Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization

Arkady Epshteyn[1], Maria Garzaran[1], Gerald DeJong[1], David Padua[1], Gang Ren[1], Xiaoming Li[1], Kamen Yotov[2], and Keshav Pingali[2]

[1] University of Illinois at Urbana-Champaign, Urbana IL 61801
[2] Cornell University, Ithaca, NY 14853

Abstract. Compilers employ system models, sometimes implicitly, to make code optimization decisions. These models are analytic; they reflect their implementor's understanding and beliefs of the system. While their decisions can be made almost instantaneously, unless the model is perfect their decisions may be flawed. To avoid exercising unique characteristics of a particular machine, such models are necessarily general and conservative. An alternative is to construct an empirical model. Building an empirical model involves extensive search of a parameter space to determine optimal settings. But this search is performed on the actual machine on which the compiler is to be deployed so that, once constructed, its decisions automatically reflect any eccentricities of the target system. Unfortunately, constructing accurate empirical models is expensive and, therefore, their applicability is limited to library generators such as ATLAS and FFTW. Here the high up-front installation cost can amortized over many future uses. In this paper we examine a hybrid approach. Active learning in an Explanation-Based paradigm allows the hybrid system to greatly increase the search range while drastically reducing the search time. Individual search points are analyzed for their information content using an known-imprecise qualitative analytic model. Next-search-points are chosen which have the highest expected information content with respect to refinement of the empirical model being constructed. To evaluate our approach we compare it with a leading analytic model and a leading empirical model. Our results show that the performance of the libraries generated using the hybrid approach is comparable to the performance of libraries generated via extensive search techniques and much better than that of the libraries generated by optimization based solely on an analytic model.

1 Introduction

Application of high-level program transformations such as loop unrolling, array tiling, and software pipelining is critical in optimizing the performance of compiled code. Deciding how to apply these transformations can be exceedingly challenging. These decisions must balance subtle interactions among characteristics of the underlying architecture, the source code, other compilation decisions, and so on. Every optimizing compiler, therefore, embodies a decision procedure either explicitly or implicitly to resolve these choices. Intuitions (confirmed by decision theory) tell us that resolving such difficult choices satisfactorily requires a great deal of information.

Most commonly, this information is supplied explicitly via prior performance models. Such models are extremely efficient, generating solutions almost instantaneously.

But the information they embody comes entirely from their designer's formal idealization of the process to be optimized. It excludes phenomena that the designer believes to be negligible or too complex to analyze.

By contrast, an empirical approach collects information directly from the system on which the compiler is deployed. This results in first-hand information which can be more accurate than that of a prior performance model. For example, many versions of a loop with different tilings crossed with various loop unrolling amounts might be generated and executed. It then selects the combination with the best measured performance. Unfortunately, searching through combinations of parameter values can be hugely expensive. As a result, this approach cannot service the real-time requests of a compiler as can the prior performance model. But it is well suited to library generation where the high cost of optimal configuration decisions can be paid once. Well-known library generators that employ empirical optimization include FFTW [12], ATLAS [17], PhiPAC [3] and SPIRAL [19].

An alternative decision procedure is an adaptive hybrid which includes only the prior information from the designer which he or she is most confident of. The rest is then filled in empirically. The prior partial model might answer some optimization questions directly but might instead suggest which measurements are likely to be most informative and so guide and limit the empirical searches. The accuracy of this decision procedure is rooted in first-hand measurement of the actual system to be optimized. But it might be efficient enough to make real-time optimization decisions or be automatically re-invoked when necessary to react to changing situations.

The possibility of adaptive models is the motivation and the subject of our current research which we offer as the first tentative steps along a lengthy but, we believe, promising path. We employ an Explanation-Based Learning paradigm [10]. Empirical results are treated as illustrations or manifestations of a deeper pattern to be discovered. They are *explained* in terms of the existing partial model and therefore serve to refine the model and reduce the need for future empirical searches.

To evaluate our adaptive approach we compare it directly with a leading analytic model and a leading empirical optimization approach. Methodologically, these three approaches must be compared on equal footing. They be applied to the same optimization task in as similar a setting as possible. To this end we use the matrix multiplication framework of ATLAS as our experimental platform but without its hand-tuned additions whose influences could be conflated with the behaviors we wish to monitor.

ATLAS produces an optimized Basic Linear Algebra Subroutine (BLAS) library including a module for optimized matrix multiplication. The generated code (referred to in this paper as the mini-MMM code) is compiled and executed to measure its observed performance. ATLAS finds parameter values that maximize the performance of mini-MMM code (in MFLOPs) using a routine that performs a near-exhaustive sampling of a region of the parameter space. It is this module that we replace in our experiments. In one experimental condition it is replaced by a leading analytic model [20], in a second it is replaced by our adaptive system, and in a third the original ATLAS routine is employed. In all three cases the remainder of the MMM generation code is unchanged as are the routines to measure MMM performance.

