Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy

Chun Chen, Jacqueline Chame and Mary Hall

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{chunchen,jchame,mhall}@isi.edu

Abstract

This paper describes an algorithm for simultaneously optimizing across multiple levels of the memory hierarchy for dense-matrix computations. Our approach combines compiler models and heuristics with guided empirical search to take advantage of their complementary strengths. The models and heuristics limit the search to a small number of candidate implementations, and the empirical results provide the most accurate information to the compiler to select among candidates and tune optimization parameter values. We have developed an initial implementation and applied this approach to two case studies, Matrix Multiply and Jacobi Relaxation. For Matrix Multiply, our results on two architectures, SGI R10000 and Sun UltraSparc IIe, outperform the native compiler, and either outperform or achieve comparable performance as the ATLAS self-tuning library and the hand-tuned vendor BLAS library. Jacobi results also substantially outperform the native compilers.

1 Introduction

We are falling far short of the peak performance on today's high end microprocessors. A key contributing factor is the growing gap between peak computation rate and limited memory bandwidth. Architectural and compiler solutions to bridge this performance gap have resulted in systems of enormous complexity. As a result, statically predicting the impact of individual compiler optimizations and the aggregate impact of a collection of optimizations is becoming increasingly difficult for both compiler and application developers.

A recent strategy to address this complexity and improve performance employs *empirical optimization*, to systematically evaluate a collection of automatically-generated *code variants* and *parameter values* [21, 2, 9, 25]. Code variants, in this context, are alternative, but equivalent and often closely related, implementations of the same computation. For a particular variant, there may additionally be parameters associated with code transformations, such as, for example, unroll factors, tile sizes, or prefetch distances. Rather than trying to predict performance properties through analysis, implementation variants are actually *executed on the target architecture* with representative input data sets across different parameter values so that performance can be measured and compared.

A recent paper provided a quantitative comparison between the empirical optimization approach used by the self-tuning linear algebra library ATLAS [21] on Matrix Multiply and a model-driven approach such as is standard in compilers [26, 27]. As methodology, they used the parameterized code variants included with ATLAS, and derived parameter values through models. The authors found using models to derive optimization parameters yielded results that were roughly comparable with that of empirical optimization, suggesting that, given the appropriate parameterized variants, compiler-derived models may be able to avoid or at least limit empirical search. Remaining questions posed by the authors of [27] are (1) how models can be used to guide empirical optimization; and, (2) how either can be made to close the considerable performance gap with handcoded implementations, such as the Basic Linear Algebra (BLAS) libraries.

This paper takes a step towards answering these questions. We believe that the best strategy should

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

employ the complementary strengths of compiler models and empirical optimization. For memory hierarchy optimization, finding a set of variants and parameters that result in high sustained performance is difficult because there are complex tradeoffs for the different memory hierarchy levels. In addition, the search space is difficult to model analytically since performance can vary dramatically with problem size and optimization parameters. Empirical results can greatly benefit the compiler in tuning the accuracy of its models and selecting the best among a set of candidate implementations. A purely empirical approach is not practical on general application code because the search space of possible variants and their parameters is prohibitively large. A compiler's understanding of the impact of code transformations on performance can be used to limit the search space and rule out the vast majority of inferior implementations.

These ideas are captured by the approach presented in this paper, which combines compiler models and heuristics with a model-driven empirical search. The approach has two distinct phases. In the first phase, compiler analysis derives a small set of parameterized variants of a computational kernel. Associated with each variant are constraints on parameter values, also derived by compiler models. In the second phase, an empirical search is performed to select among the variants derived in the first phase, and to choose appropriate parameter values within the constraints imposed by the first phase. In addition, the search phase performs those code transformations that depend upon parameter values (for example, scalar replacement and prefetching). Because the compiler intelligently prunes both the space of variants and their associated parameter values, we only need to search a tiny subset of the overall search space.

We implemented a preliminary version of this approach in the Stanford SUIF compiler infrastructure, and applied it to two case studies: Matrix Multiply (as in [26, 27]) and Jacobi, and measured performance on an SGI R10000 and a Sun UltraSparc IIe. Matrix Multiply yields stable performance across a wide range of array sizes and outperforms the native compilers and the hand-coded vendor BLAS libraries. In addition, it outperforms ATLAS on the SGI and achieves 98% of the ATLAS performance on the UltraSparc IIe. Jacobi, which we can only compare against the native compilers, also substantially outperforms the native compilers.

We attribute these compelling results to several unique features of our approach as compared to prior work in locality optimization, in addition to the restructuring of the compilation process as previously outlined. First, we select the level of the memory hierarchy for which we want to target the storage of individual arrays according to their reuse patterns. That is, we determine which arrays should go in registers, which in L1 cache, etc., and use this to guide optimization of each memory hierarchy level. Second, we use techniques that help smooth the search space and enhance the compiler's ability to prune it analytically. For example, we employ copying to eliminate conflict misses in cache. Third, we simultaneously optimize all levels of the memory hierarchy so as to avoid destroying the benefits of one optimization with another conflicting optimization. While at this point we have demonstrated the approach on only these two kernels, this work represents a step towards a general compiler algorithm for fully utilizing the memory hierarchy.

The remainder of the paper is organized as follows. The next section motivates the need for empirical search by examining the complex relationship between optimizations for different levels of the memory hierarchy. Section 3 describes the approach. The subsequent section presents experimental results comparing our approach to the native compilers for two kernels Matrix Multiply and Jacobi, and to ATLAS and BLAS for Matrix Multiply only. We discuss related work in Section 5, and then provide conclusions and future implications for this work.

