space can mispredict [97], imply that empirical search is still necessary for tuning.[7]

For matrix multiply, algorithmic variations that require fewer than $O(n^3)$ flops for $n \times n$ matrices, such as Strassen's algorithm, are certainly beyond the kind of transformations we expect general purpose compilers to be able to derive. Furthermore, like cache-oblivious algorithms, practical and highly efficient implementations of Strassen's algorithm still depend on highly-tuned base-case implementations in which register- and instruction-level tuning is critical [43, 82].

[7] Indeed, recent work has qualitatively confirmed the need and importance of fast "base case" implementations in recursive implementations [32, 34, 65].

The BLAS-tuning ideas have been applied to higher-level, parallel dense linear algebra libraries. In the context of cluster computing in the Grid, Chen, *et al.*, have designed a self-tuning version of the LAPACK library for Clusters (LFC) [19]. LFC preserves LAPACK's serial library interface, and decides at run-time whether and how to parallelize a call to a dense linear solve routine, based on the current cluster load. In a similar spirit, Liniker, *et al.*, have applied the idea of run-time selection to the selection of *data layout* in their distributed parallel version of the BLAS library [55, 7]. Their library, called DESOBLAS, is based on the idea of delayed evaluation: all calls to DESOBLAS library routines return immediately, and are not executed until either a result is explicitly accessed by the user or the user forces an evaluation of all unexecuted calls. At evaluation time, DESOBLAS uses information about the entire sequence of operations that need to be performed to make decisions about how to distribute data. Both LFC and DESOBLAS adopt the library interface approach, but defer optimization until run-time.

Kernels arising in sparse linear algebra, such as sparse matrix-vector multiply, complicate tuning compared to their dense counterparts because performance depends on the non-zero structure of the sparse matrix. For sparse kernels, the user must choose a data structure that minimizes storage of the matrix while still allowing efficient mapping of the kernel to the target architecture. Worse still, the matrix structure may not be known until run-time. Prototype systems exist which allow a user to specify separately both the kernel and the data structure, while a specialized generator (or restructuring compiler) combines the two specifications to generate an actual implementation [9, 70, 81]. At present, such systems do not explicitly address the register- and instruction-level tuning issues, nor do they adequately address the run-time problem of choosing a data structure given a sparse matrix. Automatic tuning with respect to these low-level tuning and data structure selection issues have been taken up by recent work on the SPARSITY system [44, 92].

5.1.2 Digital signal processing

Recent interest in automatic tuning of digital signal processing (DSP) applications is driven both by the rich mathematical structure of DSP kernels and by the variety of target hardware platforms. One of the best-studied kernels, the discrete Fourier transform (DFT), admits derivation of many fast Fourier transform (FFT) algorithms. The fast algorithms require significantly fewer flops than a naïve DFT implementation, but since different algorithms have different memory access patterns, strictly minimizing flops does not necessarily minimize execution time. The problem of tuning is further complicated by the fact that the target architectures for DSP kernels range widely from general purpose microprocessors and vector architectures to special-purpose DSP chips.

FFTW was the first tuning system for various flavors of the discrete Fourier transform (DFT) [32]. 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. Search boils down to selecting the best

fully-unrolled base case implementations, or equivalently, the base cases with the best instruction scheduling. The search process occurs only at run-time because that is when the problem size is assumed to be known. There have since been additional efforts in signal processing which build on the FFTW ideas. The SPIRAL system is built on top of a symbolic algebra system, allows users to enter customized transforms in an interpreted environment using a high-level tensor notation, and uses a novel search method based on genetic algorithms [71]. The performance of the implementations generated by these systems is largely comparable both to one another and to vendor-supplied routines. One distinction between the two systems is that SPIRAL's search is off-line, and carried out for a specific kernel of a given size, whereas FFTW chooses the algorithm at run-time. The most recent FFT tuning system has been the UHFFT system, which is essentially an alternative implementation of FFTW that includes a different implementation of the code generator [60]. In all three systems, the output of the code generator is either C or Fortran code, and the user interface to a tuned routine is via a library or subroutine call.

5.1.3 Other kernel domains

In the area of parallel distributed communications, Vadhiyar, *et al.*, propose techniques to tune automatically the Message Passing Interface (MPI) collective operations [86]. The most efficient implementations of these kernels, which include "broadcast," "scatter/gather," and "reduce," depend on characteristics of the network hardware. Like its tuning system predecessors in dense linear algebra, this prototype for MPI kernels targets the implementation of a standard library interface. Achieved performance meets or exceeds that of vendor-supplied implementations on several platforms. The search for an optimal implementation is conducted entirely off-line, using heuristics to prune the space and a benchmarking workload that stresses message size and number of participating processors, among other features.

Empirical search-based tuning systems for sorting have shown some promise. Recent work by Arge, *et al.*, demonstrate that algorithms which minimize cache misses under simple but reasonable cache models lead to sorting implementations which are suboptimal in practice [2]. They furthermore stress the importance of register- and instruction-level tuning, and use all of these ideas to propose a new sorting algorithm space with machine-dependent tuning parameters. A preliminary study by Darcy shows that even for the well-studied quicksort algorithm, an extensive implementation space exists and exhibits distributions of performance like those shown in Figure 2 (*top*) [24]. Lagoudakis and Littman have shown how the selection problem for sorting can be tackled using statistical methods not considered in this paper, namely, by reinforcement learning techniques [52]. Together, these studies suggest the applicability of search-based methods to non-numerical computational kernels.

Recently, Baumgartner, *et al.*, have proposed a system to generate entire parallel applications for a class of quantum chemistry computations [6]. Like SPIRAL, this system provides a way for chemists to specify their computation in

a high-level notation, and carries out a symbolic search to determine a memory and flop efficient implementation. The authors note that the best implementation depends ultimately on machine-specific parameters. Some heuristics tied to machine parameters (*e.g.*, available memory) guide search.

Dolan and Moré have identified empirical distributions of performance as a mechanism for comparing various mathematical optimization solvers [26]. Specifically, the distributions estimate the probability that the performance of a given solver will be within a given factor of the best performance of all solvers considered. Their data was collected using the online optimization server, NEOS, in which users submit optimization jobs to be executed on NEOS-hosted computational servers. The primary aim of their study was to propose a new "metric" (namely, the distributions themselves) as a way of comparing different optimization solvers. However, what these distributions also show is that Grid-like computing environments can be used to generate a considerable amount of performance data, possibly to be exploited in run-time selection contexts as described in Section 4.