Our results confirm that the the adaptive approach can perform better than the analytic model and is much more efficient than the empirical approach. The analytic model is based on an architectural idealization that cannot perfectly capture the actual machine to be optimized. On the other hand, the ATLAS routine samples broadly from a large but limited region of the parameter space that, on occasion does not contain the optimal configuration. The adaptive approach only samples those points deemed to be informative given the results of previous samples. This can greatly increase the range of parameter values it entertains, but it only does so when there is an expectation of optimization improvement.

In library installation efficiency is less crucial since cost can be amortized over the lifetime of the machine. But even here there are at least four situations in which efficiency can be important.

1) Adaptation may have to be applied at runtime, in which case an extensive search is not possible, and prior models (when available) may not be accurate enough. This type of search involves measuring the performance of various versions of pre-compiled code during the sampling phase of the executing, and then using the best version during the (much longer) production phase [11]. Note that runtime searching tailors the optimization system to the requirements of the user not available at library installation time (for instance, small blocking parameter values will be selected if the user only multiplies small matrices).
2) Efficient adaptation can be applied at the time of compilation. [16] describes a compile-time optimization framework that employs empirical search which receives performance feedback from a fast estimator.
3) The space of possible versions can be too large even for once-in-a-life time installation. Empirical search complexity grows exponentially with the number of interacting optimization parameters.
4) An interesting application of library routines is as a benchmark to evaluate alternative machine designs. More efficient adaptation can enable a wider exploration of possible designs.

The paper is organized as follows: we describe the search module of ATLAS in Section 2. The model approach to optimization is discussed in Section 3. Our hybrid approach is presented in Section 4. Finally, experimental results are shown in Section 5.

2 ATLAS

ATLAS is a system that employs empirical search to generate highly-tuned BLAS libraries [17]. In this paper, we focus on the optimization of the matrix-matrix multiplication (MMM) routine. This is the key routine in BLAS since many other kernel operations use it as a primitive. ATLAS contains a generator search module and a multiple implementations search module. The generator search contains a code generator that outputs a kernel based on input parameters. This module searches the inputs that result in the best performing kernel. The multiple implementation module searches among

hand-written codes for MMM kernels. ATLAS selects the best-performing kernel out of both modules. ATLAS also records results from previous installations on the target platform and can reduce the installation time by using these instead of the empirical search.

In this work, we focus on the generator search module. The search is used during the installation procedure to find the optimal values of code transformation parameters (amount of tiling, unrolling, etc.). It consists of: (1) generating the versions of matrix multiplication with the parameter values to be tested, (2) compiling and executing them, and (3) selecting the version that perform best.

ATLAS is not a restructuring compiler, but the code generated by ATLAS can be seen as the result of applying a sequence of compiler transformations. We first examine these code transformations (Section 2.1). Then, we explain how ATLAS searches for the most appropriate parameter values of these transformations (Section 2.2).