2 Background

This section motivates the problem by discussing optimization choices for two dense-matrix kernels. Both kernels have regular access patterns and exhibit data reuse that optimizing compilers can easily recognize. Figure 1 shows the original Matrix Multiply (mm) and two optimized versions of the kernel, and Figure 2 illustrates Jacobi. The code transformations used in the examples are well-known optimizations that target the memory hierarchy of high-end systems: loop permutation to select a particular loop order, unroll-and-jam plus scalar replacement (also called register tiling) for exploiting data reuse in registers, loop tiling and copy optimization for cache, and software prefetching for hiding latencies.

Currently most compilers consider each of these optimizations somewhat in isolation, with parameters tuned according to the goals of the particular optimization. However, when considering these optimizations collectively, each kernel benefits most from quite different optimization choices.

To make this discussion concrete, Table 1 shows performance data for the optimized codes of Figure 1(b)(c) and Figure 2(b). Each of the first five rows corresponds

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

```
DO K = 1,N
  DO J = 1,N
    DO I = 1,N
      C[I,J] = C[I,J]+A[I,K]*B[K,J]
```

(a) Original Matrix Multiply

```
new P[TK,TJ]
DO KK = 1,N,TK
  DO JJ = 1,N,TJ
    copy B[KK..KK+TK-1,JJ..JJ+TJ-1] to P
    DO I = 1,N,UI
      DO J = JJ,min(JJ+TJ-1,N),UJ
        load C[I..I+UI-1,J..J+UJ-1]
                into registers
        DO K = KK,min(KK+TK-1,N)
          prefetch A's
          multiply A's and P's to registers
        store C[I..I+UI-1,J..J+UJ-1]
```

(b) Optimized Matrix Multiply (small arrays)

```
new P[TK,TJ]
new Q[TI,TK]
DO KK = 1,N,TK
  DO JJ = 1,N,TJ
    copy B[KK..KK+TK-1,JJ..JJ+TJ-1] to P
    DO II = 1,N,TI
      copy A[II..II+TI-1,KK..KK+TK-1] to Q
      DO J = JJ,min(JJ+TJ-1,N),UJ
        DO I = II,min(II+TI-1,N),UI
          load C[I..I+UI-1,J..J+UJ-1]
                  into registers
          DO K = KK,min(KK+TK-1,N)
            prefetch P's
            multiply Q's and P's to registers
          store C[I..I+UI-1,J..J+UJ-1]
```

(c) Optimized Matrix Multiply (large arrays)

Figure 1. Matrix Multilply

```
DO K = 2,N-1
  DO J = 2,N-1
    DO I = 2,N-1
      A[I,J,K] = c*(B[I-1,J,K]+B[I+1,J,K]+
                    B[I,J-1,K]+B[I,J+1,K]+
                    B[I,J,K-1]+B[I,J,K+1])
```

(a) Jacobi original code

```
DO JJ = 2,N-1,TJ
  DO K = 2,N-1,UK
    DO J = JJ,min(JJ+TJ-1,N-1),UJ
      load B[1..2,J..J+UJ-1,K..K+UK-1] into registers
      DO I = 2,N-1
        prefetch A's and B's
        load B[I+1,J..J+UJ-1,K..K+UK-1] into registers
        compute A[I,J..J+UJ-1,K..K+UK-1]
```

(b) Jacobi optimized code

Figure 2. Jacobi

to a version of Matrix Multiply, defined by a particular set of optimization parameters: tile sizes `TI`, `TJ`, `TK` and Pref listed in columns 2 to 5. The performance data was obtained by executing each version on an SGI Octane R10000 and using the performance monitoring interface PAPI [3] to access data collected by the hardware performance monitor. The same matrix size, which is larger than the second-level (L2) cache, was used in all experiments reported in the table.

For Matrix Multiply, $mm1$ has the lowest number of L1 misses, achieved by exploiting the reuse of `B(K,J)` across iterations of loop `I`. Tiling all three loops in $mm3$ reduces L2 misses dramatically but it also affects L1 misses. $mm5$ achieves the lowest number of cycles by balancing locality between the L1 and L2 caches (as $mm3$) even though it has the highest number of loads and none of the best L1 or L2 misses. In addition, prefetching in $mm5$ achieves an extra 3% reduction in total cycles with respect to $mm4$.

Jacobi has group-temporal reuse in all three loops and spatial reuse in the innermost loop. In $j1$, none of the loops are tiled. In $j3$ loops `J` and `K` are tiled targeting reuse at the L1 cache. In $j5$ loop `I` and `J` are tiled, reducing the L2 misses with respect to $j3$, but the L1 misses remain about the same and its performance is worse than that of $j3$. Versions $j2$, $j4$ and $j6$ correspond to $j1$, $j3$ and $j5$, respectively, with prefetching added, improving performance by about 20%. Although $j3$ performs better than $j5$, the corresponding prefetching versions show better performance for $j6$, which has less L2 and TLB misses.