A key problem in the run-time selection framework we present in Section 4 is the classical statistical learning problem of *feature selection*. In our case, features are the attributes that define the input space. The matrix multiply example assumes the input matrix dimensions constitute the best features. Can features be identified automatically in a general setting? A number of recent projects have proposed methods, in the context of performance analysis and algorithm selection, which we view as possible solutions. Santiago, *et al.*, apply the statistical experimental design methods to program tuning [74]. These methods essentially provide a systematic way to analyze how much hypothesized factors contribute to performance. The most significant contributors identified could constitute suitable features for classification. A different approach has been to codify expert knowledge in the form of a database, recommender, or expert system in particular domains, such as a partial differential equation (PDE) solver [56, 72, 42], or a molecular dynamics simulation [51]. In both cases, each algorithmic variation is categorized by manually identified features which would be suitable for statistical modeling.

Note that what is common to most of the preceeding projects is a library-based approach, whether tuning occurs off-line or at run-time. The Active Harmony project seeks to provide a general API and run-time system that supports run-time selection and run-time parameter tuning in the setting of the Grid [85]. This work, though in its early stages, highlights the need for search in new computational environments.

5.2 Compiler-centric empirical search-based tuning

The idea of using data gathered during program execution to aid compilation has previously appeared in the compiler literature under the broad term *feedback-directed optimization* (FDO). A recent survey and position paper by Smith reviewed developments in subareas of FDO including profile-guided compilation (Section 5.2.2) and dynamic optimization (Section 5.2.4) [78]. FDO methods

are applied to a variety of program representations: source code in a general-purpose high-level language (*e.g.*, C or Java), compiler intermediate form, or even a binary executable. These representations enable transformations to improve performance on general applications, either off-line or at run-time. Binary representations enable optimizations on applications that have shipped or on applications that are delivered as mobile code. The underlying philosophy of FDO is the notion that optimization without reference to actual program behavior is insufficient to generate optimal or near-optimal code.

In our view, the developments in FDO join renewed efforts in superoptimizers (Section 5.2.1) and the new notion of self-tuning compilers (Section 5.2.3) in an important trend in compilation systems toward the use of empirically-derived models of the underlying machines and programs.

5.2.1 Superoptimizers

Massalin coined the term *superoptimizer* for his exhaustive search-based instruction generator [57]. Given a short program, represented as a sequence of (six or so) machine language instructions, the superoptimizer exhaustively searched all possible equivalent instruction sequences for a shorter (and equivalently at the time, faster) program. Though extremely expensive compared to the usual cost of compilation, the intent of the system was to "superoptimize" particular bottlenecks off-line. The overall approach represents a noble effort to generate truly "optimal" code.[8]

Joshi, *et al.*, substitute exhaustive search in Massalin's superoptimizer with an automated theorem prover in their Denali superoptimizer [46]. One can think of the prover as acting as a modeler of program performance. Given a sequence of expressions in a C-like notation, Denali uses the automated prover to generate a machine instruction sequence that is provably the fastest implementation possible. However, to make such a proof-based code generation system practical, Denali's authors necessarily had to assume (a) a certain model of the machine (*e.g.*, multiple issue with pipeline dependencies specified but fixed instruction latencies), and (b) a particular class of acceptable constructive proofs (*i.e.*, matching proofs). Nevertheless, Denali is able to generate extremely good code for short instruction sequences (roughly 16 instructions in a day's worth of time) representing ALU-bound operations on the Alpha EV6. As the Denali authors note, it might be possible to apply their approach more broadly by refining the instruction latency estimates, particularly for memory operations, with measured data from actual runs—again suggesting a combined modeling and empirical search approach.

5.2.2 Profile-guided compilation and iterative compilation

The idea behind *profile-guided compilation* (PGC) is to carry out compiler transformations using information gathered during actual execution runs [50, 38]. Compilers can instrument code to gather execution frequency statistics at the

[8] A refinement of the original superoptimizer, based on `gcc`, is also available [39].

level of subroutines, basic blocks, or paths. On subsequent compiles, these statistics can be used to enable more aggressive use of "classical" compiler optimizations (*e.g.*, constant propagation, copy propagation, common subexpression elimination, dead code removal, loop invariant code removal, loop induction variable elimination, global variable migration) along frequent execution paths [17, 4]. The PGC approach has been extended to help guide prefetch instruction placement on x86 architectures [5]. PGC can be viewed as a form of empirical search in which the implementation space is implicitly defined to be the space of all possible compiler transformations over all inputs, and the user (programmer) directs the search by repeatedly compiling and executing the program.

The search process of PGC can be automated by replacing the user-driven compile/execute sequence with a compiler-driven one. The term *iterative compilation* has been coined to refer to such a compiler process [49, 87]. Users annotate their program source with a list of which transformations—*e.g.*, loop unrolling, tiling, software pipelining—should be tried on a particular segment of code, along with any relevant parametric ranges (*e.g.*, a range of loop unrolling depths). The compiler then benchmarks the code fragment under the specified transformations. In a similar vein, Pike and Hilfinger built tile-size search using simulated annealing into the Titanium compiler, with application to a multigrid solver [68]. The Genetic Algorithm Parallelisation System (GAPS) by Nisbet addressed the problem of compile-time selection of an optimal sequence of serial and parallel loop transformations for scientific applications [63]. GAPS uses a genetic algorithms approach to direct search over the space of possible transformations, with the initial population seeded by a transformation chosen by "conventional" compiler techniques. The costs in all of these examples are significantly longer compile cycles (*i.e.*, including the costs of running the executable and re-optimizing), but the approach is "off-line" since the costs are incurred before the application ships. Furthermore, the compile-time costs can be reduced by restricting the iterative compilation process to only known application bottlenecks. In short, what all of these iterative compilation examples demonstrate is the utility of a search-based approach for tuning general codes that requires minimal user intervention.

5.2.3 Self-tuning compilers

We use the term *self-tuning compiler* to refer to recent work in which the compiler itself—*e.g.*, the compiler's internal models for selecting transformations, or the optimization phase ordering—is adapted to the machine architecture. The goal of this class of methods is to avoid significantly increasing compile-times (as occurs in iterative compilation) while still adapting the generated code to the underlying architecture.

Mitchell, *et al.*, proposed a scheme in which models of various types of memory access patterns are measured for a given machine when the compiler is installed [61]. At analysis time, memory references within loop nests are decomposed and modeled by functions of these canonical patterns. An execution time model is then automatically derived. Instantiating and comparing these

models allows the compiler to compare different transformations of the loop nest. Though the predicted execution times are not always accurate in an absolute sense, the early experimental evidence suggests that they may be sufficiently accurate to predict the relative ranking of candidate loop transformations.

The Meta Optimization project proposes automatic tuning of the compiler's internal *priority* (or *cost*) functions [80]. The compiler uses these functions to choose a code generation action based on known characteristics of the program. For example, in deciding whether or not to prefetch a particular memory reference within a loop, the compiler evaluates a binary priority function that considers the current loop trip count estimates, cache parameters, and estimated prefetch latency,[9] among other factors. The precise function is usually tuned by the compiler writer. In the Meta Optimization scheme, the compiler implementer specifies these factors, their ranges, and a hypothesized form of the function, and Meta Optimization uses a genetic programming approach to determine (*i.e.*, to evolve) a better form for the function. The candidate functions are evaluated on a benchmark or suite of benchmark programs to choose one. Thus, priority functions can be tuned once for all applications, or for a particular application or class of applications.