2.1 Transformations

```
for (j = 1; j <= M; j++)
 for (i = 1; i <= N; i++)
  for (k = 1; k <= K; k++)
   C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

Fig. 1. Matrix Multiplication Code

The code implementing a MMM is shown in Figure 1. Yotov et al [20,21] and Cooper et al [8] found that computing this matrix multiplication using the library generated by ATLAS results in higher performance than that obtained when the naive MMM implementation in Figure 1 is compiled using a general purpose compiler. The reason for this performace gap is that compilers do not apply the appropriate transformations and/or they do not use the correct parameter values for these transformations [8,20,21].

The code generated by ATLAS can be seen as the result of applying well-known compiler transformations to the code in Figure 1. To increase the locality ATLAS uses blocking, while to increase Instruction Level Parallelism (ILP) ATLAS uses pipeline scheduling. Next, we examine these transformations.

- Blocking: This transformation converts matrix multiplication into a sequence of smaller matrix multiplications. Blocking can be accomplished by a loop transformation called tiling, which was introduced by Wolfe [18]. ATLAS applies blocking at the cache and the register level:
  - Cache Blocking: ATLAS uses blocking to decompose the matrix multiplication of large matrices into the multiplication of smaller sub-blocks. The size of each sub-block is $NB \times NB$, where NB is an optimization parameter that needs to be chosen so that the working set of the sub-blocks being multiplied fits in the cache [4,7,18]. We call the resulting code mini-MMM.

- Register blocking: The mini-MMM code itself is blocked and then unrolled to optimize the utilization of the registers. The resulting code, that we call micro-MMM, multiplies a column of *MU* elements of matrix *A* by a row of *NU* elements of matrix *B* and stores the result into a $MU \times NU$ sub-matrix of *C*. *MU* and *NU* are optimization parameters that must be chosen so that $MU + NU + MU \times NU$ fit in the registers of the processor [2].

To improve register allocation, ATLAS uses scalar replacement [5]: each element of A, B and C that is accessed in the unrolled micro-MMM code is assigned to a scalar. The array accesses in the micro-MMM code are replaced by these scalar variables. ATLAS expects that the compiler will assign registers to these scalars. Also, ATLAS copies the $NB \times NB$ sub-matrices to consecutive memory locations. This reduces the number of cache and TLB misses. Additional transformations such as loop unrolling and load scheduling applied in ATLAS are described in detail in [17,20,21].

2.2 Search

ATLAS does an almost exhaustive search of the parameter values presented in the previous Section. Since ATLAS searches for several parameters, when searching for one parameter, ATLAS needs to assign values to the other parameters it has not yet optimized. These values are initially assigned based on results obtained from the execution of benchmarks. These benchmarks estimate characteristics of the platform on which ATLAS is being installed, such as cache size and number of registers. After a parameter is optimized, the value that obtains the best performance is used for the search of the subsequent parameters. Parameter values are searched in the same order that appears in our explanation below.

1. L1 cache blocking ($NB \times NB$): ATLAS generates versions of the mini-MMM code with a matrix size $NB \times NB$, where *NB* varies from 16 to the minimum of (80 and $\sqrt{L1\,Size}$), in steps of 4.
2. Register blocking (*MU* and *NU*): ATLAS exhaustively searches for the best values of *MU* and *NU*. All possible combinations of *MU* and *NU* satisfying $MU \times NU + MU + NU + Latency \leq Number\ Of\ Registers$ are tried, and the best performing combination is selected.
3. Loop unrolling, instruction scheduling parameters, etc. are described in [17,20]

More details about ATLAS can be found in [17,20].

3 Model

Yotov et al. [20,21] challenged the notion that empirical optimization is more effective than model-driven optimization by demonstrating that a model-based optimization strategy can calculate near-optimal parameter values without incurring the sampling cost of empirical search. We use Yotov's model as our initial guess of the parameter values. We also compare the experimental results obtained by our approach with the

results obtained by Yotov's model. Thus, in this Section we summarize it. A further description of the model can be found in [20,21].

The model depends on accurate estimates of machine parameters that include the L1 cache and line size, the number of registers, the latency of the multiply instruction, the existence of a fused multiply-add instruction, and the number of functional units.

1. L1 cache blocking ($NB \times NB$): The idea of the model is to compute the value of *NB* that optimizes the use of the L1 data cache. The model is based on the memory access trace of the mini-MMM, and takes into account the loop order, L1 cache and line size, and the LRU replacement strategy of caches. This analysis finds that for a JIK order, the optimal value for *NB* is the maximum value of *NB* that satisfies the inequality below:

$$\left\lceil \frac{NB^2}{L1\,Line\,Size} \right\rceil + 3 * \left\lceil \frac{NB}{L1\,Line\,Size} \right\rceil + 1 \leq \frac{L1\,Size}{L1\,Line\,Size}$$