3 Approach

Overall, the previous examples show that the best performance is obtained not from the code that has the minimum number of Loads, L1 misses, or TLB misses, but rather one that achieves a balance across all lev-

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

Version	TI	TJ	TK	Pref	Loads	L1 misses	L2 misses	TLB misses	Cycles
$mm1$	1	32	64	no	4,197,888,365	141,808,006	21,606,389	230,692	10,151,010,869
$mm2$	1	16	128	no	4,101,809,180	210,484,202	35,313,075	105,150,856	12,453,215,635
$mm3$	8	256	256	no	4,076,548,485	319,204,376	7,191,241	4,419,779	9,704,824,712
$mm4$	16	512	128	no	4,109,285,149	182,281,362	8,012,517	2,782,444	9,474,730,186
$mm5$	16	512	128	yes	5,119,308,380	188,163,516	8,044,926	2,781,031	9,175,706,120
$j1$	1	1	1	no	25,548,546	8,779,902	1,646,383	7,521	181,368,676
$j2$	1	1	1	yes	33,990,298	8,816,871	1,643,267	7,484	137,233,130
$j3$	1	16	8	no	27,955,826	6,099,362	1,315,500	18,338	155,449,279
$j4$	1	16	8	yes	40,800,890	7,616,195	1,318,205	18,610	124,725,626
$j5$	300	16	1	no	25,492,847	8,787,403	1,184,828	9,987	159,178,262
$j6$	300	16	1	yes	33,975,863	8,837,514	1,187,137	9,868	121,580,192

Table 1. Performance variation with optimization parameters

els of the memory hierarchy. Our approach to simultaneously optimizing across all levels of the memory hierarchy is organized into two separate phases. In the first phase, analysis derives a collection of parameterized variants of the computation. The second phase is a search among variants and parameter values, guided by models.

3.1 Deriving Parameterized Variants

The first phase derives code variants, using the algorithm in Figure 3, as follows: selects a loop order for each variant, the loops to which unroll-and-jam should be applied, the loops that should be tiled, and the data structures for which to consider copying.

The order of memory hierarchy levels (from registers as the lowest level to main memory as the highest) defines the order in which locality optimizations are evaluated. For each memory hierarchy level, a number of variants may be derived, and each such variant is processed when evaluating optimizations for the next level. Each level also associates with each variant constraints (C(level)) on parameter values (P(loop)) that will be used in the model-guided empirical search of the next section.

For each level of the memory hierarchy, from registers to the last cache level (and also considering TLB), the algorithm identifies a set of array references and a loop carrying temporal reuse for the references to that level. The goal is to keep the reused data in that memory hierarchy level in between iterations of the loop carrying the reuse. To achieve that, references associated with the register level should have a smaller reuse distance than references selected to be kept at higher levels, and loops carrying reuse for the register should be innermost with respect to loops carrying reuse for references associated with the L1 cache, and so on.

In the remainder of this subsection, we describe the compiler analysis and models that guide transformation and constrain the values of the parameters, and then discuss how transformations are applied.

3.1.1 Compiler Analysis and Models

The compiler uses analysis to determine the relationship between the data access patterns in the code and how the data will be mapped to levels of the memory hierarchy of a specific architecture. Architecture-specific information for each level of the memory hierarchy includes *Capacity(level)* and *Associativity(level)*, where the latter refers to set-associative caches. To determine whether a data structure will fill up a particular level of the memory hierarchy within a set *Tiles* of tiled loops, a compiler analysis derives *Footprint(Refs,loop,Tiles)*, based on the results of *reuse analysis*. Reuse analysis uses data dependence information to quantify the reuse of each array reference in the loop nest. These analyses and models are based on established approaches from the literature, so are only presented at a high level here.

Register reuse and footprint analysis. To quantify reuse at the register level, the analysis presented in [5] is used to examine the dependence graph to identify loops that carry register reuse, and quantify the reduction in memory accesses due to unroll-and-jam and scalar replacement as a function of unroll factors of loops selected for unroll-and-jam. Register reuse is distinguished from reuse at the cache level, described below, because only temporal reuse is exploited. Also, since registers must be explicitly named to exploit register reuse, the analysis must be more precise than cache reuse analysis (and thus more conservative).

The number of registers that can be used is bounded by the size of the register file. Since the number of registers required increases with the unroll fac-

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

```
Algorithm DeriveVariants
Inputs: Loops   /* set of loops in loop nest */
        Refs    /* set of array references in nest */
Output: V   /* set of parameterized variants */

V = InitialVariant {
   LoopOrder = ∅   /* ordered set of loops */
   UnrollLoops = ∅   /* set of loops to unroll */
   ControlLoops = ∅   /* tile controlling loops */
   P = ∅   /* optimization parameters */
   C = ∅   /* constraints on parameter values */
}
level = 0
while ((Loops ≠ ∅) &&
   (level < MEMORY_LEVEL)) {
   L = MostProfitableLoops(Loops, Refs)
   V_new = ∅
   foreach v ∈ V
      foreach l ∈ L
         V_new = V_new ∪ GenVariant(v, l, level)
   V = V_new
   level = level + 1
}
foreach v ∈ V {
   Order(ControlLoops)
   Push(ControlLoops, LoopOrder)
}
end Algorithm

GenVariant(v, l, level)
   Loops = Loops - l
   Retained = MostProfitableRefs(l, Refs)
   if (level == REGISTER_LEVEL) {
      foreach i ∈ Loops {
         < i^U, U_i > = Unroll(i)
         UnrollLoops = UnrollLoops + i^U
         Loops = Loops - i + i^U
         P(i^U) = U_i /* set loop parameter to unroll size */
      }
      Push(l, LoopOrder)
      C=C ∪ (Footprint(Retained,l, (U_i, i ∈ UnrollLoops))
         ≤ Capacity(RegisterFile))
      return (v)
   }
   else {
      foreach i ∈ Loops {
         /* select loops to tile */
         if (i ∉ ControlLoops) {
            < i^T, i^C, T_i > = Tile(i)
            Loops = Loops - i + i^C
            ControlLoops = ControlLoops + i^C
            Push(i^T, LoopOrder)
            P(i^T) = T_i /* set loop parameter to tile size */
         }
      }
      Push(l, LoopOrder)
      C=C ∪ (Footprint(Retained,l, (T_i, i ∈ LoopOrder))
         < f(Capacity(level),Associativity(level)))
      C=C ∪ (Footprint(Retained,l, (T_i, i ∈ LoopOrder)))
         < Capacity(TLB)
      /* return two variants, w/ and w/o copying for this level */
      v^copy = CreateCopyVariant(v,Retained,l,(T_i, i ∈ LoopOrder))
      /* avoid conflicts in copied data */
      C = C ∪ (mod(Size(CopyArrays),Capacity(level-1)) ≠ 0)
      return (v, v^copy)
   }
```

