A Cache-conscious Profitability Model for Empirical Tuning of Loop Fusion

Apan Qasem Ken Kennedy

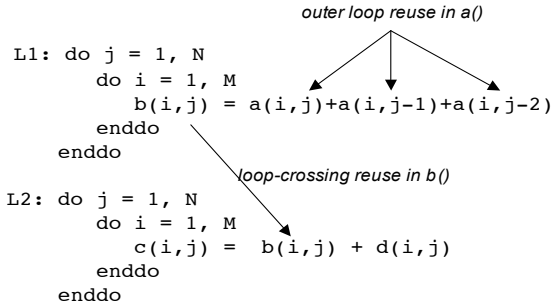
Department of Computer Science
Rice University
Houston, TX
{qasem,ken}@cs.rice.edu

Abstract. Loop fusion is recognized as an effective program transformation for improving memory hierarchy performance. However, unconstrained loop fusion can lead to poor performance because of increased register pressure and cache conflict misses. The complex interaction between different levels of the memory hierarchy with the input program makes it very difficult to always make the right choice in fusing loops. In this paper, we present a cache-conscious analytical model for profitable loop fusion to be used with a constrained weighted fusion algorithm. We then extend the model to show its effectiveness in the context of an empirical tuning framework. A preliminary evaluation of the model is presented using hand experiments on four applications.

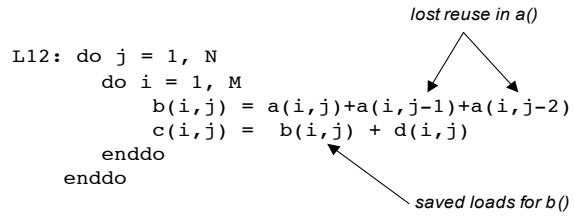
1 Introduction

Loop fusion is recognized as an effective program transformation for improving memory hierarchy performance of applications. Fusion improves data locality by merging loops that access the same data. Although fusion is a useful transformation it is not always profitable. Previous research has shown that unconstrained application of fusion may sometime lead to performance loss [4, 10].

Consider the code in Fig 1. In the first loop nest we compute values for array `b`. These same values are then used in the second loop nest. We can exploit this locality in array `b` by performing a two-level fusion operation. In the fused loop nest shown in Fig 1(b) the two references to array `b` are close enough to be put into a register. Thus as a result of fusion we can potentially save NM memory operations. However, there is also an outer loop reuse in array `a` for the references to `a(i,j-1)` and `a(i,j-2)` in loop nest L1 that we need to consider. In the unfused version the same memory locations in array `a` are touched in every iteration of the outer loop. In the fused version, although we do touch the same locations in array `a`, the amount of data that we bring into cache between reuses has increased. In the fused version, we will be accessing locations in arrays `b`, `c` and `d` before we get to the reused reference of `a`. If the intermediate data between reuses is larger than the cache capacity then we will incur 2NM cache misses due to the references to `a`. Moreover, by bringing in data from different arrays between reuses we also increase the likelihood of conflict misses. The occurrence



(a) code before fusion



(b) code after 2-level fusion

Fig. 1. Example of nonprofitable fusion

of conflict misses in the loop nest can be even more damaging to performance because it can lead to lost spatial locality in both arrays `c` and `d`. Thus for the code in Fig 1 fusion will not yield an overall profit. (Many readers will observe that these issues can be ameliorated by tiling the loop that results after fusion. Although we do not analyze the interaction of tiling with fusion in the body of this paper, we discuss the subject in the final section.)

Fusion can also degrade memory performance by increasing register pressure for the innermost loop. When fusing loops at the innermost level the register requirements may increase to the point where a large number of register spills occur. The cost of these spills may offset any benefits gained by improved locality in the fused loop. The possibility of exceeding the instruction cache capacity is also a concern when fusing loops with large instruction counts in the innermost loop bodies.