   Notice that the model in [21] is more accurate that the one just discussed. In fact, the model in [21] also considers interactions between the L1 cache and the register file and avoids the need for micro-MMM cleanup code by choosing the value of *NB* so that it is a multiple of *MU* and *NU*. We started the work reported in this paper before the model was improved and we are using the simpler model from [20]. In any case the value found using the more elaborate model in [21] is close to the value found by the model described above and presented in [20].
2. Register blocking (*MU* and *NU*): To estimate the appropriate values of the register blocking parameters, the model takes into account how the ATLAS generator allocates registers to variables, and the need of *Latency* additional registers to hold the temporary results of the multiplication. With all this, the model picks the maximum values of *MU* and *NU* such that $NU \approx MU$ and $MU \times NU + MU + NU + Latency \leq Number\ Of\ Registers$.

The model just presented mimics ATLAS in that it computes a blocking value for the L1 cache. However, sensitivity analysis reported in [20,21] shows that in some machines blocking values that overflow the L1 cache obtain better performance. The conjecture is that, in these machines, the large block size that results in the best performance corresponds to the block size that fits in the L2 cache. Blocking for L2 may result in higher performance than blocking for the L1 cache because in out-of-order processors, which have a deep pipeline, the latency of accessing the L2 cache can usually be hidden without stalling the processor. The rationale is that the processor can continue executing instructions that do not depend on the missed data. A larger block size also increases the opportunity for higher ILP and for the compiler to reorder instructions [6]. Notice that tiling for L2 may not always be the best choice, because large tiles can result in more time spent in the cleanup code, which can degrade performance for some of the codes calling the MMM library generated by ATLAS [1]. However, it has been shown that in some cases it is necessary to tile for L2 [1], and this is confirmed by our experiments (Figure 5) where the MMM library generated by ATLAS is evaluated in the contex of matrix-matrix multiplication.

Given that for some behavioral profiles it may be advantageous to block for the L2 cache, we would like to extend the model from [20,21] to estimate an appropriate L2

blocking parameter value. The inequality above used to compute the L1 cache blocking factor cannot be used to compute the L2 cache blocking factor because it does not take conflict misses into account. Ignoring conflict misses in the L1 cache is safer than ignoring conflict misses in the L2 cache because the difference in latencies between the L1 and L2 caches is much smaller than the difference in latencies between the L2 cache and the main memory.

To compute the L2 blocking factor we use a conservative approach that ensures that *NB*x*NB* blocks of data from all three matrices *A, B*, and *C* fit in the L2 cache. This happens when the combined size of these three blocks ($3 * NB^2$) is equal to the size of the L2 cache.

4 Adaptive Modeling

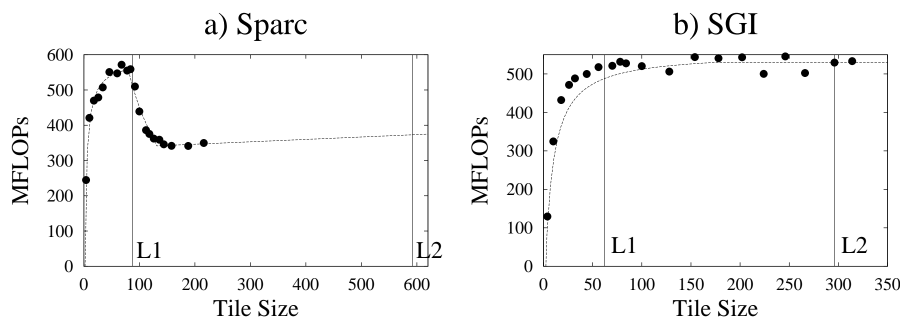

Fig. 2. Performance as a function of cache block size *NB* (complete instruction cache unroll: *KU=NB*)

Our adaptive approach combines the information embedded in the model from Section 3 with feedback information obtained from the execution of versions of the mini-MMM code. Both types of information are used to search for the maximum of the mini-MMM performance function. The approach determines the shape of this function through experimentation. Each experiment consists of generating, compiling, and executing mini-MMM code. The mini-MMM code is generated by ATLAS's code generation module, ensuring that the space of available transformations is the same for ATLAS search and the adaptive approach. The parameter values for transformations, however, are determined by our algorithm. The feedback provided by each experiment (in form of mini-MMM performance) is used to design subsequent experiments to maximize information about the location of performance-maximizing parameter values. Maximizing performance can be done either via a local search (e.g., by performing hill climbing) or by modeling the whole performance function globally via appropriately chosen *regression curves*. Experiments that provide the best feedback about the shape of the regression function are preferred. The location of the maximum in this scenario is determined indirectly from the shape of the regression function. Prior knowledge obtained from the model is used to indicate to the family of regression curves that the maximum performance is going to be located in the neighborhood of the model-predicted values.

In our experiments, we focus on optimizing the cache blocking parameter *(NB)*: This is done by analyzing the general shape of the plot of mini-MMM performance as a function of the cache blocking parameters. Figure 2, for example, shows sampled data collected on two different machines. In each plot, the points show the performance of the mini-MMM code (Y-axis) for different values of cache block size (X-axis). As these sampled points are being collected, a regression curve is fitted to the data (this curve is shown in Figure 2 as well). The shape of the curve is adjusted with each newly collected sample point. The best values of the optimization parameters can be determined directly from the location of the maximum point on the regression curve.