Figure 3. Algorithm to derive parameterized variants

tors, register reuse analysis derives constraints on unroll factors to prevent register pressure beyond *Capacity(REGISTER_LEVEL)*, as a function of the unroll factors. In practice, only a subset of the register file is available for use by replaced memory accesses, and an accurate static model is difficult to derive, particularly in the presence of aggressive optimizations for instruction-level parallelism. For this reason, the constraints provide an upper bound, but the search algorithm is used to detect the largest unroll factors that do not cause register pressure.

Cache reuse and footprint analysis. Reuse analysis for the cache levels uses the approach in [22], which distinguishes between the following types of reuse: *self-temporal* reuse happens when a reference reuses a same data item in distinct iterations, and *self-spatial* reuse happens when it reuses the same cache line. *Group-temporal* and *group-spatial* reuse happen when a group of references reuse the same data and cache line, respectively, in distinct iterations. Under this framework, loops carrying reuse are identified as reuse vectors in the loop iteration space. Finally, the amount of reuse of a reference r in loop l, R_l, is a function of the reuse type, the number of loop iterations N_l, and the cache line size (CLS). If r has temporal reuse in l, $R_l(r) = N_l$. If r has spatial reuse, $R_l(r) = CLS$, and if l carries no reuse, $R_l(r) = 1$. The computation of group-reuse can be found in [22].

At each cache level, the algorithm evaluates the amount of data referenced in a loop in terms of number of cache lines. The *cache footprint* of r in l is the set of cache lines used by r in l. A reference with spatial reuse in l accesses a new memory location at each iteration of l but the cache footprint increases only once every few iterations, as a function of the cache line size. If r has no reuse in l each memory access brings in a new cache line, leading to a larger cache footprint than those of references with reuse. As a heuristic, we set an upper bound on the constraints associated with a cache footprint of a tile of $(n-1)/n * Capacity$(level) for an n-way set associative cache ($n > 1$). This heuristic seeks to avoid conflict misses with other references that are not retained at this cache level.

TLB footprint. A TLB footprint reflects the number of memory pages accessed by references, and is evaluated at each cache level. For large data sets, TLB misses can have a profound negative impact on performance. The algorithm uses the TLB footprint to select loop order of outer loops (according to data layout order) and to constrain tile sizes such that TLB misses are avoided.

Profitability analysis. From reuse analysis, the algorithm determines the profitability of the loop trans-

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

formations being considered. The function *MostProfitableLoops(Loops, Refs)* returns the loop or set of loops that carries the most (unexploited) reuse. The set of loops is the algorithm's working set, and represents those loops that have not been selected to carry the reuse at an earlier level of the memory hierarchy. Initially only references that have not been mapped to an earlier level of the memory hierarchy are considered, but if no such references exist, then the algorithm may select a reference that has already been mapped. First, the algorithm determines which loop carries the most temporal reuse. If multiple loops have the same amount of temporal reuse, the algorithm considers spatial reuse, too. If there is still a tie, multiple loops are returned, resulting in the creation of multiple variants.

The function *MostProfitableReferences(l, Refs)* returns for loop l the set of references with the most temporal reuse carried by l not yet exploited at an earlier level of the memory hierarchy. Such references either have not been mapped to an earlier level of the memory hierarchy, or in the event when no such references remain, have been mapped to registers. For this reason, only a subset of references are considered.

3.1.2 Applying Code Transformations

Registers and Unroll-and-jam. The algorithm selects a loop that carries the most temporal reuse to be the innermost loop, to keep the reuse distance small. (In the event of a tie, multiple variants are created.) Unroll-and-jam is applied to all other loops, with the goal of unrolling the outer loops and jamming the copies of the unrolled iterations in the innermost loop to expose reuse. After the reuse is exposed, scalar replacement is applied to references that should be kept in registers. Since scalar replacement is more effective after the loops are unrolled, and the unroll factors are parameters of the variants, scalar replacement is deferred until the search phase.

Caches and Tiling/Copying. At subsequent levels of the memory hierarchy, the algorithm identifies the loop that carries the most *unexploited* reuse. That is, the algorithm targets data structures that were not placed in previous levels of the memory hierarchy. (Again, multiple variants may be considered.) The loop l carrying the reuse is moved to the outermost position, and the resulting inner loops are tiled so that the reused data remains in cache between iterations of l. For example, in *mm*, `C(I,J)` is associated with the register level, and `A(I,K)` is associated with the L1 cache.If all references are already associated with the register level but also have reuse in any of the outer loops, the outer loop l carrying the most reuse is selected for the outermost position and the other loops are tiled as needed. In Jacobi, where all three loops carry reuse for references to array `B`.

Once the loop(s) to be tiled are selected, the references mapped to this level of the memory hierarchy are considered a candidate for the copy optimization. In the variant that uses copying, the data tile with temporal reuse in the cache is copied to a temporary array to avoid cache conflicts. *CreateCopyVariant(v,Refs,l,Tiles)* associates with a loop l and its inner tiles *Tiles* the size and shape of the copy array. The algorithm also derives constraints to prevent conflict misses in the copy array in earlier levels of the memory hierarchy, ensuring that the stride for the copy array of each retained reference is not a multiple of the cache size for the previous level.