The problem of finding the optimal fusion solution has been shown to be NP-complete [2]. For large applications with many fusible loops finding a good fusion solution involves using good heuristics. In this paper, we present a strategy that combines an architecture sensitive cost model with empirical tuning to perform profitable loop fusion. Our cost model considers the size, associativity

and latency of various levels of the cache in determining if it is profitable to fuse a pair of loops. We incorporate this cost model into a constraint-based fusion algorithm. We formulate two constraints for the fusion algorithm to ensure that performance does not degrade as a result of increased pressure on system resources due to fusion. Finally, we use empirical tuning to tune a set of fusion parameters which cannot be measured accurately through static analysis.

In the sections that follow we discuss related work, present our analytical model, demonstrate how it can be used in an empirical tuning system, present a preliminary evaluation of the model and finally discuss our conclusions and future work.

2 Related Work

Fusion has been studied in the literature both as a tool for improving data locality and increasing the granularity of parallelism [8]. In this paper we look at fusion in the context of improving data locality only.

In its general form the task of finding the optimal fusion has been shown to be NP-complete [2]. Several algorithms have been described that use heuristics to find good fusion solutions in reasonable time. Lim and Lam use affine transformations to apply fusion [9]. Gao et. al. use a max-flow-min-cut algorithm to partition loop nests into fusible clusters [5]. Kennedy describes a fast-greedy weighted fusion algorithm that runs in polynomial time [7]. In our work we do not look at algorithms for performing loop fusion but rather focus on establishing suitable profitability constraints for legally fusible loops.

Many researchers have proposed models for performing loop fusion to improve memory performance. Ding and Kennedy have looked at reducing effective bandwidth through loop fusion [4]. Verdoolaege et. al. [14] describe a greedy fusion algorithm for incremental loop fusion at multiple levels. However, their locality models do not consider input dependences or the costs associated with cache misses. Song et. al. [13] present a model that combines loop fusion, loop alignment and array contraction. In their model, the primary profitability consideration is reducing bandwidth through reduced-sized arrays. Although they apply conditions to check for excessive register pressure and cache capacity they do not address the issue of conflict misses.

There are two main differences between our approach and the previous work done in this area. Firstly, unlike previous models our approach uses machine specific information (e.g. cache line size, latency) in combination with reuse distances in determining if fusion is profitable for a pair of loops. Secondly, we extend our model to be used in the context of an empirical framework. To our knowledge fusion has not been applied in this setting.

3 Profitability Model

3.1 Quantifying Reuse in Fusible Loops

Capturing inter-loop nest reuse: To determine if it is profitable to fuse a pair of loops we first need to compute the amount of reuse that is exploited as a result of fusion. Fusion improves locality by merging loops that access the same data. Thus any memory location that is accessed in the first loop nest and then re-accessed in the second loop nest is a candidate for potential reuse. This 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 $M = M'$.

To quantify reuse in fusible loops 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.

Pruning the dependence graph: The extended dependence graph described above is able to identify points of potential reuse in fusible loops. However, in cases where there are multiple inter loop dependences with overlapping thresholds the graph might overestimate the amount of reuse exploited by fusion. To account for such situations we need to *prune* the graph so that the sink of each loop-crossing dependence represents a potential savings in memory operations. We note that if there are multiple loop-crossing dependences emanating from the same source reference then all but one of the loop-crossing dependence edges can be eliminated. The edge that remains is the one that points to the sink reference that has no incoming dependence edge from within the loop nest. Similarly, if there are multiple loop-crossing dependences that have a single reference as their sink we can eliminate all but one of the edges. In this case, the edge that remains is the one that has a source with no dependence edges flowing into it from within the loop nest.

In addition to handling the loop-crossing dependences we also need to prune the dependence graph for each loop nest so that the pruned graph has at most one predecessor for each reference and that predecessor refers to the most recent use of the sink. This pruning is essential for our cost model which assumes one predecessor per sink in order to avoid double counting of cost on particular references. We adopt strategies described by Carr [1] to perform this pruning. The strategy involves eliminating all killed dependences from the graph and in cases of group temporal reuse keeping only those edges that have the smallest dependence threshold.

Hierarchical classification of reuse: Once, we have the pruned dependence graph we need to augment it to include information about reuse distances and memory hierarchy levels. The effects of fusion may not be beneficial across all levels of the memory hierarchy. Fusing a pair of loops may improve locality

at some level of cache but actually hurt locality at other levels. Hence, to improve overall memory performance we need to be able to quantify reuse that is exploited at each level of the memory hierarchy.