Certain characteristics of the shape of the plot can be guessed before any data is collected. For example, we expect the peak in the curve of Figure 2-(a) to coincide with the optimal cache blocking (*NB*) factor predicted by the model. This is the point where the L1 cache is fully utilized. Further increase in the block size results in L1 cache overflow that results in performance degradation. We expect to see a phenomenon similar to this on most of the architectures under consideration. Information about the shape of the performance curve that is available before any data is collected is known as prior information. In statistics, prior information is captured by a probability distribution in the space of optimization parameters. We use the model from Section 3 to construct such a distribution.

Optimization requires a sophisticated algorithm because multiple levels of the cache hierarchy introduce multiple local maxima in the performance function. For example Figure 3 shows the performance obtained by the mini-MMM code as the tile size increases. On Pentium III, the figure shows two distinct peaks, each corresponding to blocking factors for L1 and L2 caches. Our optimization algorithm is described in detail in Section 4.1.

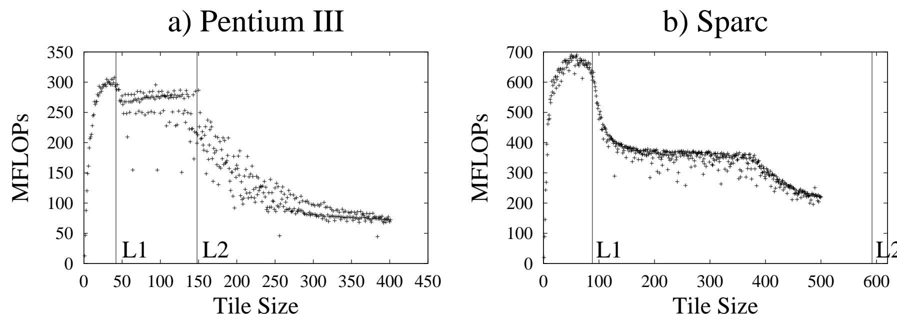

Fig. 3. Complete sampled performance curves on two machines. The vertical lines correspond to the blocking factors for L1 and L2 as predicted by the model

4.1 Cache blocking parameters

The adaptive approach constructs a nonlinear regression curve representing the sampled performance of the mini-MMM code as a function of tile size, with register blocking parameters being held constant. Figure 2 shows examples of such curves fitted to the data collected on the platforms that we evaluate in Section 5.2.

The optimal tile size is calculated directly from the fitted regression function. Thus each point provides global information about the location of the maximum by affecting the shape of the regression curve. A set of regression curves that could fit the data sample is hard-coded and available before any data is collected. In this work, we use a double-peaked family of regression curves, each peak corresponding to a blocking factor for one of the caches.

The details of our estimation algorithm are described in subsequent sections.

Model as prior information A typical performance profile is presented in Figure 3-(a). We expect performance to improve until the L1 cache is fully utilized. At that point, it drops off, but begins to improve again as the tile size increases until it reaches the point where L2 cache is fully utilized. The regression curves where the maxima are located at the model-predicted locations are initially favored. As more data is collected, the preference of the system is shifted towards the regression curves that fit the data best. This trade-off is governed by the size of the collected sample. The Bayesian framework determines precisely how to quantify this trade-off. The optimal regression curve chosen by the algorithm represents a balance between respecting the data and respecting the model-based prior information.

Instead of taking the model-predicted cache blocking factors for granted, we impose a probability distribution on the space of tile sizes that defines, for each set of tile sizes [l_1,l_2] the probability that the first peak occurs at l_1, while the second peak occurs at l_2. Initially, this probability is maximized at model-predicted cache blocking factors L_1 and L_2 described in Section 3, and shown in the plots in Figure 2. In our setup, we use the Normal probability distribution (N) centered at L_1 and L_2: $\pi(l_1, l_2) = N(\begin{bmatrix} L_1 \\ L_2 \end{bmatrix}, \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix})$, where σ_1^2 and σ_2^2 are user-controlled parameters representing one's confidence in the model's prediction.

Let $\beta = (w; l_1, l_2)$ be the complete set of parameters defining the regression curve. β includes the tile sizes l_1, l_2 and all the other parameters w necessary to completely specify the regression curve. The Bayesian (maximum a posteriori) approach to estimation involves updating the probability distribution over β after we see the sample D using Bayes' rule as follows: $P(\beta|D) = P(D|\beta)\pi(\beta)/P(D) \propto P(D|\beta)\pi(l_1, l_2)$ and picking the regression curve $\widehat{\beta}$ that maximizes $P(\beta|D)$ [14]. Notice that maximizing the posterior involves a trade-off between fitting the data ($P(D|\beta)$) and respecting the prediction of the model ($\pi(l_1, l_2)$).

$P(D|\beta) = (\frac{1}{\sqrt{2\pi\sigma^2}})^n e^{-\sum_{i=1}^{n} (performance_i - \beta(tile\ size_i))^2/(2\sigma^2)}$ is the distribution of n data points $(tile\ size_1, performance_1)$... $(tile\ size_n, performance_n)$ in the sample D with respect to the regression curve β assuming independent identically distributed gaussian noise. This term depends on the total squared error $\sum(performance_i - \beta(tile\ size_i))^2$. It favors the curves β that fit the data well. $\pi(\beta)$, on the other hand, favors the curves that agree with the model. As the sample size increases, more points contribute to the total squared error and penalize the curves that do not fit the data more heavily, while $\pi(\beta)$ remains unchanged. Thus, the system con-

