

Profitable Loop Fusion and Tiling Using Model-driven Empirical Search*

Apan Qasem
Department of Computer Science
Rice University
Houston, TX 77005, USA
qasem@cs.rice.edu

Ken Kennedy
Department of Computer Science
Rice University
Houston, TX 77005, USA
ken@cs.rice.edu

ABSTRACT

Loop fusion and tiling are both recognized as effective transformations for improving memory performance of scientific applications. However, because of their sensitivity to the underlying cache architecture and their interaction with each other it is difficult to determine a good heuristic for applying these transformations profitably across architectures. In this paper, we present a model-guided empirical tuning strategy for profitable application of loop fusion and tiling. Our strategy consists of a detailed cost model that characterizes the interaction between the two transformations at different levels of the memory hierarchy. The novelty of our approach is in exposing key architectural parameters within the model for automatic tuning through empirical search. Preliminary experiments with a set of applications on four different platforms show that our strategy achieves significant performance improvement over fully optimized code generated by state-of-the-art commercial compilers. The time spent in searching for the best parameters is considerably less than with other search strategies.

Categories and Subject Descriptors

C.1 [Processor Architectures]: General; D.3.4 [Programming Languages]: Processors—*compiler optimization*

General Terms

Design, Experimentation, Performance

*This material is based on work supported by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and 12783-001-05 49 from the Los Alamos National Laboratory. This work was also supported in part by the Rice Terascale Cluster funded by NSF under Grant EIA-0216467, Intel, and HP.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'06 June 28-30, Cairns, Queensland, Australia
Copyright 2006 ACM 1-59593-282-8/06/0006 ...\$5.00.

Keywords

Empirical tuning, loop fusion, tiling, memory hierarchy

1. INTRODUCTION

Loop fusion and tiling are both recognized as effective transformations for improving memory hierarchy performance of scientific applications. Tiling has been widely used in commercial compilers for some time; loop fusion, on the other hand, is gaining increased importance because of the increased usage of array assignments in languages like Fortran 95. However, because of their sensitivity to the underlying cache architecture it is difficult to determine a good heuristic for applying these transformations profitably across a range of platforms. Often, fusion and tiling will have different effects on different levels of memory. In some cases, improved cache performance comes at a cost of increased register pressure, while in other situations, reduced TLB misses might result in lost locality for some level of cache. Thus, in the absence of detailed architectural information, compilers are forced to resort to heuristics that favor one level of memory hierarchy over another. This of course implies a compromise in overall application performance.

The problem of how best to use loop fusion and tiling becomes even more difficult when we try to apply the two transformations in concert. Fusion and tiling interact with each other in complex ways. Because of this, the effectiveness of one transformation is often strongly influenced by decisions made in applying the other transformation. As evidence of this interaction, we present in Figs. 1 and 2, results from running a set of benchmarks with both fusion and tiling. We evaluated the effect of these two transformations on both the L2 cache and the TLB on an SGI R12K machine. The experimental data in Figs. 1 and 2 suggest considerable interaction between fusion and tiling. In some instances, this interaction has a positive impact on memory performance (e.g. `vpenta`), while in other cases, a negative interaction results in performance loss (e.g. `advect3d`). These experimental results emphasize the need for considering the interaction between loop fusion and tiling at different levels of the memory hierarchy. Applying these transformations in isolation is likely to result in less than desired performance.

Accounting for fusion-tiling interaction through analytic modeling is not enough to ensure high performance, however. Because of the tremendous complexity of modern architectures even the most detailed cost model is unlikely to determine the best optimization parameters across all ar-

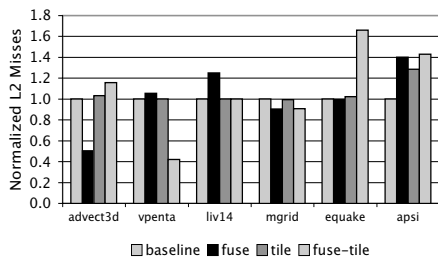


Figure 1: Effects of fusion and tiling on L2 Cache Misses

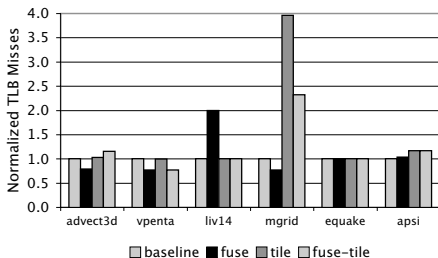


Figure 2: Effects of fusion and tiling on TLB Misses

architectures. To address this issue, many strategies for empirical tuning have been proposed [22, 4, 9, 20]. Although the empirical approach has been quite successful in generating highly-tuned domain specific libraries, its application in tuning general scientific programs has been limited. The chief obstacle in this regard is the prohibitively large search space that needs to be explored to find optimal transformation parameters. For example, Zhao et. al. [28] show that the search space for fusing n statements into m loops without any reordering can be as large as $\binom{n-1}{m-1}$. Clearly, exploring such a large space is infeasible for a general-purpose compiler. Recent papers advocate model-guided tuning as a means of pruning this enormous search space [26, 11, 2]. In this paper, we propose a new approach to pruning the optimization search space. Instead of exploring the search space of parameterized transformations we identify key architectural parameters that affect the profitability of loop fusion and tiling and search for the best values of those parameters. Our strategy includes a cost model based on reuse analysis that considers the effects of the two transformations on multiple levels of the memory hierarchy and characterizes the interaction between them. We present an algorithm that uses this cost model to make both fusion and tiling decisions simultaneously. Finally, we show how the model can be parameterized to expose architectural features for empirical tuning.