When considering multiple levels of the memory hierarchy, the reuse classification described in [15] is somewhat inadequate. We introduce a new classification of temporal reuse based on the level at which locality is exploited. We associate with each sink node in the dependence graph a value that expresses the level at which the reuse is exploited. This term is called the *reuse level* of a reference and we define this 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 the smallest k such that

$$ReuseDistance(r) \leq Capacity(L_k)$$

3.2 Accounting for Conflict Misses

Conflict misses can be a big concern for profitable fusion. When fusing loops we often bring accesses to a number of different arrays within the iterations of a single loop nest. If the array locations overlap in cache then we would have to pay the penalty of increased conflict misses. To account for conflict misses we extend the cache associativity model described by Mark and Hill in [6]. We compute 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.

Let,

r_1 and r_2 = references to the same cache line
m = reuse distance between r_1 and r_2
s = number of sets in cache
a = associativity

If we assume, each line from m is equally likely to be mapped to any of the sets then (this assumption is revisited in Section 5)

$$\begin{aligned} Pr[a\ lines\ landing\ in\ line\ occupied\ by\ r_1] &= Pr[conflict\ miss\ on\ r1] \\ &= \sum_{i=a}^{m} \binom{m}{i} \left[\frac{1}{s}\right]^i \left[\frac{s-1}{s}\right]^{m-i} \\ &= 1 - \sum_{i=0}^{a-1} \binom{m}{i} \left[\frac{1}{s}\right]^i \left[\frac{s-1}{s}\right]^{m-i} \end{aligned}$$

Now, we introduce a tolerance term T that expresses how high a probability of a conflict miss we are willing to accept. We then have,

$$T \geq Pr[conflict\ miss\ on\ r1] = 1 - \sum_{i=0}^{a-1} \binom{m}{i} \left[\frac{1}{s}\right]^i \left[\frac{s-1}{s}\right]^{m-i}$$

From this inequality we can derive an upper bound on m for a given value of T.

$$m \leq E(a, s, T)$$

Here, $E(a, s, T)$ is the maximum integral m such that $Pr[conflict\ miss\ on\ r1] \leq T$.

Now, given a tolerance term T and the size and associativity of a cache at level k we can express our formula for *effective cache capacity (ECC)* in the following manner:

$$ECC(L_k) = E(a_k, s_k, T) \tag{1}$$

where, s_k and a_k refer to the size and associativity of the cache at level k.

Based on this model of *effective cache capacity* we now have a new definition for the *reuse level* of a reference.

Definition 3 Let L_i refer to the memory at level i. Then the *reuse level* of a reference r involved in temporal reuse is the smallest k such that

$$ReuseDistance(r) \leq ECC(L_k)$$

3.3 Estimating Profitability

With reuse information and the heuristics for conflict miss in place we are now able to estimate the profitability of fusing a pair of loops. For each loop-crossing dependence in the pruned graph we want to determine how many memory operations are saved as a result of placing the source and the sink within the same iteration of the fused loop.

Let,

l_1 and l_2 = candidate loops for fusion that have the same nesting depth
D = set of loop-crossing true and input dependences between l_1 and l_2
C = set of dependences carried by either l_1 or l_2
$ReuseLevel_{\{pre,post\}}(d)$ = reuse level for d before and after fusion
L_k = cache at the k^{th} level where $0 \leq k \leq L$, L_0 refers to the register level and L_L refers to main memory
$cost(L_k)$ = cost of a miss access to L_k

Then for each $d \in D$ we assign a weight w based on the following condition:

if $ReuseLevel_{pre}(d) > ReuseLevel_{post}(d)$
then

$$w(d) = \sum_{i=ReuseLevel_{post}(d)}^{ReuseLevel_{pre}(d)-1} cost(L_i)$$

else

$$w(d) = 0$$

Then total weight is just

$$\sum_{d \in D} w(d)$$

Computing the number of memory operations saved from loop-crossing dependences is not enough to determine if fusion is profitable. As illustrated in the example in Fig 1 in some cases fusion may destroy locality within loop nests. When fusing two loops the reuse distance of any carried dependence increases if that reuse is also not involved in a loop-crossing dependence. We need to account for all such cases where fusion might lead to loss of potential reuse.