TLB and Tile Controlling Loops. Since the loop order and optimizations selected for a memory hierarchy level should not disturb other levels, the order of the tile controlling loops is evaluated. Since each iteration of a tile controlling loop accesses a new data tile, the tile controlling loop should access consecutive data tiles in the array layout in consecutive iterations, to avoid conflicts at the L2 cache and TLB.[1] This can be achieved by selecting the position of the tile controlling loop in the nest.

3.2 Model-Guided Empirical Search

In the second phase, a guided empirical search performs a series of experiments to derive the best parameter values for each of the variants. Once the parameters of a variant are selected, code transformations that depend on parameter values are performed – loop unrolling, scalar replacement, inserting temporary arrays for copying and prefetching. The result is a complete version of the code for each variant. After all versions are derived by searching for each variant, the version with the best performance is selected.

For each code variant, the search for tiling parameters of all levels of the memory hierarchy is performed first. Once the tiling parameters are selected, a search for prefetching parameters is performed on the variant, assuming the tile parameters selected. After the prefetching parameters are selected, the tiling parameters might need to be adjusted accordingly. The components of the search are described next.

Search tiling parameters and perform tiling-related transformations. The search for tiling pa-

[1] Assuming that consecutive array data are mapped to memory pages that do not collide in caches or TLB. To assure this is the case it may be necessary to touch the pages in the desired order during initialization, to induce the operating system's page-coloring algorithm to assign different colors to consecutive regions of the arrays [4].

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

rameters (for register and caches) is divided into stages such that the parameter values of each stage are searched independently, but the parameter values selected in one stage are carried over into the next stage. Each stage corresponds to one or more levels of the memory hierarchy. A given parameter may be associated to more than one level of memory hierarchy (for example, in matrix multiply the value of `TK` affects the tile sizes of both L1 and L2 caches). In this case the search of tiling parameters for both levels is performed in the same stage.

For each stage, a set of initial parameter values is determined by a simple heuristic as follows. The tile shape is proportional to the reuse weight of each dimension of the tiled iteration space. The tile footprint occupies the full capacity of a direct-mapped cache, and $(n-1)/n$ of the capacity of an n-way set associative cache ($n > 1$), as in Section 3.1. For the register level, the initial tile size is the size of the register file.

The search is performed by varying the *shape* and the *footprint size* of the tile. Starting from the initial values determined by the heuristics, the footprint size is kept constant, and a binary search on the tile shape is conducted by doubling one dimension and reducing the other dimension accordingly. Once a tile shape is selected for this footprint size, the footprint size is reduced by half and the search for the tile shape is performed again. This process is repeated until all neighboring points have worse performance than the current best tile shape/size. After that, a linear seach is done on each tiling parameter by simply increasing or decreasing a small amount which can be, for example, the maximum of register tile size and cache line size. To simplify the code generated, tiling parameter values that are multiples of any tile size or unroll factor previously selected are favored.

After all tiling parameters have been selected for a variant, code transformations for register tiling, that is, unroll-and-jam and scalar replacement, are performed as follows. For each register tiling parameter (unroll amount), the associated loop is unrolled, and scalar replacement is performed to replace array references with scalar temporaries that will be mapped to registers by the backend compiler. To simplify the code generated and reduce unnecessary loop overhead, the unroll amount in each dimension should favor a factor that evenly divides the loop bounds.

Search prefetching parameters. Using the tiling parameter values derived from the previous step, the impact of prefetching is explored. Prefetching is not always profitable since, although it hides memory latency, it consumes processor cycles and memory bandwidth, and it may pollute the cache. With the knowledge of which data structures target which level of memory hierarchy, prefetch instructions for each data structure are added one at a time. Starting from a prefetch distance of one iteration, the appropriate prefetch instructions are inserted into the code, which is then executed. If there is a performance gain by adding prefetching with a distance of one, further experiments are conducted to select the smallest prefetch distance that gives the overall best result. If there is no benefit, the added prefetch instructions are removed. Using the version of the code with previous prefetching decisions incorporated, the same process is repeated for prefetching another data structure until all data structures are evaluated.

Adjust tiling parameters after prefetch. As more data is prefetched, it reduces the amount of temporal reuse that can be exploited by tiling. Larger innermost loop bounds generally make prefetching more effective but may result in less optimal tiling parameters. Thus, the tiling parameters are adjusted to reflect this tradeoff. The search is done by running a series of experiments increasing the number of iterations of the innermost loop, with other tiling parameters adjusted to match, according to the constraints. The search stops when there is no further improvement.

4 Experimental Results

In this section we present an experimental evaluation of our approach on two architectures, an SGI R10000 and a Sun UltraSparc IIe (described in Table 2). We use two kernels, Matrix Multiply and Jacobi, and for each kernel we compare the performance of our code versions (denoted ECO) with those of the native compiler (Native) of each architecture. In addition, for Matrix Multiply we also present the performance of the ATLAS self-tuning library version of BLAS and the vendor's BLAS library.

We have developed a prototype implementation of our approach and integrated it in the Stanford SUIF compiler infrastructure. The implementation includes the analysis, code generation and search algorithm described in Section 3, and was used to automatically generate the ECO versions used in our experiments. We use SUIF as a source-to-source translator, with Fortran code as its input and a set of possible Fortran codes as its output. The optimized Fortran codes generated by our SUIF implementation are then compiled by the native compiler. Much of the analyses and code transformations used in our implementation were already available in SUIF (except copy and prefetch), so the main challenge in the implementation was integration of analyses and code transformations, eliminating

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

Architecture	Clock Rate	Registers	L1 cache	L2 cache	TLB
SGI R10000	195MHZ	32 floating-point	32 KB 2-way data	1MB 2-way unified	64
UltraSparc IIe	500 MHZ	32 floating-point	16KB direct data	256KB 4-way unified	64

Table 2. Comparison of two systems

undesired functionality, and generating and searching a collection of optimized codes.