In addition to internal models, another aspect of the compiler subject to heuristics and tuning is the optimization phase ordering, *i.e.*, the order in which optimizations are applied. While this ordering is usually fixed through experimentation by a compiler writer, Cooper, *et al.*, have proposed the notion of an adaptive compiler which experimentally determines the ordering for a given machine [23, 22]. Their compiler uses genetic algorithms to search the space of possible transformation orders. Each transformation order is evaluated against some metric (*e.g.*, execution time or code size) on a pre-defined set of benchmark programs.

The Liberty compiler research group has proposed an automated scheme to organize the space of optimization configurations into a small decision tree that can be quickly traversed at compile-time [84]. Roughly speaking, their study starts with the Intel IA-64 compiler and identifies the equivalent of k internal binary flags that control optimization. This defines a space of possible configurations of size 2^k. This space is systematically pruned, and a final, significantly smaller set of configurations are selected.[10] (In a traditional compiler implementation, a compiler writer would manually choose just 1 such configuration based on intuition and experimentation.) The final configurations are organized into a decision tree. At compile-time, this tree is traversed and each configuration visited is applied to the code. The effect of the configuration is predicted by a static model, and used to decide which paths to traverse and what final configuration to select. This work combines the model-tuning of the other self-

[9] The minimum time between the prefetch and its corresponding load.

[10] In the original work's experiment, not all flags considered are binary. Nevertheless, the size of the original space is equivalent to the case when $k = 19$. The final number of configurations selected is 12. Also note that the paper proposes a technique for pruning the space which may be a variant of a common statistical method known as fractional factorial design (FFD). FFD has been applied to the automatic selection of compiler flags [20].

tuning compiler projects and the idea of iterative compilation (except that in this instance, performance is predicted by a static model instead of by running the code.)

5.2.4 Dynamic (run-time) optimization

Dynamic optimization refers to the idea of applying compiler optimizations and code generation at run-time. Just-in-time (JIT) compilation, particularly for Java-based programs, is one well-known example. Among the central problems in dynamic optimization are automatically deciding what part of an application to optimize, and how to reduce the run-time cost of optimization. Here, we concentrate on summarizing the work in which empirical search-based modeling is particularly relevant. We refer the reader to Smith's survey [78] and related work on dynamic compilation software architectures [54, 15] for additional references on specific run-time code generation techniques.

Given a target fragment of code at run-time, the Jalapeño JIT compiler for Java decides what level of optimization to apply based on an empirically derived cost-benefit model [3]. This model weighs the expected pay-off from a given optimization level, given an estimate of the frequency of future execution, against the expected cost of optimizing. Profiling helps to identify the program hotspots and cost estimates, and evaluation of the cost-benefit model is a form of empirical-model based search.

Two recent projects have proposed allowing the compiler to generate multiple versions of a code fragment (*e.g.*, loop body, procedure), enabling run-time search and selection for general programs [25, 91]. Diniz and Rinard coined the term *dynamic feedback* for the technique used in their parallelizing compiler for C++ [25]. For a particular synchronization optimization, they generate multiple versions of the relevant portion of code, each of which has been optimized with a different level of aggressiveness. The generated program alternates between sampling and production phases. During sampling, the program executes and times each of the versions. Thus, the sampling phase is essentially an instance of empirical search. During the (typically much longer) production phase, the best version detected during sampling executes. The length of each phase must be carefully selected to minimize the overall overhead of the approach. The program continues the sampling and production cycle, thus dynamically adjusting the optimization policies to suit the current application context. The dynamic feedback approach has been revisited and generalized in the ADAPT project, an extension of the Polaris parallelizing compiler [91]. The ADAPT framework provides more generalized mechanisms for "optimization writers" to specify how variants are generated, and how they may be heuristically pruned at run-time. In contast to the assumed model of run-time selection in Section 4, where the statistical models are generated off-line, in this dynamic feedback approach the models themselves must be generated at run-time during the sampling phase.

Kistler and Franz propose a sophisticated system architecture, built on top of the Oberon System 3 environment, for performing *continuous program optimization* [48]. They take a "whole systems" view in which the compiler, the dynamic

loader, and the operating system all participate in the code generation process. The compiler generates an executable in an intermediate binary representation. When the application is launched, this binary is translated into machine language, with minimal or no optimizations. The program is periodically sampled to collect profile data (such as frequency, time, or hardware counter statistics). A separate thread periodically examines the profile data to identify either bottlenecks or changes in application behavior that might warrant re-optimization, and generates a list of candidate procedures to optimize. An empirical cost-benefit analysis is used to decide which, if any, of these candidates should be re-optimized. The code image for re-optimized procedures is replaced on the fly with the new image, provided it is not currently executing. For the particular dynamic optimizations they consider in their prototype—trace-based instruction rescheduling and data reorganization—off-line search-based optimization still outperforms continuous re-optimization for BLAS routines. Nevertheless, their idea applies more generally and with some success on other irregular, non-numerical routines with dynamic (linked) data structures. However, the cost of continuous profiling and re-optimization are such that much of the benefit can be realized only for very long running programs, if at all.

6 Conclusions and Future Directions

For existing automatic tuning systems which 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 area for research. This article uses statistical methods to address some of the challenges which arise. Our survey of related work indicates that the use of empirical search-based tuning is widespread, and furthermore suggests that the methods proposed in this article will be relevant in a number of contexts besides kernel-centric tuning systems.

Among the current automatic tuning challenges is pruning the enormous implementation spaces. Existing tuning systems use 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).

Another challenge is finding 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.

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 [21]. Problems treated in this 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 [31]).

One possible criticism of the present study is that matrix multiply represents only one in many possible families of applications. However, our survey of Section 5 reveals that search-based methods have demonstrated their utility for other kernels in scientific application domains like discrete Fourier transform (DFT) and sparse matrix-vector multiply (SpMV). These other computational kernels differ from matrix multiply in that they have less computation per datum ($O(logn)$ flops per signal element in the case of the DFT, and 2 flops per matrix element in the case of SpMV), as well as additional memory indirection (in the case of SpMV). Moreover, search-based tuning has shown promise for non-numerical kernels such as sorting or parallel distributed collective communications (Section 5). The effectiveness of search in all of these examples suggests that a search-based methodology applies more generally.

In short, this work connects high performance software engineering with statistical modeling ideas. The idea of searching is being incorporated into a variety of software systems at the level of applications, compilers, and run-time systems, as our survey in Section 5 shows. This further emphasizes the relevance of search beyond specialized tuning systems.

Acknowledgements

We wish to thank Andrew Ng for his feedback on our statistical methodology. This research was supported in part by the National Science Foundation under NSF Cooperative Agreement No. ACI-9813362, NSF Cooperative Agreement No. ACI-9619020, the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and a gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] B. S. Andersen, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Y. Yalamov. LAWRA–Linear Algebra With Recursive Algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, September 1999.