verges to the best regression curve in the limit even if the prior information is inaccurate, but this convergence happens much faster when the model is good.

Active Sampling The search performs a dual function. First of all, prior knowledge may be inaccurate. Figure 3-(b) shows an example of a peak that does not coincide with any of the predicted blocking factors. Search can verify the tile sizes that fully utilize the caches and adjust them empirically. Second of all, prior knowledge alone does not indicate which cache (L1 or L2) to tile for (see Figure 3-(a)). Search resolves this problem by empirically determining which peak is the dominant one. Moreover, the adaptive search produces a statistical measure of confidence in its estimate that is not available with either pure model or ATLAS search.

As the search is conducted, each sample point is generated by 1) selecting some parameter values for optimization parameters, 2) generating a mini-MMM program based on those parameters, 3) compiling the program, and 4) measuring the program's execution time. The main source of efficiency of the search comes from its ability to select informative sample points intelligently. This process, known as active sampling, represents a major deviation from the philosophy of ATLAS and other empirical optimization engines - the system uses feedback from conducted experiments to adjust its sampling strategy, while ATLAS samples at pre-determined locations.

In doing so, it must take into account conflicting objectives: reducing the time to collect the sample and selecting the most informative points. The first objective directs the system to sample points close to the origin, because the sampling time increases with increasing tile size *NB* due mainly to the significant increase in the amount of time required to compile the program.[3] The second objective is to select the points that provide more information about the location of the peak of the function.

To reconcile these objectives, a heuristic that simulates potential fields is used. It places a negative charge at each sample point to discourage oversampling in the same region and a positive charge at the origin to encourage less time-consuming data points (since programs generated with smaller values of cache blocking/unrolling take less time to compile). Positive charges encourage sampling in the region around them, negative charges have the opposite effect. The point that minimizes the potential field is selected for sampling. A positive charge is also imposed on regions contributing information about the highest peak. This charge is proportional to the estimated probability that the peak that appears to be the highest actually is the highest.

An example of the heuristic can be seen in Figure 4. The potential field $U(x)$ is a function of the tile size x. Tile sizes with low potentials experience the least amount of repulsive force and the greatest amount of attractive force. The system computes $U(x)$ for every tile size x and chooses the tile size with minimum potential energy for sampling. The potential field is calculated as a sum of contributing factors. Each previously sampled tile size y contributes $\frac{\nu}{(x-y)^2}$ to the potential field at x, creating a repulsive force that increases at tile sizes x close to the sampled point y. The attractive field at the origin contributes $\xi * (x-0)^2$ to the potential field at x, resulting in an attractive force

[3] With bigger tile sizes, the size of the completely unrolled register loop nest increases, forcing the optimizing compiler to spend more time on instruction scheduling. Increasing the cache block size from 40 to 400 on the SGI machine increases compilation time from 4 seconds to 4 minutes.

that decreases with increasing tile size. ν and ξ are user-defined constants controlling the strengths of the forces creating the field. The advantage of using this heuristic is its efficiency in combining multiple objectives.

Examples of application of this heuristic are presented in Figure 2. In Figure 2-(a), a two-peaked function is used to fit the data. The first peak (blocking for the L1 cache) is the dominant one. The location of the L1 peak is estimated by the system from the sampled data. The L2 peak is predicted from the intersection of the regression function that fits the data and the location of the L2 blocking factor determined by the prior knowledge. The uncertainty of the estimated regression curve parameters is used to calculate the probability that blocking for the L1 peak yields better performance than blocking for L2. This probability, in turn, forces the sampling heuristic to direct its attention to the points that contribute information about the L1 peak. This, in conjunction with the fact that smaller block sizes correspond to less expensive sample points (in terms of compilation time), prompts the system to direct its attention to the region around the L1 blocking factors.

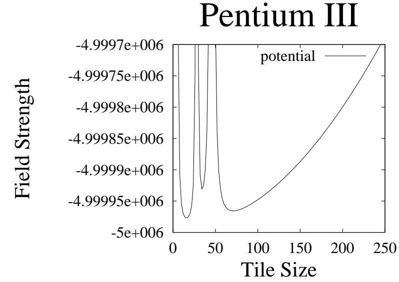

Fig. 4. Potential Field for Active Sampling. The field is constructed based on three sample points. It increases away from the origin and at previously sampled locations.