2. RELATED WORK

Both fusion and tiling have been studied extensively in the literature [24, 8, 3, 15, 10, 6, 13, 21]. However, relatively few papers have addressed the issue of combining these transformations with other transformations. In particular, the combined application of loop fusion and tiling with empirical search has not been studied previously.

Song et. al. [18] present a model that combines loop fusion and array contraction to reduce memory bandwidth requirements. Although they apply conditions to check for excessive register pressure and cache capacity, they do not address the issue of conflict misses. Wolf et. al. [23] describe a strategy that combines several loop transformations including fusion and tiling. Although, they look at a larger class of transformations their model does not capture all of the interactions between loop fusion and tiling. In their model, tiling decisions are made after the optimal loop structure has been determined through fusion and distribution. Moreover, their cost model is based solely on static estimators. No mechanism is provided for empirically tuning the model for different architectures.

A number of empirically-tuned library-generators have been successful in delivering high performance for a range of architectures. ATLAS [22] produces highly optimized BLAS routines by probing the underlying hardware for platform-specific information and using a global search to find the best transformation parameters, searching for these parameters one transformation at a time. Other empirically-tuned libraries, such as SPIRAL [25], FFTW [7] and PhiPAC [1], are all known to deliver high performance across a range of platforms. The main distinction between our approach and these empirically tuned libraries is that we do not attempt to exploit properties of any specific domain. Our goal is to develop an empirical tuning strategy for general scientific applications.

There has been some work in using empirical techniques for general purpose compilers. Cooper et. al. [4] use genetic algorithms to find the best sequence of compiler phases. The OSE compiler [20] uses static models available in Intel’s high-level optimizer to prune the optimization search space. Kulkarni et. al. [12] describe efficient ways of reducing the running time of the search algorithms. Our approach is different from theirs, in that we look at numerical parameters for program transformations which is inherently different from the search space of transformation sequences.

Fursin et. al. [9] use empirical tuning to select the best unroll, blocking and padding factors. They use a variety of search techniques in exploring the optimization search space. However, their work does not attempt to prune the search space using analytical models. Kisuki et. al. [11] use cache models to reduce the number of program evaluations for tuning unroll and tiling factors. Yotov et. al. [26] show that analytic modeling alone can deliver performance that is comparable to that of ATLAS. Chen et. al. [2] use model-guided tuning for a set of transformations including tiling. Our approach to tuning is distinct from previous work in that we aim to tune architectural parameters integrated in our cost model rather than the space of transformation parameters.

3. AN EXAMPLE

In this section, we use an example code to explain some of the interactions between loop fusion and tiling. The code in Fig 3(a) has two loop nests L_A and L_B . In L_A , values of $a()$ are used to compute values of $b()$ and in L_B , values of $b()$ and $d()$ are used to compute $c()$. We observe several types of reuse in this code. There is cross-loop reuse of $b(i, j)$ from L_A to L_B , some inner-loop reuse of $d(j)$ in L_B and also some outer-loop reuse of $a()$ at reference $a(i, j-1)$ in L_A . We now look at the trade-offs in applying fusion and tiling

to this code in order. (i.e. fusion before tiling and vice versa).

Fusion first: If we fuse L_A and L_B as shown in Fig 3(b), references to the same locations in $b()$ will be located within the same iteration of the innermost loop. This can potentially lead to saved loads of $b()$. However, in the fused loop nest we will access roughly twice as much data as compared to each of the unfused loop nests. This implies that the reuse distance for the outer-loop-carried reuse in $a(i, j-1)$ will increase in the fused nest, leading to cache misses on every iteration of the outer loop. Moreover, since we access more arrays, we also increase the chances of conflict misses. A good heuristic for loop fusion is likely to consider these negative effects on cache use and refrain from fusing the two loops, sacrificing the saved loads of $b()$.

Of course, some of the negative effects of loop fusion can be ameliorated by tiling the fused loop nest. Tiling the inner loop will reduce the reuse distance for $a(i, j-1)$ and ensure that reused blocks of $a()$ remain in cache during each iteration of the outer loop. The potential for conflict misses can also be reduced by picking tile sizes that make the working set significantly smaller than the cache size. Thus, if we do not consider tiling when making our fusion decision this code is likely to suffer some performance loss.

Tiling first: A good tiling heuristic will be able to pick a tile size for L_A to exploit the outer-loop reuse of $a()$. However, after fusion, the working set of the fused nest will increase and hence, the original tile size will no longer be effective. Thus, we would need to readjust the tile size to fit the new larger working set in cache. This however, does not solve the entire problem. We notice that tiling L_B will reduce some of the inner-loop reuse of $d()$. The amount of lost reuse will increase for smaller tile sizes. Hence, we need to pick a tile size that is large enough to minimize the loss of inner-loop reuse and at the same time small enough to exploit the reuse in the outer loop. If we cannot find such a tile then we need to reconsider our decision to fuse the two loops.

4. COST MODEL

In this section, we present a cost model that captures the complex interactions between loop fusion and tiling. Based on this model, we present an algorithm that performs the task of applying the two transformations together.

4.1 Program Model

In our framework, a program is a collection of statements, each enclosed by one or more loops. Loops are perfectly nested and loop bounds are affine expressions of loop iterators. All array references are considered to be *uniformly generated*. Without loss of generality, we assume that arrays are stored in column-major order. For the reuse distance analysis we assume loop bounds are known at compile time. Although this is not a realistic assumption for many applications, making such an assumption is not a problem in general. The issue of unknown loop bounds can be handled in several different ways. One approach is to simply assume loop bounds are large enough so that we get no reuse at any of the outer levels.

4.2 Quantifying Reuse for Fusion and Tiling

To apply fusion and tiling effectively, we first need a mechanism to quantify the amount of reuse available in the loop

nests. Reuse analysis has been a widely used tool for quantifying loop nest locality [24, 14, 19, 2]. However, most of the approaches in the literature have one of two limitations that make them ill-suited for our model. Most reuse-based analyses do not consider reuse across loop nests. Quantifying this inter-loop reuse is essential for multi-loop transformations such as fusion. Also, reuse at multiple levels of the memory hierarchy is not addressed by most of these approaches. As we noted in the example in Fig. 3, fusion and tiling can have conflicting effects on different levels of memory. Hence, it is necessary to develop a model that captures the effect of transformations on the entire memory hierarchy.

4.2.1 Capturing Inter-loop Reuse

Inter-loop reuse can be captured in a dependence graph through the use of loop-crossing dependence edges. A loop-crossing dependence is defined as follows:

DEFINITION 1. Let L_1 and L_2 be two fusible loop nests where reference r_1 accesses location M in some iteration i in L_1 and reference r_2 accesses location M' in some iteration j in L_2 . Then there is a *loop-crossing dependence* from r_1 to r_2 if $M = M'$.

We start with the dependence graph for single loop nests. Then, for each pair of adjacent loop nests we add loop-crossing dependence edges between the two dependence graphs. Once, we have identified all inter-loop reuse, we combine our model with the reuse model for single loop nests. In the single loop nest model, temporal reuse is classified as being either *self-temporal* or *group-temporal*¹. In addition, reuse can also be classified by the level of the loop that carries the reuse (*inner* and *outer* for a two-level loop). Thus, combining inter-loop reuse with the single loop nest model gives rise to *nine* different classes of reuse as shown below:

(i)	{loop-crossing}
(ii)	{self, inner}
(iii)	{group, inner}
(iv)	{self, outer}
(v)	{group, outer}
(vi)	{loop-crossing, self, inner}
(vii)	{loop-crossing, self, outer}
(viii)	{loop-crossing, group, inner}
(ix)	{loop-crossing, group, outer}

As we will see in Section 4.3, fusion and tiling can have different effects on each of these nine types of reuse. Hence, we need to consider each type of reuse individually.

4.2.2 Hierarchical Reuse Analysis

We use hierarchical reuse analysis [16] to determine the effects of fusion and tiling on different levels of the memory hierarchy. The key idea in this analysis is to associate with each reused reference, a value that denotes the level of memory where reuse is exploited. This is called the *reuse level* of a reference and is defined formally as follows:

DEFINITION 2. Let L_i refer to the memory at level i . Then the reuse level of a reference r involved in temporal reuse is

¹We currently do not explicitly handle spatial reuse. Spatial locality is exploited by selecting tile sizes that are multiples of the cache line size.

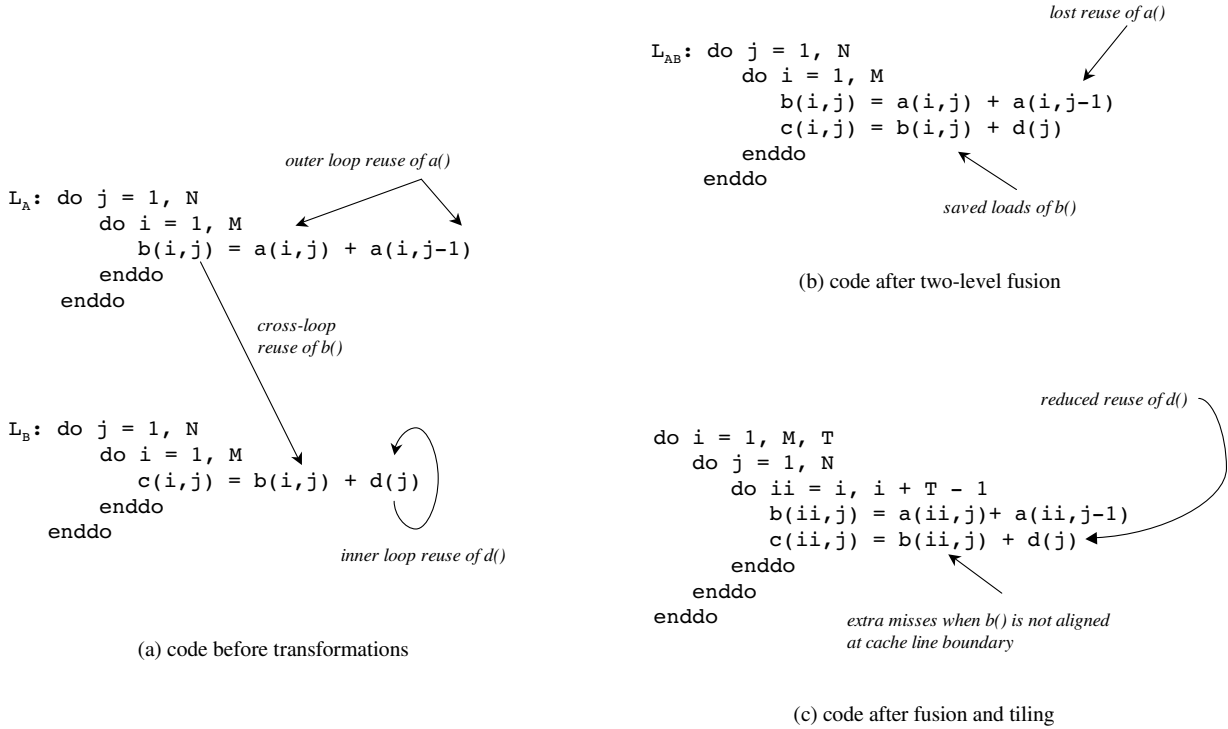


Figure 3: Effects of fusion and tiling on reuse

the smallest k such that

$$\text{ReuseDistance}(r) \leq \text{Capacity}(L_k)$$

where $\text{Capacity}(L_k)$ is the number of lines that can simultaneously occupy the cache at level k .

The analysis in Section 4.3 is used to determine the *reuse level* of each reused reference. Overall profitability is then determined simply by comparing the *reuse levels* of references both before and after applying the transformations.

4.3 Modeling Fusion-Tiling Interaction

We first look at the effects of fusion and tiling on reuse separately. We then present the analysis that determines the combined effect of these two transformations. For simplicity, we limit the discussion to a two-level loop nest. We use the following notation:

l_1, l_2	loop nests considered for fusion
j, i	outer and inner loop in each loop nest, respectively
ii	tilted loop
N, M	loop bounds of outer and inner loops, respectively
$FP_{(i,l_1)}$	footprint of one iteration of loop i in loop nest l_1
T	tile size
B	line size of target cache

4.3.1 Fusion Effects

Loop-crossing reuse is our prime target for improving locality with loop fusion. After fusion, the source and the sink of a loop-crossing dependence are executed in the same iteration of the innermost loop of the fused nest. Even if

the loop-crossing reuse is involved in a carried reuse, after fusion, the most recent use of the sink always occurs within the same iteration of the innermost loop². This leads us to our first theorem:

THEOREM 1. *Fusion decreases the reuse distance of any loop-crossing dependence and any loop-crossing dependence that is also involved in a carried dependence.*

Loop fusion has a negative impact on any carried reuse that is not involved in a loop-crossing reuse. After fusion the footprint of the inner loop in the fused nest increases to $FP_{(i,l_1)} + FP_{(i,l_2)}$. Consequently, the footprint for the outer loop increases to $FP_{(j,l_1)} + FP_{(j,l_2)}$. Hence, we have:

THEOREM 2. *Reuse distance of any outer or inner-loop-carried dependence that is not involved in a loop-crossing dependence increases as a consequence of fusion.*

4.3.2 Tiling Effects

When we tile the inner loop with respect to the outer loop, reuse distance of any outer-loop-carried dependence is reduced from $FP_{(j,l_1)}$ to $T * FP\{ii, l_1\}$. Since the source and the sink of the loop-crossing reuse reside in two separate loop nests tiling will not impact their reuse distance. This leads to the following theorem:

THEOREM 3. *Tiling decreases the reuse distance for any reuse carried by the outer loop. Tiling has no impact on references that are only involved in a loop-crossing reuse.*

²The situation when a loop-crossing dependence becomes carried after fusion is special and requires more complex analysis. Due to space constraints we do not include that analysis here.

By splitting the iterations, tiling can adversely affect the locality in the inner loop. In an untilted loop nest, the most recent use of an inner-loop dependence with a threshold of d always occurs d iterations before the current iteration of the inner loop. On the other hand, in a tiled loop nest, every time we begin working on a new tile, the sink of the reuse is separated from its source by $N \times T$ iterations of the inner loop. Thus, we can state the following about the effects of tiling on inner loop reuse:

THEOREM 4. *Tiling increases the reuse distance for some instances of inner-loop-carried reuse. The number of instances for which the reuse distance is increased is M/T .*

4.3.3 Combined Fusion and Tiling Effects

From Theorems 2 and 3 we know that fusion and tiling affect the reuse distance of outer-loop-carried reuse in opposing directions. However, the increased reuse distance due to fusion can generally be compensated by choosing a smaller tile size. This will ensure that the reuse distance for any outer-loop-carried reuse is small enough for the reused value to still be in cache. The only place where this approach will not work is if we are forced to pick a tile size that is smaller than one cache line. The above observation leads us to the following theorem:

THEOREM 5. *If $Capacity(L) > T > B$ then applying both fusion and tiling will result in saved memory operations for any reuse carried by the outer loop.*

It follows directly from Theorems 1 and 3 that fusion and tiling applied together, is beneficial for references involved only in loop-crossing reuse or a loop-crossing reuse and an outer-loop reuse. When a loop-crossing dependence is involved in an inner-loop-carried dependence, we get the full benefits from the loop-crossing reuse due to fusion. Moreover, the negative effects of tiling on the reuse distance is no longer observed. After fusion, the sink of any loop-crossing reference always gets the value from its most recent use, which is in the same iteration of the innermost loop. Therefore, we can state the following about loop-crossing dependences:

THEOREM 6. *Applying fusion and tiling together results in a net gain in memory operation cost for any reference that is involved in a loop-crossing dependence.*

Both fusion and tiling have a negative impact on references that are involved only in inner-loop-carried dependences. Thus, when applying tiling and fusion together we will suffer additional misses on such references.

THEOREM 7. *Applying fusion and tiling together will increase the reuse distance for some instances of the sink of any inner loop reuse which is not involved in loop-crossing reuse. The number of additional misses incurred is inversely related to the tile size.*

Based on the theorems presented in this section, we can characterize the interaction between fusion and tiling as follows: generally, fusion and tiling will interact favorably to reduce the number of cache misses for both loop-crossing and outer-loop-carried reuse. However, by increasing the footprint of the inner loop, fusion imposes a constraint on the tile size to be selected. Meeting this constraint may imply loss of locality in the inner loops. In cases where this loss exceeds the gains from fusion, it will be more profitable to tile the loops separately.

4.4 Putting It Together

The combined cost model for fusion and tiling can be incorporated into a constraint-based weighted pair-wise fusion algorithm [5]. In this algorithm, fusion is formulated as a graph clustering problem in which vertices represent loops in the program and weights represent the amount of benefit obtained by fusing the endpoints. Pair-wise greedy fusion considers for fusion only *prime edges* - edges whose endpoints are joined by no other path. At each step, the algorithm picks for fusion the heaviest *prime edge* in the graph whose endpoints can be fused without exceeding the resource constraints. After each fusion operation, weights are recomputed and the graph is updated with new successor, predecessor and prime edge information. At termination the algorithm produces the best set of fused loops according to the greedy heuristic.

The main consideration for using our cost model in a weighted fusion algorithm is incorporating tiling decisions within the algorithm. To do this we need to have additional information associated with each edge and each loop node. In the combined fuse-tile algorithm, we tag each edge with a parameter T that represents the tile size of the resulting fused loop nest. This tag is updated each time we update the edge weights in the graph. $T < 1$ implies no tiling for the fused loop nest whereas a negative weight implies the two loops should be left unfused. In such cases, the parameter T is a pair that represents the individual tiling sizes of the two loops. Tiling of all such loops is performed on a separate pass at the end of the greedy phase.

5. EMPIRICAL SEARCH

As mentioned in earlier discussion, our search strategy does not aim to tune parameters for individual transformations. Instead our approach is to identify key architectural parameters that can affect the profitability of a set of transformations. For each such parameter P , we introduce a tolerance term T and construct a function that computes the *effective size* of P based on the given value of T .

$$P' = EffectiveSize(T, P, \dots) \text{ s.t. } P' \leq P$$

The rationale for using the concept of *effective size* is that generally the amount of resources that can be exploited by a program is some fraction of the resources that is actually available on the target machine. For example, the effective cache size at a particular level is reduced by the possibility of conflict misses, as we shall see below. Hence, to apply resource-dependent transformations (i.e. tiling) profitably, we need to use analytical models that can estimate how much of a given resource is available to the program. However, the amount of resource available is determined by a host of factors. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays we access in the program and also the size of each of those arrays. A static model that attempts to capture all these parameters is unlikely to be totally accurate for all architectures. The goal of our tuning strategy is to correct for these inaccuracies in the analytical model.

Fig. 4 gives an abstract algorithm for our tuning strategy. For each tuning parameter in the search space we start off conservatively with a low tolerance term and increase the value of T at each subsequent iteration. For each tolerance

```

/* src is source of program to be tuned */
/* T is the set of tolerance values */
/* numDims is number of dimensions in the search space */

T ← InitTolerance()
curVariant ← DoCostAnalysis(src, T)
execTime ← CompileRunMeasure(curVariant)
bestTime ← execTime
bestVariant ← curVariant
for i = 1 to numDims do
  repeat
    /* increase tolerance and generate new program variant */
    repeat
      T ← IncTolerance(T, i)
      curVariant ← DoCostAnalysis(src, T)
      numInc++
    until curVariant ≠ bestVariant
    execTime ← CompileRunMeasure(curVariant)
    /* keep track of best execution time and best program variant */
    if (execTime ≤ bestTime) then
      bestTime ← execTime
      bestVariant ← curVariant
    else
      T ← DecTolerance(T, i, numInc)
    end if
  until (execTime > bestTime)
  /* stop search in this dimension when performance no longer im-
  proves */
end for

```

Figure 4: Algorithm for empirically tuning fusion and tiling parameters

value, we apply our cost model to produce a new program variant. We stop the iterative process either when performance degrades or when we have reached the availability threshold of a particular resource. Thus, our search strategy is *sequential* and *orthogonal*. For n resources (e.g. cache levels), we have an n -dimensional search space where the size of each dimension is the range of tolerance values for a particular resource. For each dimension we perform a sequential search. When searching in a particular dimension we use reference values for all other dimensions.

We currently apply our search to two architectural parameters. We discuss the tolerance terms and feedback metrics for each of these parameters next.

(i) Effective Cache Capacity: Estimating the data cache capacity at different levels is a key consideration for most memory hierarchy transformations. For tiling, tile sizes are chosen such that the working set of the tiled loop fits into cache. On the other hand, decisions for outer loop fusion are made based on the criteria that the reuse distance for inter-loop reuse is reduced to the extent that it is less than the capacity of some cache level. We use a probabilistic model to compute the effective cache capacity for different levels of cache [16]. The model derives the probability of a cache line being evicted before it is reused based on the size and associativity of the cache and the reuse distance. We then introduce a tolerance term that expresses how high a

probability of a conflict miss we are willing to accept. This tolerance term is used to derive an upper bound on the reuse distance, which serves as the effective cache capacity. Based on this model we have,

$$\text{Effective Capacity}(L_k) = E(s_k, a_k, T)$$

where L_k is the cache at level k , s_k and a_k refer to the size and associativity of the cache and T is a tolerance term that expresses the probability of a conflict miss between two references for this cache level.

The feedback metric used in tuning this parameter is the number of misses at the corresponding cache level.

(ii) Effective Register Set: Our current model only considers tiling for cache. Hence, the effective register set parameter is tuned for optimizing fusion choices only. We use the following formula to compute the effective register set:

$$\text{Effective Registers} = \lceil T \times \text{Register Set Size} \rceil$$

where $0 \leq T \leq 1$

The feedback metric we use here is the number of loads, which serves as a good indicator for register spills.

6. EXPERIMENTAL RESULTS

We implemented our cost model and search algorithm in a performance-based empirical tuning framework [17]. In this section, we present an evaluation of our strategy using some preliminary experimental results. We divide the discussion into two parts. First, we evaluate the effectiveness of our tuning strategy by comparing performance results with commercial compilers on a variety of platforms. Next, we focus on the search itself and compare our strategy with *multi-dimensional direct search* - a search strategy known to be effective in finding good values for transformation parameters [17, 27].

6.1 Experimental Setup

We select four programs that exhibit opportunities for loop fusion and tiling: **advect3d**, an advection kernel for weather modeling, **erle**, a differential equation solver, **liv18**, a hydrodynamics kernel from Livermore loops, and **mgrid**, a multi-grid solver from SPEC 2000. We apply our cost model to each program and use a source-to-source code transformer to restructure the code with the desired optimization parameters. The transformed source is then compiled using the native compiler on the target platform. To avoid conflicts with the optimization strategies of the native compiler, transformed programs are compiled with fusion and tiling turned off. We present results of running experiments on four different platforms. The chosen platforms display wide variations in cache and TLB organization and hence, serve as a good basis for evaluating our tuning strategy. The memory configuration for the four platforms are presented in Table 1. In the discussion that follows we use the following terms to refer to the different optimization strategies:

baseline	no fusion or tiling
native	fully optimized by native compiler
model-based	tuning strategy described in this paper
direct	cost model + direct search on tile sizes

	MIPS
CPU	300 MHz R12000
L1 Cache	32 KB, 32 B/line, 2-way
L2 Cache	8 MB, 128 B/line 2-way
TLB	128 entries, 16 KB/p, Full

	Itanium
CPU	900 MHz Itanium2
L1 Cache	16 KB, 64 B/line, 4-way
L2 Cache	256 KB, 128 B/line, 6-way
L3 Cache	1.5 MB, 128 B/line, 8-way
TLB	128 entries, 16 KB/p, Full

	PowerPC
CPU	2.5 GHz G5
L1 Cache	32 KB, 128 B/line, 2-way
L2 Cache	512 KB, 128 B/line
TLB	1024 entries, 4KB/p, 4-way

	Pentium III
CPU	800 MHz PIII
L1 Cache	16 KB, 32 B/line, 4-way
L2 Cache	256 KB, 32 B/line, 8-way
TLB	8 entries, 4 MB/p, 4-way

Table 1: Platforms

6.2 Tuning Strategy Performance

6.2.1 MIPS

Performance results on the MIPS for the four applications are presented in Fig. 5. These results show that **model-based** is able to outperform both **baseline** and **native** on each application. The most significant improvement is observed for **liv18**. In this case, **model-based** decides to fuse all three loops all the way through. The MIPSPro compiler, by contrast, refrains from fusing the third loop nest. This may be due to alignment issues or because of a heuristic used in the compiler to account for register pressure. We note, that although **model-based** incurs some extra loads because of aggressive fusion it is able to compensate for the loss with reductions in L2 and TLB misses.

On **advect3d** and **erle**, **native** performs worse than **baseline**. The tile sizes chosen by the native compiler for **erle** cause conflicts in both the level two cache and the TLB. For TLB this conflict is quite severe causing almost 10 times as many misses as that of **baseline**. **model-based** is able to pick tile sizes good enough to improve L2 performance without causing conflicts in the TLB. For **advect3d**, **native** does an overly aggressive fusion which creates a large inner-loop body. This results in a high number of register spills and conflict misses in the L2 cache. **model-based** refrains from fusing all the loops because of the register pressure constraint.

model-based shows only marginal improvement over **native** on **mgrid**. Although, **model-based** improves locality in the two caches, it pays severe penalty in the TLB. For **mgrid**, the tile sizes chosen by **model-based** are very small. For all the outer loops the tile sizes chosen is the minimum allowed within the search space. To find out why our model exhibited such poor performance for TLB, we manually changed the lower bound for the outer tile size

and ran the code with smaller tile size values. This resulted in severe misses in the L1 and L2 caches (most likely due to lost spatial locality and high loop overhead). Thus, choosing tile sizes any smaller and reducing the working set further is unable to avoid the conflicts in the TLB for **mgrid**. This result suggests that tiling alone is not enough to exploit locality in this case. To fully exploit locality in **mgrid**, we need to explore data layout optimizations. We address this issue in the concluding section of this paper.

6.2.2 Itanium

Performance results on the Itanium are presented in Fig. 6. The best performance on the Itanium is observed for **erle**. In this case, **model-based** is able to reduce the number of misses for both levels of the cache and also the TLB. For **advect3d** the performance improvement is not as great as it was for the MIPS. In this case, the Intel compiler, like the MIPSPro, decides to fuse the loops all the way through. However, since the Itanium has a much larger register set, it is able to withstand some of the register pressure of the large inner-loop body.

For **liv18** the Intel compiler does not perform any fusion or tiling. **model-based** in this case, is able to improve locality at both the L3 cache and the TLB. For **mgrid** both **model-based** and **native** perform about the same. **model-based** does not suffer from TLB thrashing as it did on the MIPS. However, it is not able to exploit much locality at either of the cache levels. Most of the improvement we observe for **mgrid** is because of the reduced number of loads due to fusion.

6.2.3 PowerPC

Performance results on the PowerPC are presented in Fig. 7. We observe the most significant performance improvement on this platform. For **advect3d**, **model-based** is able to fuse all loops without a corresponding increase in register spills. This behavior can be attributed to the larger register set on the PowerPC. We also observe significant performance improvement for **erle** and **liv18**. For these two programs, **model-based** chooses to tile for the 32KB L1 cache and is able to find tile sizes that make the working sets small enough to fit in the cache. The PowerPC has relatively large L1 cache lines (16 words). In each case, our cost model chooses tile sizes that fully exploit the spatial locality on these larger cache lines. There is no difference in performance between **baseline** and **native** on this platform.

6.2.4 Pentium III

Fig. 8 shows the performance results on Pentium III. The main architectural feature we are able to exploit on this machine is the large TLB pages. The 4MB pages on the Pentium III allow us to fuse more loops without causing too many conflicts in the TLB. Moreover, we also have the freedom to explore larger tile sizes for the outer loops, which allows us to exploit more instances of outer-loop reuse. With **liv18**, for example, **model-based** chooses an outer tile size of 32. This tile size is small enough to avoid TLB conflicts, yet large enough to exploit a good amount of outer-loop reuse.

The mean performance results for all platforms is presented in Fig. 9. These results show that **model-based** is able to achieve significant performance improvement on all

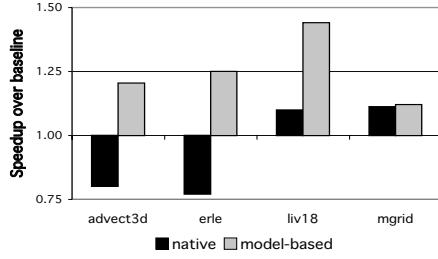


Figure 5: Performance improvement on MIPS

four platforms. In several instances, `model-based` achieves good performance, while the optimization strategies of the native compiler results in performance loss. We should note, however, that the effectiveness of our tuning strategy varies from one platform to another. The effectiveness of our strategy depends, to some degree, on how well we are able to model the underlying architecture. Thus, for more complex machines we observe less speedup. The performance is also influenced by the interaction of our strategy with the optimization strategies of the native compiler. Sometimes the optimizations used by the native compiler make fusion or tiling less effective, leading to reduced performance.

6.3 Comparison with Direct Search

As discussed earlier, our approach of tuning architectural parameters explores a much smaller search space than strategies that search for transformation parameters directly. We perform a set of experiments to determine how much performance is lost as a result of moving into the smaller search space. For these experiments, we replace our search strategy with a multi-dimensional direct search that explores the search space of possible tile sizes. To keep the comparison fair, direct search for tile sizes is conducted on program source that is already tuned for loop fusion.

Performance results from four applications on the MIPS are presented in Fig. 10. The results show that `model-based` is able to find values that are very close to the values found by `direct`. The performance gap is never more than 5%. On the other hand, in terms of tuning time we pay a high premium when we apply direct search. Fig. 11 shows that on average `direct` requires about four times as many program evaluations as `model-based`. In the context of empirical tuning, where number of program evaluations is the principal bottleneck, this savings in tuning time will be significant for any decent-sized application. In such cases, the savings in tuning cost will make the small sacrifice in performance worthwhile.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a model-driven approach of empirically tuning loop fusion and tiling parameters. The experimental results in Section 6 show that our analytical model is able to estimate the trade-offs between fusion and tiling with reasonable accuracy. By combining our static model with empirical search, we are able to adapt the transformed programs to achieve good performance on different platforms. Our approach of tuning architectural parameters results in a significant reduction in the size of the optimization search space, while incurring only a small performance

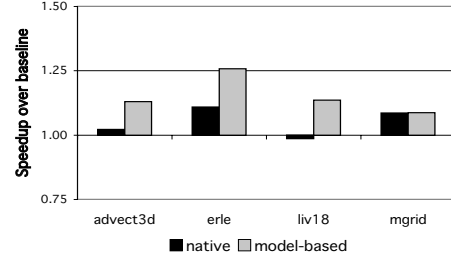


Figure 6: Performance improvement on Itanium

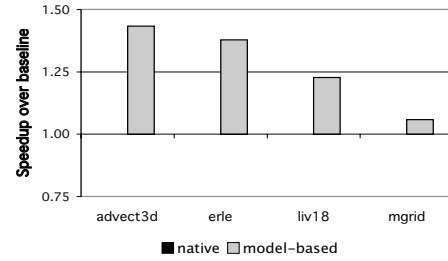


Figure 7: Performance improvement on PowerPC

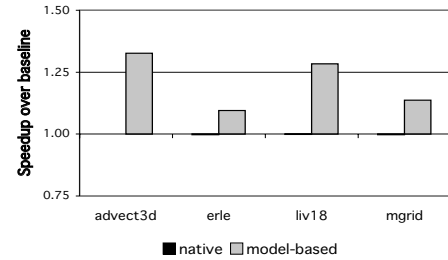


Figure 8: Performance improvement on Pentium III

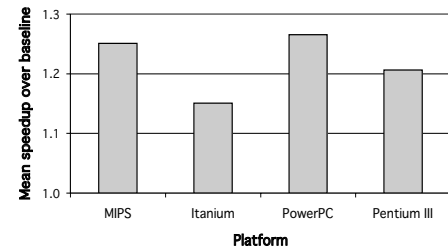


Figure 9: Mean performance across platforms

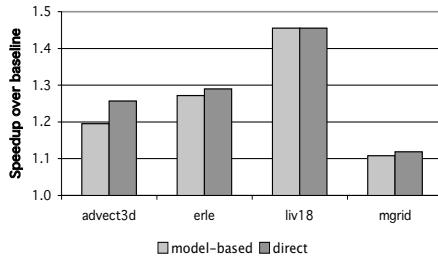


Figure 10: Performance comparison: model-based vs direct

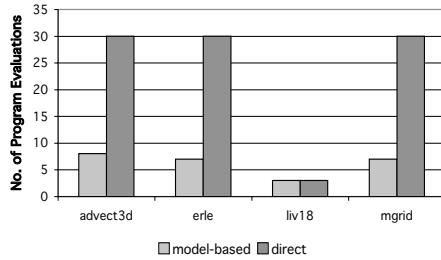


Figure 11: Tuning time comparison: model-based vs direct

penalty in the resulting code.

The experimental results in Section 6 also show that in one instance our strategy is unable to avoid conflicts in one level of the memory hierarchy. Although it does not result in overall performance loss, this issue needs to be considered more carefully. The issue of cache interference is intricately tied to how data is laid out for a program. Thus, transformations that only focus on locality of reference in loop nests are inherently limited in their ability to avoid conflicts in memory. For this reason, we need to look beyond loop fusion and tiling and explore tuning strategies for global data layout transformations. In the future, we plan to incorporate an analytical model for array padding that can be used with our empirical tuning strategy.

8. REFERENCES

- [1] J. Bilmes, K. Asanovic, C.-W. Chen, and J. Demmel. Optimizing matrix multiply using phipac: a portable high-performance ansi-c coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
- [2] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, San Jose, CA, 2005.
- [3] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [4] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.
- [5] C. Ding and K. Kennedy. Resource-constrained loop fusion. Technical report, Dept. of Computer Science, Rice University, Oct. 2000.
- [6] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *International Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001. (Best Paper Award.).
- [7] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
- [8] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.
- [9] G.G.Fursin, M.F.P.O'Boyle, and P.M.W.Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Fifteenth International Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.
- [10] K. Kennedy. Fast greedy weighted fusion. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, 2000.
- [11] P. 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*, 16:247–270, 2004.
- [12] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.
- [13] A. Lim and M. Lam. Cache optimizations with affine partitioning. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, Virginia, Mar. 2001.
- [14] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
- [15] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(5), 1998.
- [16] A. Qasem and K. Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In *Proceedings of the Eighteenth International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, Oct. 2005.
- [17] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2004.

- [18] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.
- [19] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, May 1994.
- [20] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Francisco, CA, 2003.
- [21] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors*, June 2003.
- [22] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
- [23] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on MicroArchitecture*, pages 274–286, 1996.
- [24] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
- [25] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP algorithms. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, 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 *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [27] H. You, K. Seymour, and J. Dongarra. An effective empirical search method for automatic software tuning. Technical report, University of Tennessee, Feb. 2005.
- [28] Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical report, Lawrence Livermore National Laboratory, Dec. 2005.

APPENDIX

A. MEMORY PERFORMANCE RESULTS

Here, we include memory performance results for the four programs.

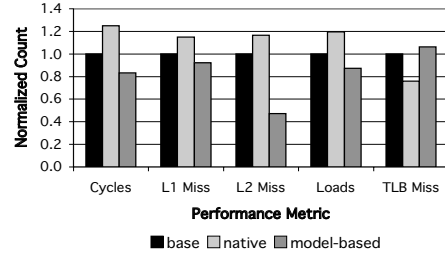


Figure 12: Memory performance of advect3d on MIPS

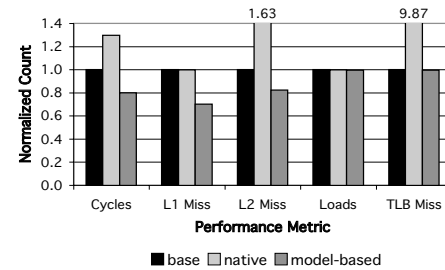


Figure 13: Memory performance of erle on MIPS

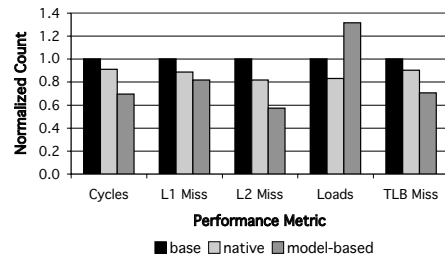


Figure 14: Memory performance of liv18 on MIPS

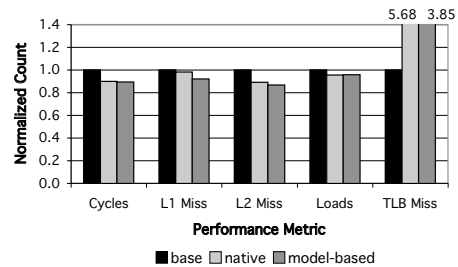


Figure 15: Memory performance of mgrid on MIPS