[2] L. Arge, J. Chase, J. S. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. *ACM Journal on Experimental Algorithmics*, 6:1–18, 2001.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In *MICRO-33:*

Third ACM Workshop on Feedback-Directed Dynamic Optimization, Monterey, CA, USA, December 2000.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, Paris, France, December 1996.

[5] R. Barnes. Feedback-directed data cache optimizations for the x86. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture, Second Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

[6] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Saddayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[7] O. Beckmann and P. H. J. Kelley. Runtime interprocedural data placement optimization for lazy parallel libraries. In *EuroPar*, LNCS. Springer, August 1997.

[8] P. J. Bickel and K. A. Doksum. *Mathematical Statistics: Basic Ideas and Selected Topics*. Holden-Day, Inc., San Francisco, CA, 1977.

[9] A. J. C. Bik and H. A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31(1):14–24, 1995.

[10] 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.

[11] 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.

[12] 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.

[13] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum, 2001. `www.netlib.org/blas/blast-forum`.

[14] E. Brewer. High-level optimization via automated statistical modeling. In *Symposium on Parallel Architectures and Algorithms*, Santa Barbara, CA, USA, July 1995.

[15] D. Bruening, T. Garnett, and S. Amarsinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, San Francisco, CA, USA, March 2003.

[16] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, pages 114–124, 1992.

[17] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software–Practice & Experience*, 21(12):1301–1321, December 1991.

[18] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, USA, June 2001.

[19] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and LAPACK for clusters. Technical Report UT-CS-03-499, University of Tennessee, January 2003.

[20] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Second Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

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

[22] K. D. Cooper, T. J. Harvey, D. Subramanian, and L. Torczon. Compilation order matters. Technical Report –, Rice University, Houston, TX, USA, January 2002.

[23] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.

[24] J. D. Darcy. Finding a fast quicksort implementation for Java, Winter 2002. `www.sonic.net/∼jddarcy/Research/cs339-quicksort.pdf`.

[25] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[26] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[27] 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.

[28] J. Dongarra and V. Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science*, Melbourne, Australia, June 2003.

[29] B. B. Fraguela, R. Doallo, and E. L. Zapta. Automatic analytic modeling for the estimation of cache misses. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, Newport Beach, CA, USA, October 1999.

[30] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, July 1997.

[31] B. Frey. *Graphical Models for Machine Learning and Digital Communications*. MIT Press, Boston, MA, USA, 1998.

[32] 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*, Seattle, WA, USA, May 1998.

[33] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache–oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, New York, NY, October 1999.

[34] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proceedings of Supercomputing*, Portland, OR, USA, November 1999.

[35] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Proceedings of the 2nd European Conference on Parallel Processing (Euro-Par'96), Lyon, France*, volume 1123,1124 of *Lecture Notes in Computer Science*, pages 128–135. Springer-Verlag, Berlin, Germany, 1996. `www.mpi-forum.org`.

[36] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[37] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, November 2002.

[38] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.

[39] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992.

[40] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4), December 2001.

[41] J. W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, May 1981.

[42] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. PYTHIA-II: A knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):277–253, June 2000.

[43] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing*, August 1996.

[44] 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.*, San Antonio, TX, USA, March 1999.

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

[46] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. Technical Report 171, Compaq SRC, August 2001.

[47] B. Kagstrom, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, 1998.

[48] T. Kistler and M. Franz. Continuous program optimization: a case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[49] 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, Aussois, France, 2000.

[50] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1(2):105–133, April–June 1971.

[51] A. N. Ko and J. A. Izaguirre. MDSimAid: Automatic optimization of fast electrostatics algorithms for molecular simulations. In *Proceedings of the International Conference on Computational Science*, LNCS, Melbourne, Australia, 2003. Springer.

[52] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518, Stanford, CA, June 2000.

[53] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[54] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report TR-490, Dept. of Computer Science, Indiana University, September 1997.

[55] P. Liniker, O. Beckmann, and P. H. J. Kelly. Delayed evaluation, self-optimising software components as a programming model. In *Euro-Par*, Paderborn, Germany, August 2002.

[56] M. Lucks and I. Gladwell. Automated selection of mathematical software. *ACM Transactions on Mathematical Software*, 18(1):11–34, March 1992.

[57] H. Massalin. Superoptimizer–a look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, CA, USA, 1987.

[58] J. D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204. ACM, March 1995.

[59] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[60] 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, Sante Fe, NM, USA, May 2000.

[61] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 81–96, San Francisco, CA, May 2001. Springer.

[62] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[63] A. Nisbet. GAPS: Iterative feedback directed parallelization using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, Paris, France, June 1998.

[64] G. E. Noether. Note on the Kolmogorov statistic in the discrete case. *Metrika*, 7:115–116, 1963.

[65] J. H. Olsen and S. C. Skov. Cache-oblivious algorithms in practice. Master's thesis, University of Copenhagen, Copehagen, Denmark, 2002.

[66] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance—matrix multiply revisited. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[67] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 101–112, Portland, OR, USA, January 2002. ACM Press.

[68] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

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

[70] W. Pugh and T. Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

[71] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast automatic generation of DSP algorithms. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 97–106, San Francisco, CA, May 2001. Springer.

[72] N. Ramakrishnan and R. E. Valdés-Pérez. Note on generalization in experimental algorithmics. *ACM Transactions on Mathematical Software*, 26(4):568–580, December 2000.

[73] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[74] N. G. Santiago, D. T. Rover, and D. Rodriguez. A statistical approach for the analysis of the relation between low-level performance information, the code, and the environment. In *Proceedings of the ICPP 4th Workshop on High Performance Scientfic and Engineering Computing with Applications*, pages 282–289, Vancouver, British Columbia, Canada, August 2002.

[75] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In D.-Z. Du and M. Li, editors, *Computing and Combinatorics*, volume LNCS 959, pages 270–281, 1995.

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

[77] J. G. Siek and A. Lumsdaine. A rational approach to portable high performance: the Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (fast) library. In *Proceedings of ECOOP*, 1998.

[78] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, Boston, MA, USA, January 2000.

[79] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. Technical Report NC2-TR-1998-030, European Community ESPRIT Working Group in Neural and Computational Learning Theory, 1998. `www.neurocolt.com`.

[80] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2003.

[81] P. Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, August 1997.

[82] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of Supercomputing '98*, Orlando, FL, November 1998.

[83] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[84] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, San Francisco, CA, USA, March 2003.

[85] C. Tăpus, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

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

[87] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline-unrolling trade-offs. In *Proceedings of the 4th International Workshop on Compilers for Embedded Systems*, St. Goar, Germany, September 1999.

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

[89] T. Veldhuizen. Arrays in Blitz++. In *Proceedings of ISCOPE*, volume 1505 of *LNCS*. Springer-Verlag, 1998.