In Figure 2-(b), a different performance profile results in a different sampling behavior. In this architecture, the optimal cache block size must take advantage of the L2 cache. After the system determines that the dominant peak lies beyond the L1 saturation point, it attempts to collect as much information as possible to ascertain how to take advantage of the L2 cache, even at the expense of incurring a higher sampling cost. It does not make any sense to sample at lower tile sizes if these points do not provide any information about the predicted optimal peak of the model.

5 Experimental Results

In this section, we evaluate the adaptive optimization algorithm. The environmental setup used for our experiments is discussed in Section 5.1 and performance results are shown in Section 5.2. Our experiments demonstrate the feasibility of application of our approach by showing that the adaptive model can achieve performance comparable to (and sometimes exceeding that of) ATLAS and outperform the analytic model, while requiring many fewer experiments than an exhaustive search.

5.1 Environmental Setup

Our experiments were performed on two different architectural platforms: Ultra Sparc III and SGI R12000). Table 1 lists the salient architectural parameters of each platform[4].

[4] ATLAS compiler and options are the defaults that ATLAS selects in each target platform

	Sparc	SGI
CPU	Ultra Sparc III	R12000
Frequency	750 MHz	300 MHz
L1d/L1i Cache	64 KB/32 KB	32 KB/32 KB
L2 Cache	8 MB	2 MB
Memory	4 GB	512 MB
OS	SunOS 5.8	IRIX64 v6.5
ATLAS Compiler	Workshop cc v5.0	MIPSPro cc v7.30
ATLAS Compiler Options	-dalign -fsingle -xO2 -native	-O3 -64 -OPT:Olimit=15000 -TARG:platform=IP30 -LNO:blocking=OFF -LOPT:alias=typed

Table 1. Test Platforms

The following algorithms were executed on each platform:

1) Model: We use the model from [20] described in Section 3. The model assumes that tiling for the L1 cache is usually optimal.
2) ATLAS search: This is the search strategy using the code generator as described in Section 2. ATLAS assumes that tiling for the L1 cache is optimal for these architectures, and performs a near-exhaustive search of the cache tile space from 16 to the minimum of (80 and $\sqrt{L1\,Cache\,Size}$), in steps of 4.
3) Adaptive: This is the approach we present in this paper, as described in Section 4. The search for the optimal cache blocking parameter values terminates after collecting 20 points.

All of the search strategies are integrated with ATLAS version 3.4.1. Each search strategy optimizes performance by generating versions of code (mini-MMM) with the parameter values under test, compiling and executing them. Once the optimal transformation parameter values are found, a library is generated that uses the discovered values to multiply user-provided matrices. While it is plausible that optimal mini-MMM performance will translate into good performance when multiplying arbitrary matrices, this is not guaranteed. In this Section we generate libraries for multiplying double-precision floating point numbers. For each algorithm and each platform under test, the following measurements are made:

– The amount of time needed to find the optimal parameter values.
– Performance of mini-MMM code generated with the values found to be the optimal.
– Performance of the generated library on a wide range of matrix sizes.

5.2 Experimental Results

Table 2 lists the optimal cache block size chosen by each strategy. Table 4 presents the amount of time required for each search strategy to complete. The model performs simple calculations and, therefore, takes a negligible amount of time to complete. The adaptive search, while slower than the model, is three-four times faster than ATLAS search.

	Model	Adaptive	ATLAS
Sparc	88	60	68
SGI	62	170	64

Table 2. Selected Block Size (NB)

	Model	Adaptive	ATLAS
Sparc	376.66	851.04	832.63
SGI	499.81	553.15	505.4

Table 3. Mini-MMM Performance (in MFLOPs)

	Model	Adaptive	ATLAS
Sparc	0:00	3:12	8:59
SGI	0:00	14:02	59:00

Table 4. Time To Complete Search (in minutes)

The measured performance of each strategy appears in Table 3. As expected, the model is outperformed by ATLAS on these two platforms since the model, while extremely fast, is brittle due to its lack of feedback. ATLAS, on the other hand, requires an extensive sample size to achieve superior performance. The adaptive optimization outperforms both the model and ATLAS after collecting a small sample of points. Its performance gain over ATLAS is most significant on the SGI machine, where it chooses to tile for the L2 cache, not considered for optimization by ATLAS.

On the Sparc machine, while it appears that the adaptive strategy significantly outperforms the model, most of the performance gain is due to the optimal setting of the MU, NU, and $Latency$ parameters which are not considered in this work. The performance gain due to the adaptive search for the optimal NB value is only ∼10%. All the reported results for this machine are also affected by the `-native` flag that we are using in the `cc` compiler of the Sparc machine (Table 1) and that is automatically selected by ATLAS. The `-native` flag should direct the compiler to optimize the code for the current machine, but apparently the code generated when using this flag corresponds to that of an older architecture. If instead of `-native` we use the flag `-xarch=v9a` which corresponds to the architecture of the target Sparc machine, we found that the performance results of the code generated by Model were very similar to those in Table 3 for ATLAS or Adaptive.