Table 3 shows the compiler and optimization flags used for each architecture, as well as the ATLAS and BLAS versions of Matrix Multiply. For ATLAS, we use architectural defaults to present its best-case performance. Architectural defaults include the best compiler and flags to use on a platform, and possible manual guidance on code generation and parameter values. For ECO, we turned off all of the optimizations that are implemented in our system, since our SUIF implementation performs locality optimizations at the source level.

For Matrix Multiply, we consider a range of problem sizes from 100 to 3500; larger sizes would exceed the memory limit of our SGI. To simplify code generation and reduce the number of experiments, we evaluate one out of four problem sizes in the range. Similarly for Jacobi, the problem sizes are from 40 to 270, and we measure one out of two problem sizes in the range.

4.1 Matrix Multiply

Table 4 shows two variants of Matrix Multiply considered by our implementation for the SGI. Variant $v1$ (which corresponds to Figure 1(b)) is considered for small array sizes. In $v1$ loop `K` is the innermost loop, and unroll-and-jam is applied to `I` and `J` with parameters `UI` and `UJ`. Loop `I` is associated with the L1 cache, and `J` and `K` are tiled to exploit the reuse of `B(K,J)` at the L1. In addition, $v1$ includes copying the data tile of `B` a temporary array, and prefetches for `A` in the innermost loop. Variant $v2$, which corresponds to Figure 1(c), has a different loop ordering and additional tiling to exploit the reuse at the L2. Loops `I` and `K` are tiled for reuse of array `A(I,K)` and loop `J` is tiled for reuse of array `B(K,J)` at the L2. Both variants have TLB footprints smaller than the number of TLB entries of the SGI, so there are no additional constraints. Although our earlier manual experiments (not shown) determined that $v1$ performs better for small problem sizes, our preliminary SUIF implementation selected variant $v2$ with `UI=UJ=4`, `TI=16`,`TJ=512`,`TK=128` for all array sizes.

On the SGI, the ECO version shows stable behavior across the full range of problem sizes, as shown in Figure 4(a). Its performance ranges between 302 and 342 with an average of 333 MFLOPS, that is, 85% of the theoretical peak of 390 MFLOPS. The Native version has far more fluctuation in performance, ranging from 84 to 343 with an average of 308 MFLOPS. It appears to suffer from severe conflict misses for some matrix sizes because the SGI compiler does not apply copying. In addition, the performance of Native trails off for larger problem sizes due to very high TLB misses. ATLAS produces far more stable results than the SGI compiler, ranging from 246 to 316 with an average performance of 308 MFLOPS. The fluctuation shown in the graph for small problem sizes is because ATLAS does not apply copy in this range (it does use copy for larger sizes). PAPI results show that ATLAS has about 1.4 times L1 and L2 cache misses compared to those of ECO, but about three fourth load instructions and much less TLB misses. The performance of the hand-tuned Vendor BLAS is very close to that of ECO, ranging from 118 to 346 with and average of 334 MFLOPS, but it still has serious performance limitations for certain problem sizes. Specifically, Vendor BLAS has severe conflict misses at problem size 2048. For some larger problem sizes, the reason for the performance fluctuation is not very clear to us. We speculate that the page coloring algorithm fails to allocate non-conflicting pages when array sizes are close to the memory capacity.

We performed a similar comparison on the Sun UltraSparc IIe as shown in Figure 4(b). The ECO version used has parameters `UI=4`,`UJ=2`, `TI=16`, `TJ=96`,`TK=128`. The average performance of the Native version on the UltraSparc IIe is only 60 MFLOPS, far below the other three versions. ATLAS performs well in this architecture, ranging from 387 to 543 with an average of 517 MFLOPS. The performance of ECO ranges from 406 to 530 with an average of 506 MFLOPS. Again, the performance of Vendor BLAS is close to that of ECO, ranging from 368 to 512 with an average of 494 MFLOPS.

4.2 Jacobi

For Jacobi our approach generates variants with different loop orders, since all loops carry temporal reuse. For each loop order, unroll-and-jam is applied to the

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

Arch	Code Version	Compiler	Flags and Building Information
SGI	ECO	MIPSpro 7.3.0	-O2 -mips4 -r10000 -64 -OPT:swp=ON:roundoff=3:IEEE_arithmetic=3
	ATLAS	MIPSpro 7.3.0	ATLAS 3.7.8 using architectural defaults
	Vendor BLAS	–	SCSL 1.4.1.2
	Native	MIPSpro 7.3.0	-O3 -mips4 -r10000 -64 -OPT:roundoff=3:IEEE_arithmetic=3
SUN	ECO	Sun Workshop 6.1	-dalign -xO5 -xarch=v8plusa -xchip=ultra2e -xprefetch=explicit
	ATLAS	Forte Developer 7.5.4	ATLAS 3.6.0 using architectural defaults
	Vendor BLAS	–	SunPerf 3.0.0
	Native	Sun Workshop 6.1	-dalign -xO5 -xarch=v8plusa -xchip =ultra2e

Table 3. Compiler, Optimization Flags and BLAS versions

Variant	Level	Loop	Transf	Param	Constraints
$v1$	Reg	K	Unroll-and-jam I and J, Prefetch A	UI, UJ	UI · UJ $\leq$ 32
	L1	I	Tile J and K, Copy B	TJ, TK	TJ · TK $\leq$ 2048
	L2	J	–	–	–
$v2$	Reg	K	Unroll-and-jam I and J, Prefetch Copy of B	UI, UJ	UI · UJ $\leq$ 32
	L1	J	Tile I and K, Copy A	TI, TK	TI · TK $\leq$ 2048
	L2	I	Tile J and K, Copy B	TJ, TK	TJ · TK $\leq$ 65536

Table 4. Code variants considered for Matrix Multiply on the SGI

two outer loops to exploit reuse in registers. Since data from three dimensions of the array are accessed on each iteration, the unroll and tiling parameters must be such that a 3D tile of data is kept on each level of memory.