[90] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, PA, USA, 1998. SIAM.

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

[92] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[93] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[94] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander. Language support for Morton-order matrices. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 24–33, Snowbird, UT, USA, 2001. ACM Press.

[95] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[96] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 169–181, Vancouver, BC Canada, June 2000.

[97] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2003.

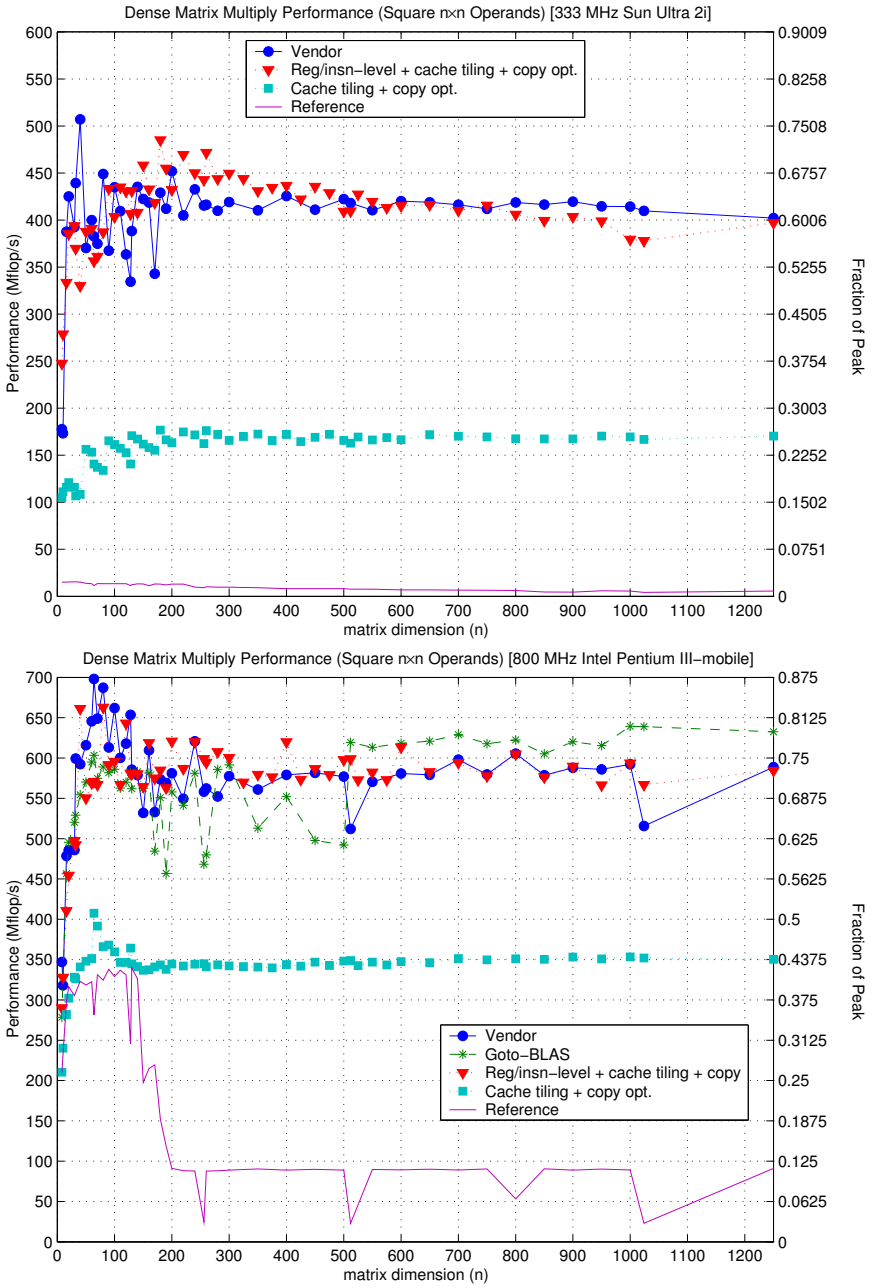

Figure 1: Performance (Mflop/s) of $n \times n$ matrix multiply for a workstation based on the Sun Ultra 2i processor (*top*) and an 800 MHz Mobile Pentium III processor (*bottom*). The theoretical peaks are 667 Mflop/s and 800 Mflop/s, respectively. We include values of n that are powers of 2. While copy optimization (shown by cyan squares) improves performance significantly compared to the reference (purple solid line), register and instruction level optimizations (red triangles) are critical to approaching the performance of hand-tuned code.

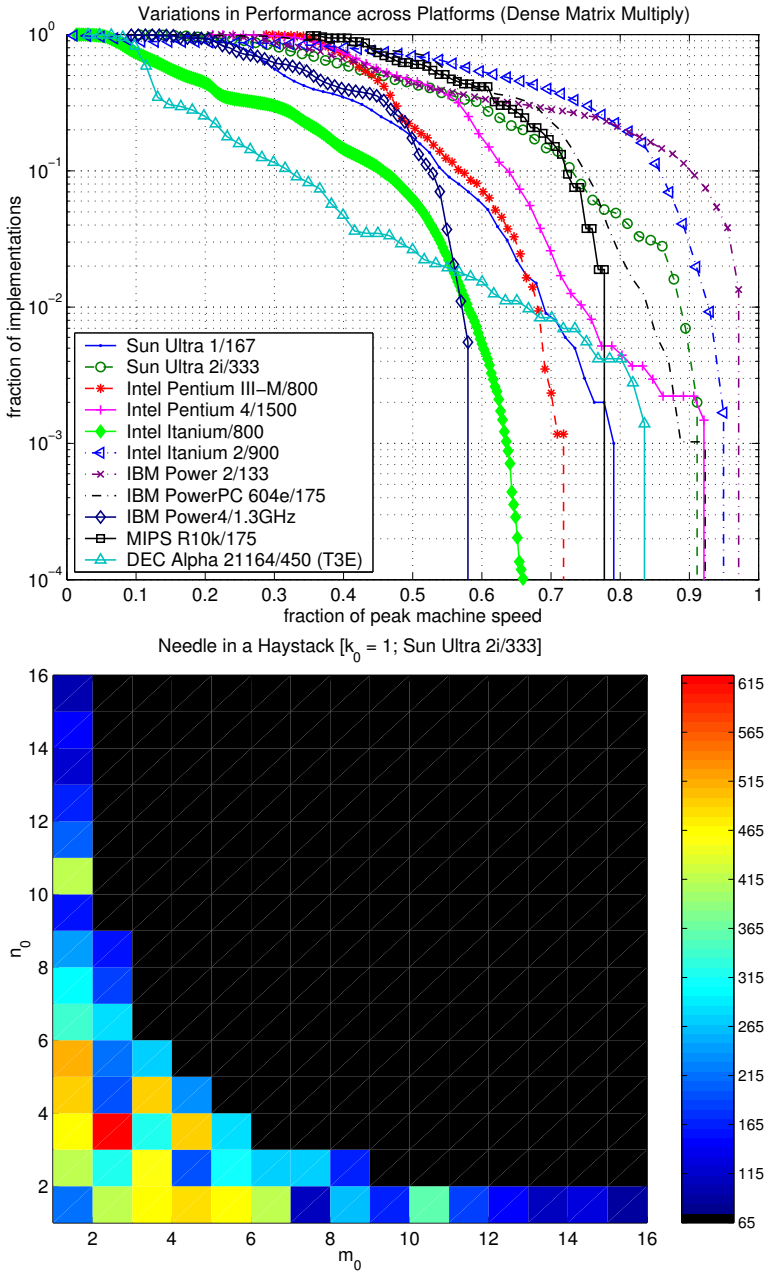

Figure 2: (*Top*) The fraction of implementations (y-axis) attaining at least a given level of peak machine speed (x-axis) on six platforms. The distributions of performance vary dramatically across platforms. (*Bottom*) A 2-D slice of the 3-D register tile space on the Sun Ultra 2i/333 platform. Each square represents an implementation ($m_0 \times 1 \times n_0$ tile) shaded by performance (color-scale in Mflop/s). The fastest occurs at ($m_0 = 2, n_0 = 3$), having achieved 615 Mflop/s out of a 667 Mflop/s peak. The dark region extending to the upper-right has been pruned from the search. Finding the optimal point in these highly irregular spaces can be like looking for a "needle in a haystack."

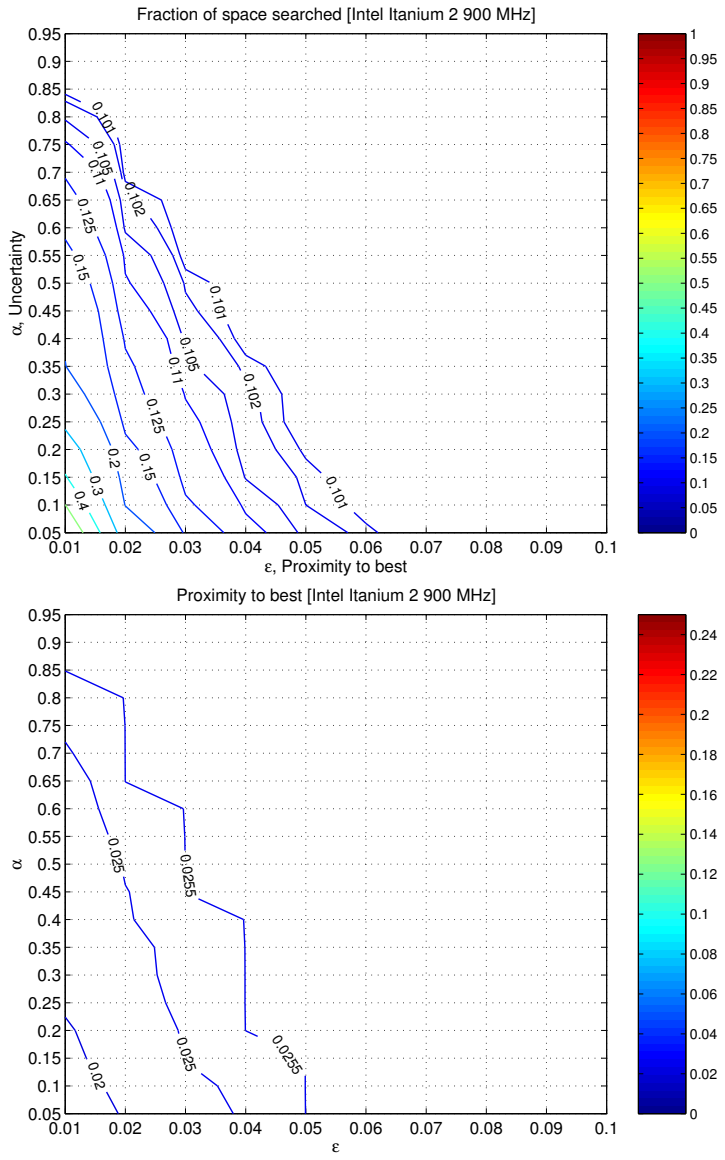

Figure 3: Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Intel Itanium 2/900 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

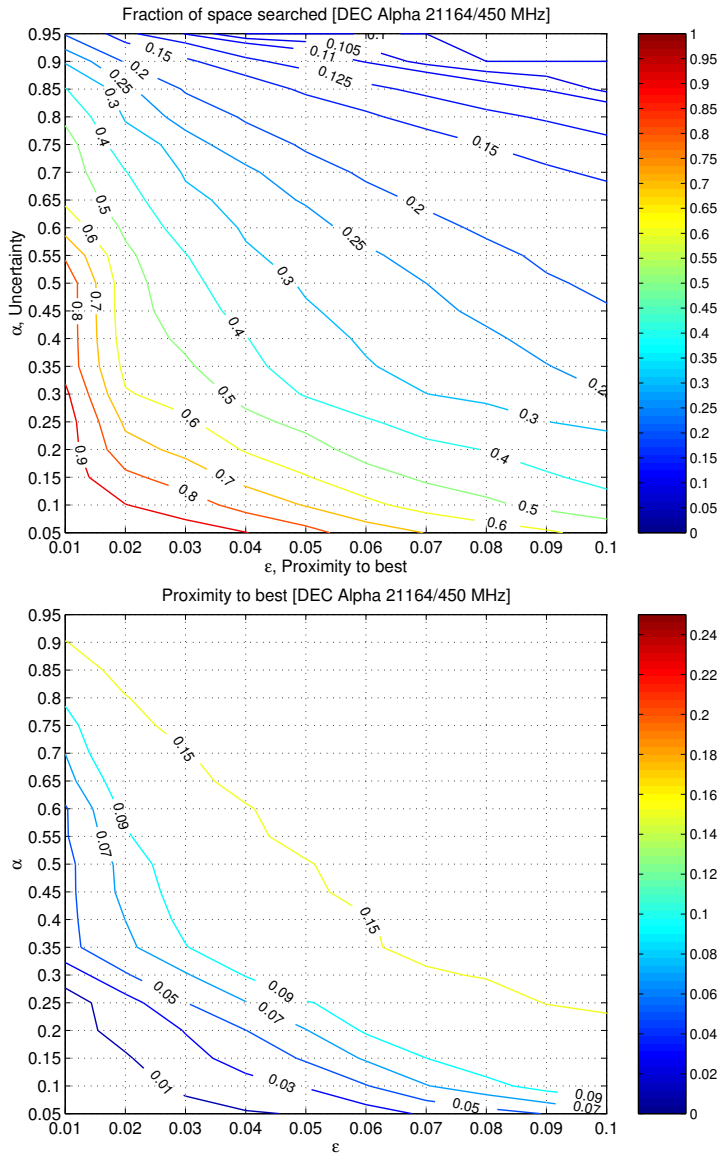

Figure 4: Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the DEC Alpha 21164/450 (Cray T3E node) platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

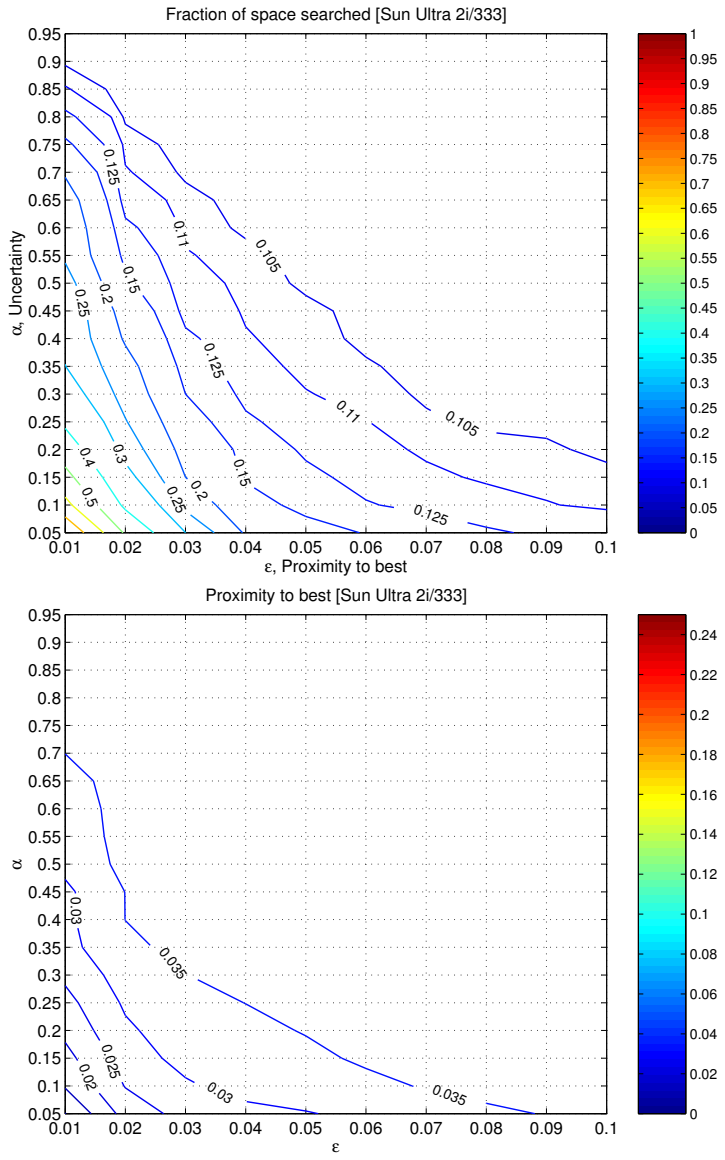

Figure 5: Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Sun Ultra 2i/333 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

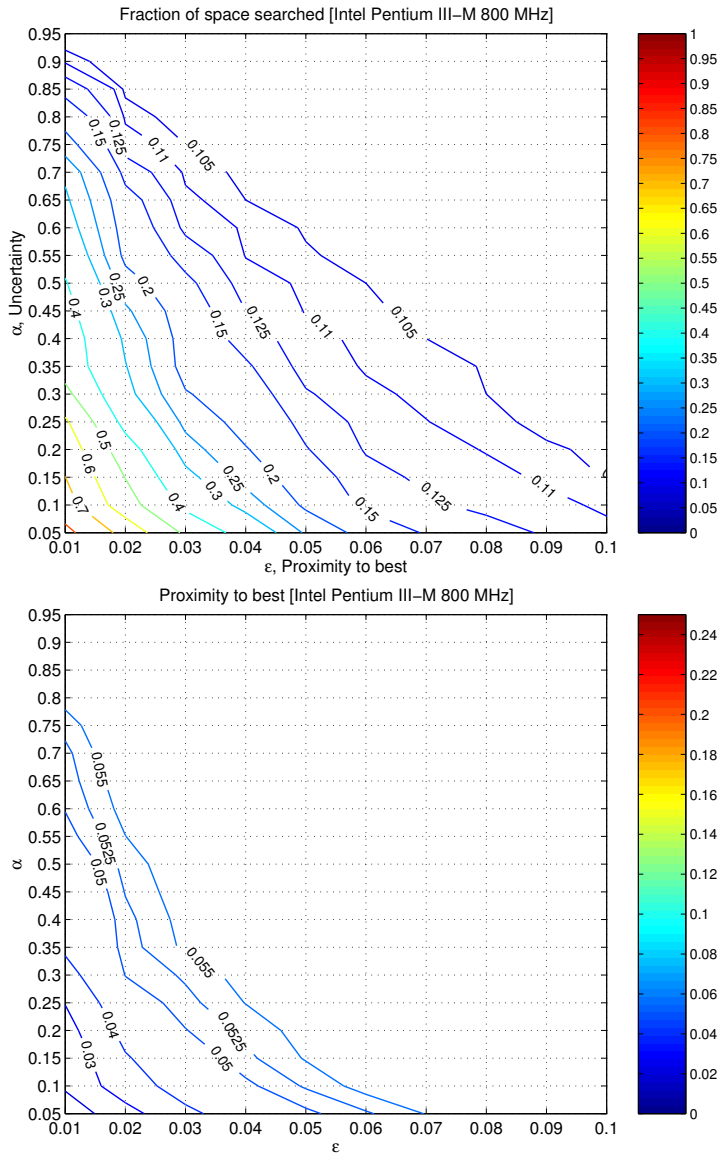

Figure 6: Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Intel Mobile Pentium III/800 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

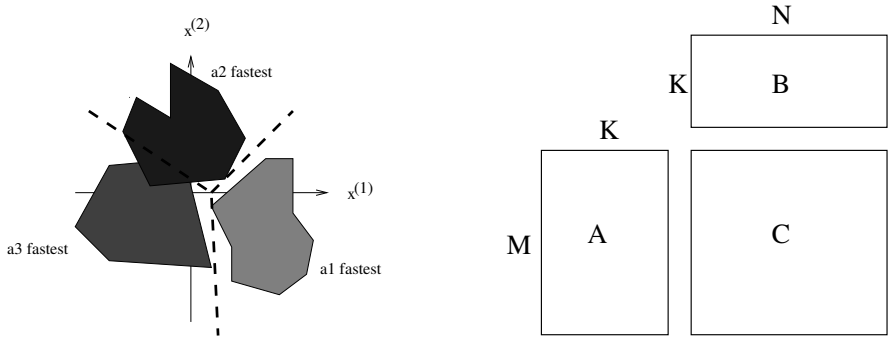

Figure 7: (*Left*) Geometric interpetation of the run-time selection problem: A hypothetical 2-D input space in which one of three algorithms runs fastest in some region of the space. Our goal is to partition the input space by algorithm. (*Right*) The matrix multiply operation $C \leftarrow C + AB$ is specified by three dimensions, M, K, and N.

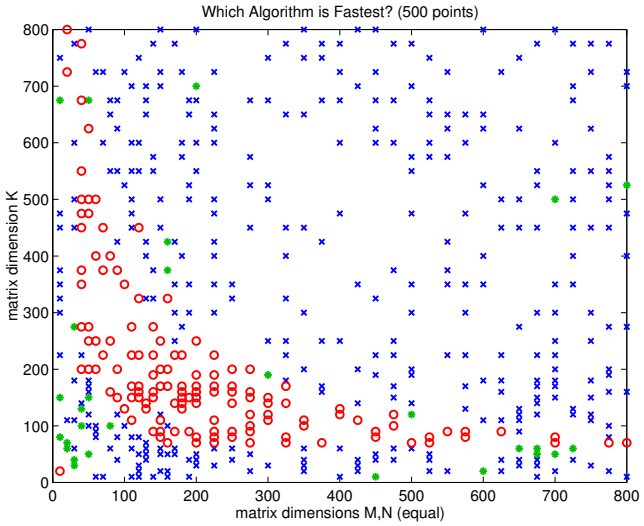

Figure 8: A "truth map" showing the regions in which particular implementations are fastest. The points shown represent a 500-point sample of a 2-D slice (specifically, $M = N$) of the input space. An implementation with only register tiling is shown with a red `o`; one with L1 and register tiling is 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%.

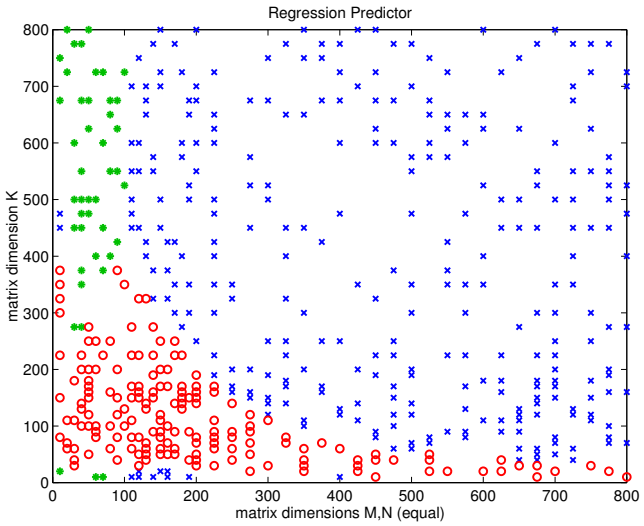

Figure 9: Sample classification results for the regression predictor on the same 500-point sample shown in Figure 8. The average misclassification rate for this predictor was 34%.

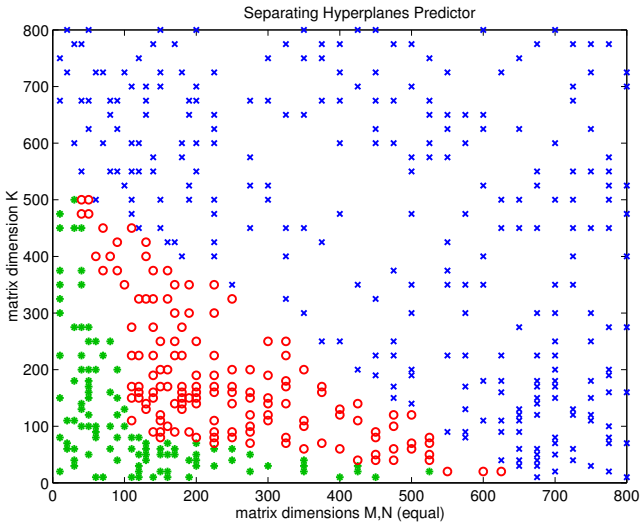

Figure 10: Sample classification results for the separating hyperplanes predictor on the same 500-point sample shown in Figure 8. The average misclassification rate for this predictor was 31%.

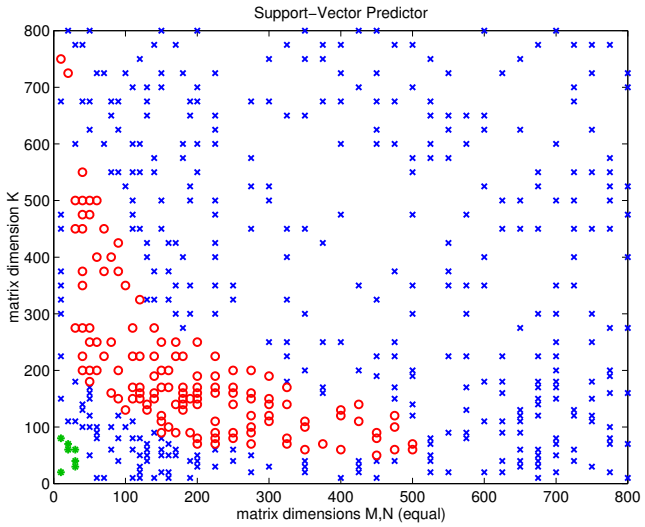

Figure 11: Sample classification results for the support vector predictor on the same 500-point sample shown in Figure 8. The average misclassification rate for this predictor was 12%.

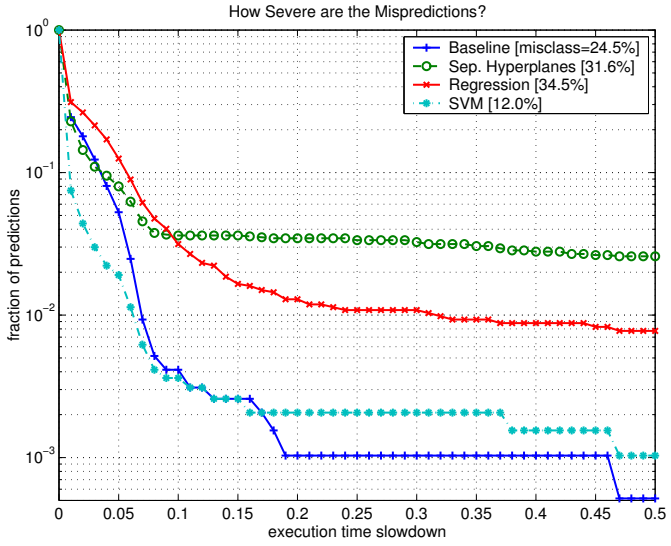

Figure 12: Each line corresponds to the distribution of slow-downs due to mispredictions on a 1936 point sample for a particular predictor. A point on a given line indicates what fraction of predictions (y-axis) resulted in more than a particular slow-down (x-axis). Note the logarthmic scale on the y-axis.