Figure 5 shows the performance of the libraries generated using the parameters in Table 2 for each of the optimization algorithms under study. Figure 5 shows the performance of each library as the size of the matrices being multiplied increases from 100 × 100 to 3000 × 3000. The Figure demonstrates that there is a strong correlation between mini-MMM performance and performance of the final generated library, the metric that the end user of the system is interested in.

6 Conclusions and Related Work

Machine learning has been applied to construct adaptive compiler optimizers before. Cooper et. al., for example, use genetic algoithms to search through sequences of optimizing code transformations [9]. Using genetic algorithms (and other machine learning optimization algorithms) can be time-consuming in a large space of possible optimizations.

These techniques have also been extended to search for entire versions of algorithms, as opposed to just code transformations. Li et. al. [13] present a two-phase algorithm for optimizing sorting. The first (offline) phase performs a search to construct a mapping from the parameters of a sorted array (its data entropy and size) to the best-performing sorting algorithm. The second (online) phase uses that mapping to

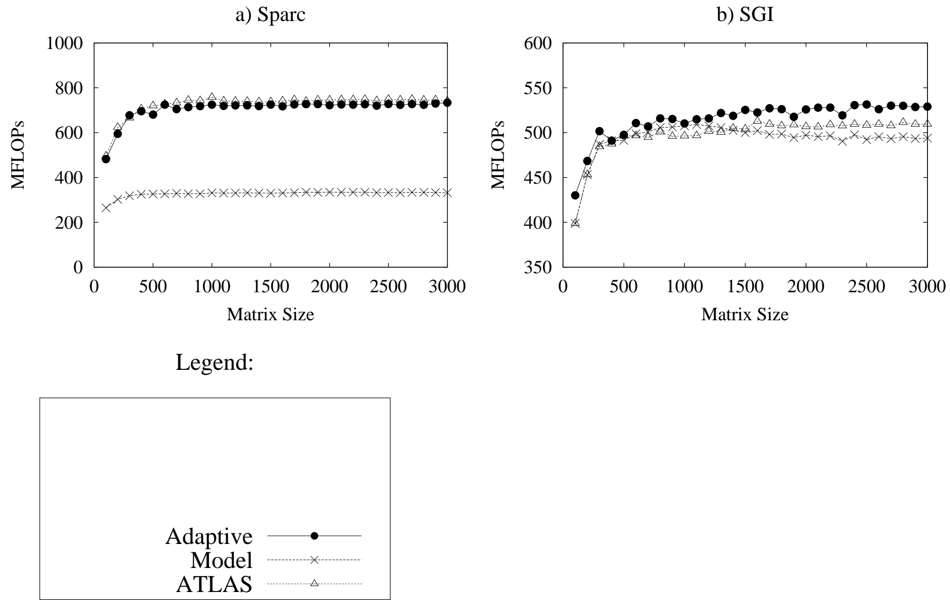

Fig. 5. Library Performance Comparison for ATLAS Search, Model, and Adaptive Search.

apply the best sorting algorithm to the given array at runtime. A similar framework was applied by Thomas et. al. to optimize parallel matrix multiplication [15].

An important feature which distinguishes our approach to searching is explicit integration of information from the analytic model to guide the search, thereby reducing its time. We believe that adaptive intelligent modeling represents a promising and important direction in code optimization. The defining motivation is to integrate all relevant information into a hybrid model which can both resolve optimization decisions and guide further information collection. The challenge is combining information from different sources that come in radically different forms. In this first proof of concept research, the forms include a general but approximate prior analytical model and empirical measurements of code samples taken directly on the system to be optimized. In our narrow but important test domain of mini-MMM optimization, our adaptive model is much more efficient than the empirical optimization approach. We believe our most significant research contribution is to open a new direction for code optimization. The principle of adaptive intelligent modeling is to actively seek out information that can be used as evidence for refining and restructuring itself so that the optimization decisions are always the best they can be. Our end goal is to expand the applicability of feedback-directed search in the online optimization setting, where both accuracy and speed are crucial.

References

1. ATLAS home page. [Online]. http://math-atlas.sourceforge.net/faq.html#NB80.

2. R. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
3. 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 Int. Conf. on Supercomputing*, Vienna, Austria, July 1997.
4. P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate Tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.
5. D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proc. of PLDI*, pages 53–65, 1990.
6. S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. *Proc. of 29th Hawaii International Conference on System Sciences*, 1996.
7. S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. of PLDI*. ACM Press, June 1995.
8. K. Cooper and T. Waterman. Investigating Adaptive Compilation Using the MIPSPro Compiler. In *Proc. the LACSI Symposium,*, Los Alamos Computer Science Institute, October 2003.
9. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1), 2002.
10. G. DeJong. Explanation-based learning. In A. Tucker, editor, *Computer Science Handbook*, pages 68.1 – 68.18. Chapman & Hall/CRC and ACM, 2nd edition, 2004.
11. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. *Proc. of PLDI*, 1997.
12. M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. *Proc. IEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 3:1381–1384, 1998.
13. X. Li, M. J. Garzaran, and D. A. Padua. A dynamically tuned sorting library. In *CGO*, pages 111–124, 2004.
14. C. P. Robert. *The Bayesian Choice*. Springer-Verlag, 1994.
15. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP) (to appear)*, 2005.
16. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. *Int. Symp. on CGO*, 2003.
17. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.
18. M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Third SIAM Conf. on Parallel Processing for Scientific Computing*, December 1987.
19. J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and a Compiler for DSP Algorithms. In *Proc. of PLDI*, pages 298–308, 2001.
20. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M.Garzaran, D. Padua, K. Pengali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. *Proc. of PLDI*, pages 63–76, 2003.
21. K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate a High Performance Blas? *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 23:358–386, February 2005.