For each $c \in C$ we need to compute the cost based on the following condition:

if $ReuseLevel_{pre}(c) < ReuseLevel_{post}(c)$
then

$$w(c) = \sum_{i=ReuseLevel_{pre}(c)}^{ReuseLevel_{post}(c)-1} cost(L_i)$$

else

$$w(c) = 0$$

Then total cost is

$$\sum_{c \in C} w(c)$$

Hence, the final formula for computing the weight between two fusible loops is:

$$W(l_1 l_2) = \sum_{d \in D} w(d) - \sum_{c \in C} w(c)$$

3.4 Resource Constraints

A detailed analysis of the savings in memory operations does not guarantee beneficial fusion. There are several factors that can affect fusion that are not captured by the model we presented for computing weights. Most of these factors have to do with the resource requirements of the fused loop. If the requirements for a particular resource is higher than what is available to the program then the benefits of improved locality through fusion may not be realized. In this section, we establish a set of constraints that need to be considered by a *constrained weighted fusion* algorithm [3].

(i) **Register Pressure:** If the number of required registers for the fused loop body is more than what is available then we have to pay the price for spill costs. To account for register pressure we enforce the following constraint:

$$Register\ Pressure(Loop_{fused}) \leq Register\ Set\ Size$$

We use the methods presented in [1] to estimate register pressure in a loop body. Information about the number of registers available to the program is collected before compilation.

(ii) **Instruction Cache Capacity:** If the number of instructions in the fused body is large enough to blow out of the instruction cache then we have to pay the penalty of fetching those instructions from memory. Again, this phenomenon should be considered when fusing two loops.

$$Instructions(Loop_{fused}) \leq Capacity(I\text{-}Cache)$$

It should be noted that although data cache capacity is another critical resource requirement for a program we do not include it as a constraint here. When using our cost model with a weighted fusion algorithm the weights of the individual edges account for the data cache miss costs. For this reason we do not consider the total data requirements of the fused loop as a separate constraint.

3.5 Using the Model with a Greedy Fusion Algorithm

The fusion model and the resource constraints that we formulated can be incorporated into a constrained weighted fusion algorithm. We choose the pair-wise greedy fusion algorithm as described by Kennedy and Ding in [3]. In this algorithm fusion is formulated as a graph clustering problem in which the vertices represent loops in the program and the weights represent the amount of benefit obtained by fusing the endpoints. At each step the algorithm picks the heaviest *prime edge* in the graph and fuses its endpoints. After each fusion operation weights are recomputed and the graph is updated with new successor, predecessor and prime edge information.

The chief issue that needs to be considered in incorporating our model with the greedy algorithm is the cost associated with recomputing the weights at every step. Since, we perform a detailed analysis in calculating the benefits of fusing two loops we need to annotate the graph with more information to make the reweighing process more efficient.

We construct the pruned dependence graph with reuse information as described previously. We then group the references within each loop nest and label the subgraphs as *supernodes*. We compute the weights between each pair of fusible loops according to the procedure described in section 3.3 We connect each pair of *supernodes* using these weights. Hence, each pair of supernodes has only one node connecting them that represents the net gain from fusing the two loops.

Now, the pair-wise fusion algorithm can proceed normally on the *supernodes* and the edges between them. After fusing a pair of loops, edge weights between *supernodes* have to be updated and the loop-crossing dependence edges adjusted. For this step, we need to examine each loop-crossing dependence coming into and out of the fused loop nest. The edges within the *supernodes* representing outer loop reuse also have to be examined. We note however, that the number of edges in both cases is bounded above by the number of arrays in the loop. Hence, the complexity of a reweighing operation will be $O(A)$ where A is the number of arrays in the program. Having the complexity of the update operation bounded by the number of arrays ensures that the fast greedy algorithm will be

able to maintain its original asymptotic time bound inspite of the more detailed profitability analysis.