Variants with tiling for both L1 and L2 caches are pruned, as they would suffer cache and TLB conflicts for large arrays. In addition, taking the TLB behavior into account results in pruning more variants. For variants with L1 tiling only, where the tile size is relatively small, the order of the tile controlling loops should follow the data layout in memory to avoid extra TLB misses. For variants with tiling for the L2, the tile size is relatively large, and so the iterations within the tile should follow the data layout in memory to avoid extra TLB misses within the tile.

The Jacobi code for the SGI is shown in Figure 2(b) with parameters `UJ=UK=2`,`TJ=16` for all problem sizes. A comparison with the native SGI compiler is shown in Figure 5(a). The performance of Jacobi is substantially lower than Matrix Multiply, as it is inherently memory-bandwidth limited on the SGI. A theoretical upper bound based on the best-case memory access behavior is 150 MFLOPS. The SGI compiler does not apply copying or padding and therefore at some points the performance of Native is extremely low due to conflict misses, ranging from 4 to 79 with an average of 61 MFLOPS. Our compiler determines that copying has too much overhead to be profitable, but as a consequence, the ECO version also suffers from conflict misses and results in performance fluctuation of between 4 to 87 with an average performance of 73 MFLOPS. Manual experiments show that array padding can be used to stabilize this behavior.

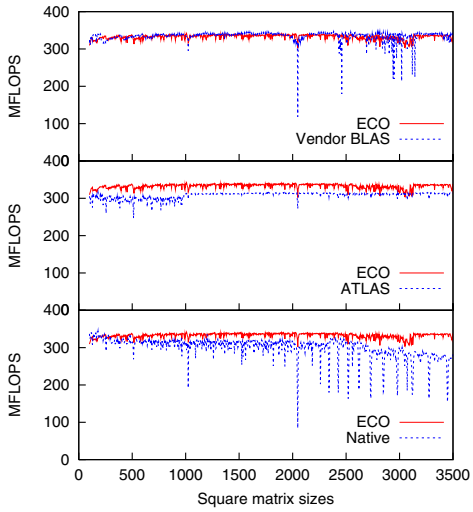
Figure 5(b) shows the results on the Sun UltraSparc IIe. The ECO version corresponds to the code in Figure 2(b) with parameters `UJ=UK=2`, `TJ=16`. The performance of Native ranges from 35 to 71 with an average of 47 MFLOPS, while the performance of ECO ranges from 38 to 63 with an average of 55 MFLOPS.

4.3 Cost of Search

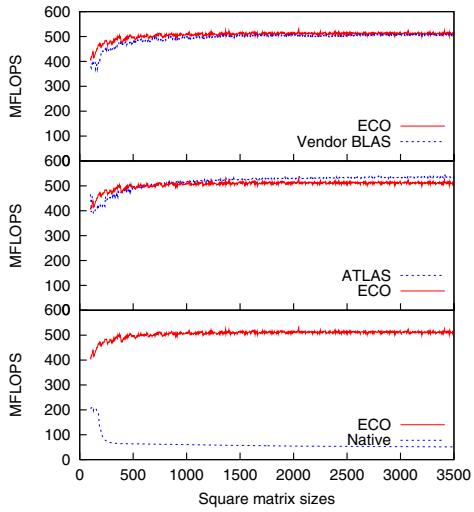
The search algorithm is the most preliminary part of the approach. As the implementation matures and we consider a richer class of application codes, we will gain significant insights to both speed up the search and improve the result. Nevertheless, to put this approach into context with the other techniques we have compared against in this section, we provide some measurements of the cost of empirical search.

Our current implementation searched 60 alternative implementations of Matrix Multiply on the SGI, which required roughly 8 minutes. On the Sun, it considered 44 points, which required 6 minutes. The Jacobi search was 94 points in 3 minutes on the SGI and 148 points in 5 minutes on the Sun. As compared to standard compile time, which was at most one or two seconds on both the SGI and Sun, this represents a fairly significant increase in cost, however it provides far better results that are comparable to the performance of hand-tuned codes on both architectures.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

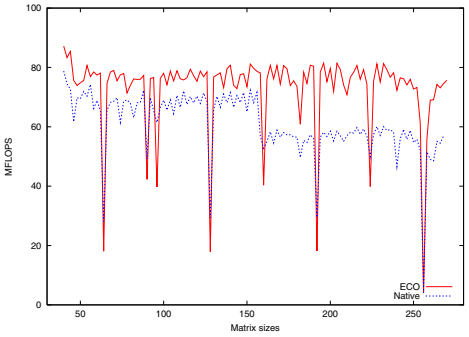


(a) Matrix Multiply comparison on SGI R10K.

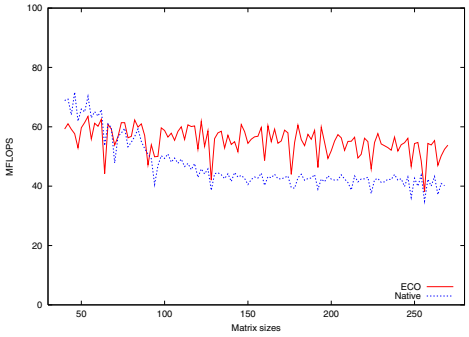


(b) Matrix Multiply comparison on Sun UltraSparc IIe.