3.6 Parameterizing the Model

Even the most detailed analytical models may not produce the optimal fusion solution. Profitable fusion depends on a number of architectural features and it is often difficult to determine *a priori* how these features will interact with the fusion choices. For example, using the model presented in 3.3 we may be able to make a prediction about the possibility of conflict misses but we cannot say how good our prediction is until the program is actually run on the target machine. Similar uncertainties remain in measuring register pressure and cache footprints. Our approach to dealing with these uncertainties is the use of empirical tuning. In this section, we show how the analytical model that we have presented in this paper can be parameterized and used in an empirical tuning framework.

The basic idea behind our algorithm for empirically tuning fusion parameters is this: we identify system resources (e.g. available registers) that impose constraints on fusion choices. We then introduce a *tolerance factor* T which determines how much of a given resource we can use in each tuning step. The relationship between the *tolerance factor* for a given resource R and the *available resource* R' can be expressed as

$$R' = f(T, R) \; s.t. \; R' \leq R$$

For example, in the instance of tuning the register pressure parameter, the function $f()$ is a multiplication of the tolerance factor T with the register set size followed by a ceiling operation on the product. We start off conservatively with a low tolerance factor and increase the value of T at each subsequent iteration. We stop the iterative process either when performance degrades or when we have reached the availability threshold of a particular resource.

Since, at each step we only *relax* some fusion constraint, it is easy to show that the set of fused loops grows monotonically during the tuning process. Because of this property we chose a search strategy that is *sequential* and *orthogonal*. For n resources we have an *n-dimensional* search space where the size of each dimension is the range of tolerance factors 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.

Our current search model includes three resources: data cache capacity, instruction cache capacity and register pressure. Although, these three resources are somewhat similar they interact with fusion choices in different ways and hence constitute individual search dimensions. We discuss the tolerance factors and feedback parameters for each of these resources next.

Effective Cache Capacity: We compute the *effective cache capacity* using Eq. 1. Intuitively, Eq. 1 tells us what fraction of the cache we can use so that there is $T\%$ probability of a conflict miss between two accesses to the same memory location. So, in this case we have

$$\textit{Effective D-Cache Capacity} = E(a, s, T)$$

Table 1. Performance results for `advect3d` (large) for different fusion strategies

Fusion Strategy	Cycles ($\times 10^8$)	L1D Misses ($\times 10^7$)	L2 Misses ($\times 10^6$)	L1 I Misses ($\times 10^5$)	Loads ($\times 10^8$)	Speedup over `no-fuse`
`ccfm`	8.41	4.48	5.13	6.14	3.66	1.17
`simple`	12.30	3.78	5.08	4.31	4.26	0.80
`mips-pro`	9.86	3.76	9.18	6.16	3.06	1.00
`no-fuse`	9.87	3.76	9.19	6.26	3.06	1.00

Table 2. Performance results for `advect3d` (small) for different fusion strategies

Fusion Strategy	Cycle ($\times 10^8$)	L1D Misses ($\times 10^7$)	L2 Misses ($\times 10^6$)	L1 I Misses ($\times 10^5$)	Loads ($\times 10^8$)	Speedup over `no-fuse`
`ccfm`	4.22	1.38	2.29	6.98	1.19	1.08
`simple`	5.70	1.68	2.79	7.80	1.61	0.80
`mips-pro`	5.73	1.68	2.80	7.80	1.61	0.80
`no-fuse`	4.58	1.46	2.49	6.98	1.30	1.00

where $E(a, s, T)$ is obtained from Eq. 1.

We start of with a low value for T ($T < 0.02$) and at each step we increment T by 0.05 and measure the number of data cache misses at different levels. We stop the search in this dimension when we reach a T for which the number of cache misses increases.

Register Pressure: For the register pressure constraint we have the following equation for T:

$$\textit{Effective Registers} = \lceil T \times \text{Register Set Size} \rceil \text{ where } 0 \leq T \leq 1$$

Feedback parameters we use here are total loads and cycle count. Both parameters serve as good indicators about the occurrence of register spills.

Instruction Cache Capacity: The instruction cache constraint is dealt separately since we do not compute reuse distances for instruction and we are mainly concerned with capacity misses. So, in this case we have:

$$\textit{Effective I-Cache Capacity} = \lceil T \times \textit{Capacity}(\textit{I-Cache}) \rceil \text{ where } 0 \leq T \leq 1$$

For feedback we measure instruction cache misses directly.

4 Preliminary Evaluation

We are currently in the process of implementing our profitability model in a performance-based empirical tuning framework[12]. The system includes a source-to-source code transformer (`LoopTool`) that is capable of performing a collection of loop optimizations including multi-level fusion. In this section, we present an evaluation of our model using the empirical tuning framework.

Table 3. Performance results for `erlebacher` for different fusion strategies

Fusion Strategy	Cycle ($\times 10^9$)	L1D Misses ($\times 10^8$)	L2 Misses ($\times 10^7$)	L1 I Misses ($\times 10^4$)	Loads ($\times 10^8$)	Speedup over `no-fuse`
`ccfm`	5.23	2.00	2.72	6.57	4.02	1.08
`simple`	5.68	1.85	3.09	6.77	3.90	0.99
`mips-pro`	5.23	1.70	2.74	9.85	4.52	1.08
`no-fuse`	5.65	2.34	2.92	5.95	4.34	1.00

Table 4. Performance results for `liv18` for different fusion strategies

Fusion Strategy	Cycle ($\times 10^9$)	L1D Misses ($\times 10^8$)	L2 Misses ($\times 10^7$)	L1 I Misses ($\times 10^4$)	Loads ($\times 10^9$)	Speedup over `no-fuse`
`ccfm`	3.77	2.14	2.33	4.52	1.55	1.46
`simple`	3.77	2.14	2.33	4.52	1.55	1.46
`mips-pro`	5.06	2.32	3.33	5.54	0.98	1.09
`no-fuse`	5.51	2.62	4.08	5.13	1.18	1.00

We applied our model by hand to a set of benchmarks. We then annotated the source with directives to tell `LoopTool` which loops to fuse. The transformed code was then compiled using the native compiler on the target platform.[1] In order to avoid conflicts with the fusion strategies of the native compiler, programs transformed by `LoopTool` were compiled with the fusion option turned off. All experiments were performed on an SGI R12K machine with a two-level cache hierarchy. Experiments were run on four different programs: `advect3d` an advection kernel for weather modeling, `erlebacher` a differential equation solver, `liv18` a hydrodynamics kernel from Livermore loops and `mgrid`, a multi grid solver from SPEC 2000. We compare results from applying our strategy (`ccfm`) with three different strategies: the `simple` strategy always fuses loops that share some common data, `mips-pro` is the fusion strategy chosen by the MIPSPro 7.3 compiler and `no-fuse` is the option of applying no fusion at all.

Results from `advect3d` using a $256 \times 256 \times 256$ data set is presented in Table 1. The results show that our strategy is able to achieve a 17% speedup over both `mips-pro` and `no-fuse`. Performance improvement of `ccfm` over `simple` is even more dramatic (46%). For `advect3d`, `ccfm` fuses all loops at the two outer levels but refrains from fusing all the innermost loops because it estimates the register pressure will exceed available resources on the target machine. `simple` fuses all loops at each nesting level and creates a large fused body for the inner loop. As a result, this version of the code incurs many register spills as indicated by the large number of issued loads in column 6 of Table 1. Although `simple` is able to achieve some locality in L1 and L2 cache and also the L1 instruction cache, the

[1] Since, we applied the model by hand we do not have numbers for the total tuning time. The measured time for the source-to-source transformation was never more than 15 seconds.

Table 5. Performance results for `mgrid` for different fusion strategies

Fusion Strategy	Cycle ($\times 10^{10}$)	L1D Misses ($\times 10^{8}$)	L2 Misses ($\times 10^{7}$)	L1 I Misses ($\times 10^{5}$)	Loads ($\times 10^{9}$)	Speedup over `no-fuse`
`ccfm`	1.05	4.63	6.37	3.39	3.64	1.07
`simple`	1.02	4.53	6.27	3.31	3.59	1.11
`mips-pro`	1.02	4.53	6.27	3.26	3.59	1.11
`no-fuse`	1.13	5.14	6.86	3.74	3.74	1.00

cost of register spills for this strategy outweighs its benefits. The peformance of `mips-pro` and `no-fuse` is almost identical in this case. Closer inspection of the generated code revealed that MIPSPro chose not to fuse any loops for `advect3d` because the data set was too large for the stack frame size for the target machine. For this reason, we ran another set of experiments with `advect3d` using a smaller ($128 \times 128 \times 128$) data set. Results from the second set of experiments are shown in Table 2. Again, `ccfm` performs significantly better than both `no-fuse` and `simple`. Although, the performance gains have somewhat diminished due to the smaller data set. The more interesting result from this set of experiments is the performance of the `mips-pro` strategy. `mips-pro` performs as poorly as `simple` in this case. We inspected the code generated by `mips-pro` and discovered that it created two separate fully fused loop nests from the 27 fusible loops in the program. In addition, it performed tiling on each fused loop nest. As it turned out the combination of fusion and tiling was not able to improve locality in the program. This is indicated by the increased number of misses at all levels of the cache. These results demonstrate that indiscriminate fusion can indeed lead to performance degradation. Our fusion strategy, although less aggressive, achieves locality at both cache levels while keeping the register spill cost at a moderate level. Hence, we are able to achieve an overall performance improvement across all levels of the memory hierarchy.

Results from our experiments with `erlebacher` are presented in Table 3. Again `ccfm` is able to outperform both `simple` and `no-fuse` through improved locality in the L2 cache. However, in this case `mips-pro` does as well as `ccfm`. For `erlebacher`, `mips-pro` fuses loops that our fusion strategy rejects because of lost reuse in the outer loops. However, as was the case with `advect3d`, the MIPSPro compiler applies tiling to these fused loops and in this case tiling is able to recover some of the lost reuse due to over fusion. Thus there is no significant increase in the number of L2 cache misses for `mips-pro`.

In Table 4 we present results from `liv18`. We observe the most significant performance improvement for this kernel. This is not surprising since all the work in `liv18` is spent in three fusible loop nests. For `liv18`, our fusion strategy chooses to fuse all three loops all the way through which is equivalent to the `simple` strategy. Thus in Table 4 the rows corresponding to `ccfm` and `simple` are identical. We notice that fusing all the way through does cause some extra loads. However, this loss is more than offset by the benefits obtained from reduced L2

cache misses. `mips-pro` does not do too well on `liv18`. It decided to fuse only two of the three fusible loops in the kernel leaving some unexploited reuse in the third loop nest. It was not totally clear as to why `mips-pro` decided not to fuse the third loop nest. We speculate that it may have been due to loop alignment issues. The loop nests in `liv18` need to be aligned before they can be fused. For `LoopTool` we use the Omega code generator which inserts guards within fused loop nests after alignment. On the other hand, it appears that MIPSPro prefers to peel off iterations of the loop nest that fall outside the alignment range. It is possible that because of this approach the third loop nest was left unfused. Thus the performance improvement we observe over `mips-pro` may not be due to an improved profitability model but rather due to a limitation in their implementation of the fusion algorithm.

The final benchmark we look at is `mgrid`. The experimental results from `mgrid` are presented in Table 5. In this case, although `ccfm` achieves better performance than `no-fuse` it is beaten by both `mips-pro` and `simple`. `mgrid` poses a similar situation as `advect3d` for our fusion strategy. Because `ccfm` expects lost reuse in outer levels it chooses to perform only a two level fusion leaving the innermost loops alone. On the other hand, `mips-pro` decided to fuse all the way through and then apply both tiling and outer loop unrolling to the fused loop nests. This combined transformation strategy improved locality for L2 cache and also reduced the number of loads for the program.

We summarize the results of our experiments in Fig 2. The experimental results presented in this section expose several key aspects for profitable loop fusion. The results show that overly aggressive fusion can indeed lead to performance loss through increased register pressure and lost reuse at outer levels of loop nests. In some cases, this loss can be mitigated by applying transformations such as tiling and unroll-and-jam. However, there are cases when these additional transformations are unable to help improve the overall performance. Thus the interaction between fusion and other transformations, particularly tiling is critical in improving memory performance. To address this issue, we have begun work on a more complex model discussed in the concluding section.

5 Accuracy of the Cache Miss Prediction Model and its Implications

The cache miss model presented in Section 3.1 makes the assumption that memory accesses between any two reused references are essentially random. Although, this scheme works well when integrated with the rest of our framework it is important to evaluate the accuracy of the model on its own. To validate our model, we performed a series of experiments with a set of synthetic benchmarks and real-world applications [11]. In this section, we provide a brief summary of the experimental results and discuss their implications.

Experimental results from [11] revealed that our model is able to predict an *upper bound* for the conflict miss rate with reasonable accuracy. However, the predicted upper bound for the miss rate can sometimes be significantly greater

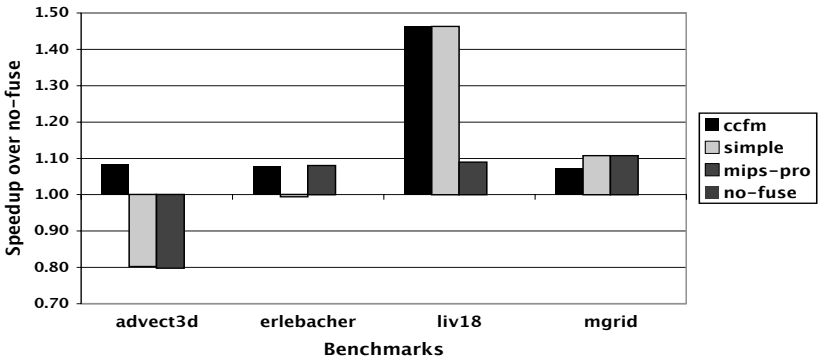


Fig. 2. Performance improvement for different fusion strategies

than the actual miss rate of the program. Although a conservative estimate suffices for profitability estimates of loop fusion it is important to consider its implications on other transformations. A key transformation for improving memory performance in numerical applications is tiling. If we use our conflict miss model with tiling then the *effective cache capacity* would directly determine the tile size for a given loop nest. In that case, a conservative estimate would imply choosing a smaller tile size which in turn may lead to lost reuse in inner loops. Therefore, in such situations we need a cache miss model that is able to predict the cache miss rate more accurately. We are currently working on such a model. Our new model incorporates the effects of tiling and also considers the layout of arrays in memory.

6 Conclusions and Future Work

In this paper, we have presented a model for estimating the profitability of loop fusion and a strategy for parameterizing the model for use in an empirical tuning framework. Preliminary experiments in Section 4 suggest that our strategy can help make the right fusion choices on a set of applications. However, to make a stronger statement about the effectiveness of our approach the model has to be evaluated on a large class of benchmarks and a variety of platforms. Our future plans include a complete implementation of the model in our empirical tuning framework and a more thorough evaluation on a large benchmark suite.

Experimental results from Section 4 also emphasize the need for considering interactions between optimizations for overall improvement in memory performance. In particular, there are complex interactions between tiling and fusion that need to be considered to make fusion profitable. By merging loop bodies fusion can increase the working set size of a loop nest and force the selection of a smaller tile size. A smaller tile size might result in lost reuse in the inner loops. If arrays are not aligned at cache line boundaries (generally the case) then a smaller tile size may result in lost reuse in outer loops as well. In such cases, it

may be profitable to tile the two loop nests separately. We are currently working on a profitability model that considers these complex interactions between tiling and fusion to improve overall memory performance. In addition, this model employs a more accurate estimator for *effective cache capacity* that takes the effects of tiling and array allocation strategies into account.

References

1. S. Carr. *Memory-Hierarchy Management*. PhD thesis, Dept. of Computer Science, Rice University, Sept. 1992.
2. A. Darte. On the complexity of loop fusion. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.
3. C. Ding and K. Kennedy. Resource-constrained loop fusion. Technical report, Dept. of Computer Science, Rice University, Oct. 2000.
4. 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.).
5. 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.
6. M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12), 1989.
7. K. Kennedy. Fast greedy weighted fusion. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, 2000.
8. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.
9. 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.
10. 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.
11. A. Qasem and K. Kennedy. Evaluating a model for cache conflict miss prediction. Technical report, Dept. of Computer Science, Rice University, Oct. 2005.
12. 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.
13. 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.
14. S. Verdoolaege, M. Bruynooghe, G. Jenssens, 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.
15. 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.