Figure 4. Matrix Multiply Performance



(a) Jacobi comparison on SGI R10K.



(b) Jacobi comparison on Sun UltraSparc IIe.

Figure 5. Jacobi Performance

For Matrix Multiply, we can also compare against ATLAS search time for just the Matrix Multiply function. (This does not include the cost of a separate microbenchmarking search to determine architecture-specific properties, such as instruction-set features and useable register and cache capacity.) For a meaningful comparison with our compiler, we consider the ATLAS search time without the use of architectural defaults, which would have bypassed portions of the search. The ATLAS search for a Matrix Multiply implementation requires 35 minutes on the SGI and 14 minutes on the UltraSparc IIe. Thus our search is 2-4 times faster.

We should also point out that the derivation of the hand-tuned vendor BLAS libraries can be considered a manual empirical search, which we assume requires on the order of days of a programmer's time to try alternative versions and arrive at a highly tuned result.

5 Related Work

There has been extensive research on improving the cache performance of scientific applications, most of it targeting loop nests with regular data access patterns [5, 23]. Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses. More recent research also addresses conflict misses, which are important to the performance of tiled loop nests. Some of the research targeting conflict misses propose algorithms for selecting tile sizes according to the problem and cache sizes [12, 7, 16]. Others use data padding [1] or combine it with tiling [16] to prevent cache conflicts. Copying data tiles to contiguous memory locations at run time [19] is yet another technique used in combination with tiling to eliminate conflict misses. In [14], block

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

data layout is analyzed and the resulting constraints on block sizes can be used to limit the ATLAS search space. Recent research proposes precise, although more complex, models of cache misses [10, 6].

There are several on-going research projects in empirical optimization of scientific libraries, ATLAS [21], PHiPAC [2] and domain-specific libraries FFTW [9], SPIRAL [25]. PHiPAC and ATLAS both generate high performance matrix-matrix multiply by searching a large space of parameter values and executing actual resulting code on target machine. FFTW uses a combination of static models and empirical techniques to optimize FFTs. SPIRAL generates optimized digital signal processing (DSP) libraries by searching a large space of implementation choices and evaluating their performance empirically.

There are also research projects that use static models to evaluate the trade-offs between different optimization goals, such as optimizing for multiple levels of the memory hierarchy [24, 13]. Recent work on iterative compilation [11] combines static models and empirical search to reduce the search space of optimizations. This method is more restrictive than ours as it considers only tiling and unrolling.

A variety of AI search techniques, such as simulated annealing [15], hill climbing [8] and genetic algorithms and machine learning [25, 17, 20, 18] have shown promise in dramatically improving optimization results; at the same time, the enormous cost of these searches can be prohibitive since they incorporate little if any domain knowledge to limit the search space. We anticipate the kind of domain knowledge used in our approach could be effectively combined with such heuristic search techniques.

6 Conclusion and Future Work

This paper describes an approach to simultaneous optimization across several levels of the memory hierarchy for dense-matrix computations. The search space of possible transformations and transformation parameters is prohibitively large, and difficult to model accurately with purely static information. Thus, the overall strategy combines compiler models and heuristics with guided empirical optimization. The models and heuristics limit the search space to a small subset of candidate implementations, and the empirical results provide accurate information to the compiler to select among candidates and tune optimization parameter values.

We presented extensive experimental results for two case studies, Matrix Multiply and Jacobi, to demonstrate the promise of this approach. The results for these two programs are compelling. For Matrix Multiply, our approach yields stable performance across a wide range of problem sizes, on average within 88% of the theoretical peak, and always outperforms the SGI compiler, the ATLAS self-tuning library, and produces results comparable to the hand-coded BLAS library for the SGI. For the Sun UltraSparc IIe we also outperform the native compiler and the vendor's BLAS, and achieve 98% of the ATLAS performance on average. Similar performance gain is shown for Jacobi over the native compilers.

Acknowledgements. The authors wish to thank Clint Whaley for his assistance in obtaining ATLAS results. This research has been supported by the National Science Foundation under award ACI-0204040.

References

[1] D. F. Bacon, J. Chow, R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. of the Conference of the Center for Advanced Studies on Collaborative Research*, Nov. 1994.

[2] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proc. of the International Conference on Supercomputing*, June 1997.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, Aug. 2000.

[4] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[5] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, Nov. 1994.

[6] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2001.

[7] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. of the Conference on Programming Language Design and Implementation*, June 1995.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

[9] M. Frigo. A fast Fourier transform compiler. In *Proc. of the Conference on Programming Language Design and Implementation*, May 1999.

[10] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[11] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16(2–3):247–270, 2004.

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

[13] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing*, Aug. 1997.

[14] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[15] G. Pike and P. N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proc. of Supercomputing '02*, Nov. 2002.

[16] G. Rivera and C. Tseng. Data transformations for eliminating conflict misses. In *Proc. of the Conference on Programming Language Design and Implementation*, June 1998.

[17] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. In *Proc. of Supercomputing '01*, Nov. 2001.

[18] M. Stephenson, S. Amarasinghe, M. Rinard, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2003.

[19] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proc. of Supercomputing '93*, Nov. 1993.

[20] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[21] 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, Jan. 2001.

[22] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.

[23] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the Conference on Programming Language Design and Implementation*, June 1991.

[24] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1996.

[25] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2001.

[26] 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 *Proc. of the Conference on Programming Language Design and Implementation*, June 2003.

[27] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), Feb. 2005.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE