A Cost Framework for Evaluating Integrated Restructuring Optimizations

Bharat Chandramouli, John B. Carter, Wilson C. Hsieh, Sally A. McKee
School of Computing
University of Utah

Abstract

Loop transformations and array restructuring optimizations usually improve performance by increasing the memory locality of applications, but not always. For instance, loop and array restructuring can either complement or compete with one another. Previous research has proposed integrating loop and array restructuring, but there existed no analytic framework for determining how best to combine the optimizations for a given program. Since the choice of which optimizations to apply, alone or in combination, is highly application- and input-dependent, a cost framework is needed if integrated restructuring is to be automated by an optimizing compiler. To this end, we develop a cost model that considers standard loop optimizations along with two potential forms of array restructuring: conventional copying-based restructuring and remapping-based restructuring that exploits a smart memory controller. We simulate eight applications on a variety of input sizes and with a variety of hand-applied restructuring optimizations. We find that employing a fixed strategy does not always deliver the best performance. Finally, our cost model accurately predicts the best combination of restructuring optimizations among those we examine, and yields performance within a geometric mean of 5% of the best combination across all benchmarks and input sizes.

1 Introduction

Processor and memory speeds have been diverging at the rate of almost 50% a year, but architectural trends suggest that cache sizes will remain small to keep pace with decreasing processor cycle times. McFarland [10] shows that for a feature size of 0.1 micron and a 1-nsec cycle time, the L1 cache can be no bigger than 32 kilobytes to maintain a one-cycle access time, or 128 kilobytes for two-cycle latency. High memory latency, increasing CPU parallelism, poor cache utilization, and small cache sizes relative to application working sets all create a memory bottleneck that results in poor performance for applications with poor locality. Bridging the growing processor/memory performance gap requires innovative hardware and software solutions that use cache space and memory bandwidth more efficiently. Myriad approaches attempt to do this in different ways, and application and input characteristics highly influence the appropriate choice of optimizations. An analytic framework that models the costs and benefits of these optimizations can be useful in driving decisions.

Iteration space transformations [5, 11, 17] and data layout transformations [3, 8] are two classes of compiler optimizations that improve memory locality. *Loop transformations*, an example of the former, improve performance by changing the execution order of a set of nested loops so that the temporal and spatial locality of a majority of array accesses is increased. This class includes loop permutation, fusion, distribution, reversal, and tiling [11, 17].

Array restructuring is a common data layout transformation. It improves cache performance by changing the physical layout of arrays that are accessed with poor locality [8]. Static restructuring changes the compile-time layout of an array to match the way in which it is most often accessed. For example, the compiler might choose column-major order over row-major order if most accesses to an array are via column walks. Static restructuring is most useful when an array is accessed in the same way throughout the program. Dynamic restructuring creates a new array at run time, so that the new layout better matches how the data is accessed. This is most useful when access patterns change during execution, or when access patterns cannot be determined at compile time.

Dynamic array restructuring is more widely applicable than static, and we focus on it in this study. The runtime change in array layout is most often accomplished by copying, and we refer to this optimization as *copying-based array restructuring*. We also consider the possibility of having smart memory hardware perform the restructuring [2]. Hardware mechanisms that support data remapping allow one to create array aliases that are optimal for a particular loop nest. We call this optimization *remapping-based array restructuring*. The tradeoffs involved are different from that of traditional copying-based array restructuring, and thus this optimization is sometimes an useful alternative when the latter is expensive.

However, hardware support does not come for free, and thus there is a need to determine automatically whether hardware support should be relied upon.

Loop transformations and array restructuring can be complementary, and often are synergistic. When applicable, loop transformations improve memory locality with no runtime overhead. However, it is often not possible to improve the locality of all arrays in a nest. For example, if an array is accessed via two conflicting patterns (e.g., $a[i][j]$ and $a[j][i]$), no loop ordering can improve locality for all accesses. Furthermore, loop transformation cannot be applied when there are complex loop-carried dependences, insufficient or imprecise compile-time information, or non-trivial imperfect loop nests. In contrast, array restructuring can always be applied, since it affects only the locality of the target array. However, all forms of dynamic restructuring incur overheads, and these costs must be amortized across the accesses with improved locality for the restructuring to be profitable. Loop and array restructuring can be integrated and the best choice depends on which combination has the minimum overall cost.

Others have integrated data restructuring and loop transformations [3, 5], but their approaches only consider static array restructuring, and do not provide any mechanisms to determine whether the integration will be profitable. In this paper, we present cost models that capture the cost/performance tradeoffs of loop transformations, copying-based array restructuring, and remapping-based restructuring. We use an integrated cost framework to decide which optimizations to apply, either singly or in combination, for any given loop nest. Code optimized using our cost model achieves performance within 95% of the best observed for a set of eight benchmarks. In contrast, the performance of any fixed optimization is at best 76% of the best combination we observed.

2 Restructuring Optimizations

Consider the simple example loop nest, *ir_kernel*, in Figure 1(a). If we assume row-major storage, two of the arrays are accessed sequentially, with U having good spatial locality and V having good temporal locality. Unfortunately, W is accessed along its columns, and X is accessed diagonally; neither will enjoy good cache performance. In this section, we review the candidates for integrated restructuring and examine their effects on *ir_kernel*.

2.1 Loop Transformations

The loop transformations we consider in this paper include loop permutation, loop fusion, loop distribution and loop reversal. McKinley *et al.*[11] choose between candidate loop transformations based on a simple cost model.

array references	innermost loop		
	i	j	k
U[k]	$1*N^2$	$1*N^2$	$\frac{1}{16}N*N^2$
V[i]	$\frac{1}{16}N*N^2$	$1*N^2$	$1*N^2$
W[j][i]	$\frac{1}{16}N*N^2$	N^3	$1*N^2$
X[i+j+k][k]	N^3	N^3	N^3
total	$\frac{9}{8}N^3+N^2$	$2N^3+2N^2$	$\frac{17}{16}N^3+2N^2$

Table 1: Estimated loop cost for the ir_kernel example and a cache line size of 16. This table shows the cost per array and total cost when each loop, in turn, is placed innermost in the nest.

They define *loop cost* to be the estimated number of cache lines accessed by all arrays when a given loop is placed innermost in the nest. Evaluating the loop cost for each loop and ranking the loops in descending cost order (subject to legality constraints) yields a permutation with least cost. The resulting loop nest is said to be in *memory order*. They did not model tiling, and we do not either since more exact analysis is required to calculate the cross-interference and self-interference misses. Applying their cost model to our example loop generates the costs shown in Table 1.

Since the loop costs in descending order are $j > i > k$, the recommended loop permutation is *jik*, shown in Figure 1(b). Arrays U, V, and W are now all accessed sequentially, but array X still has poor cache locality. Loop permutation alone is insufficient to optimize all accesses in our example loop.

2.2 Copying-based Array Restructuring

Array restructuring directly improves the spatial locality of array accesses. It makes sense to store one array in row-major order if it is accessed row-by-row and another array in column-major order if it is accessed column-by-column. Array restructuring generalizes this idea to any direction. For instance, Figure 2 shows how the skewed access pattern in the original array becomes sequential after array restructuring. We say that an array reference is in *array order* if its access pattern in the loop matches its storage order.

When applying copying-based array restructuring to the loop nest in Figure 1(a), we create copies of X and W, cX and cW, that are in array order. Leung and Zahorjan derived the formalisms needed to create index transformation matrices that specify how the indices of each new array relate to those of the corresponding original [9]. In this case, the array X is transformed such that $cX[i-j+N][j]$ maps to $X[i][j]$, and W such that $cW[j][i]$ maps to $W[i][j]$. Figure 1(c) shows the *ir_kernel* code after copying-based array restructuring. Arrays cX and cW have stride one ac-

```
double U[N],V[N],W[N][N],X[3N][N];
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0; k<N; k++)
         U[k] += V[i] + W[j][i] + X[i+j+k][k];
```

(a) *original nest*

```
double U[N],V[N],W[N][N],X[3N][N];
for(j=0; j<N; j++)
   for(i=0; i<N; i++)
      for(k=0; k<N; k++)
         U[k] += V[i] + W[j][i] + X[i+j+k][k];
```

(b) *loop transformed*

```
double U[N],V[N],W[N][N],X[3N][N],cX[4N][N],cW[N][N];
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      cW[j][i]=W[i][j];
for(i=0;i<3*N;i++)
   for(j=0;j<N;j++)
      cX[i-j+N][j]=X[i][j];
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0;k<N;k++)
         U[k]+=V[i]+cW[i][j]+cX[i+j+N][k];
```

(c) *copy restructured*

```
double U[N],V[N],W[N][N],X[3N][N],*rW,*rX;
map_shadow(&rW,TRANSPOSE,W_params);
map_shadow(&rX,BASESTRIDE,X_params);
for(i=0;i<N;i++)
   for(j=0;j<N;j++){
      offset=(i+j)*N;
      remap_shadow(&rX,offset);
      for(k=0;k<N;k++)
         U[k]+=V[i]+rW[i][j]+rX[k];
      flush_cache(rX);
   }
```

(d) *remap restructured*

Figure 1: An example loop nest and three optimizations.

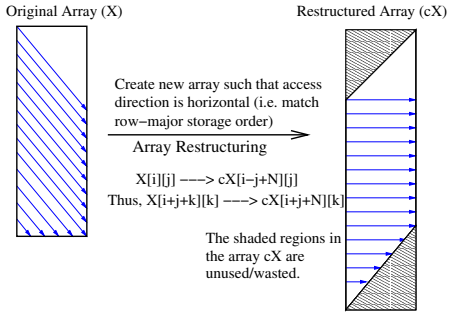

Figure 2: Restructuring an array with a skewed access pattern.

cesses in the loop nest, and thus exhibit good cache and TLB locality.

Restructuring may waste storage, as illustrated by the shaded regions of the restructured array in Figure 2. In the optimized *ir_kernel*, shown in Figure 1(c), there is no unused memory in array *cW*, and the amount of unused memory in array *cX* is $4N^2 - 3N^2 = N^2$. The profitability of array restructuring depends on the array and loop nest parameters. For example, if the size of the array is much larger than the amount of use it has in the loop nest, then the setup costs of creating the new arrays might dominate the benefits of improved spatial locality of the restructured array.

2.3 Remapping-based Array Restructuring

In this section, we briefly explain the details of the hardware mechanism we use to implement remapping-based restructuring, and show how we apply the optimization to the example *ir_kernel*.

The Impulse adaptable memory system expands the traditional virtual memory hierarchy by adding address translation hardware to the main memory controller (MMC) [2]. The memory controller provides an interface for software (e.g., the operating system, compiler, or runtime libraries) to remap physical memory to support different views of data structures in the program. Thus, applications can improve locality by controlling how the new view to the data is created. To use Impulse's address remapping, an application performs a *map_shadow* system call indicating what kind of remapping to create — e.g., a virtual transpose of an array — along with the original starting virtual address, the element size, and the dimensions. The OS configures the Impulse MMC to respond to accesses to this remapped data appropriately. When the application accesses a cache line at an address corresponding to a remapped data structure, the Impulse MMC determines the real physical addresses corresponding to the remapped elements requested and loads them from memory. To support this functionality, the MMC contains both a pipelined address calculation unit and a TLB. The elements are loaded into an output buffer by an optimized DRAM scheduler and then sent back to the processor to satisfy the load request.

Figure 1(d) shows how the memory controller is configured to support remapping-based restructuring for the *ir_kernel*. The first *map_shadow()* system call configures the memory controller to map reference $rW[i][j]$ to $W[j][i]$. The second map_shadow() call configures the memory controller to map reference $rX[k]$ to $X+offset+k*stride$, where $stride$ is $N+1$. The *offset*, which is $(i+j)*N$, is updated each iteration of the j loop using the *remap_shadow* system call to re-

flect the new values of i and j, i.e., the new diagonal being traversed. Thus, a reference $rX[k]$ translates to $X+((i+j)*N)+k*(N+1)$, which is simply the original reference $X[i+j+k][k]$. We flush rX from the cache to maintain coherence when the offset changes.

After applying the remapping-based restructuring optimization, all accesses are in array order. The cost of setting up a remapping is quite small compared to copying. However, subsequent accesses to the remapped data structure are slower than accesses to an array restructured via copying, because the memory controller must re-translate addresses on the fly and "gather" cache lines from disjoint regions of physical memory. Our cost model accounts for this greater access latency for remapped data when considering which restructuring optimization to apply, if any.

3 Analytic Framework

To optimize to a program, a compiler must perform the following two steps. First, it *decides* upon a part of the program to optimize and a particular transformation to apply to it. Second, it *transforms* the program and verifies that the transformation either does not change the meaning of the program or changes it in a way that is acceptable to the user. Bacon *et al.* [1] compare the first step to a black art, because it is difficult and poorly understood. In this section, we explore this first step in the domain of restructuring optimizations. We present analytic cost models for each of the optimizations, and analyze tradeoffs in individual restructuring optimizations, both qualitatively and quantitatively. We show why it might be beneficial to consider them in an integrated manner, and present a cost framework that allows integrated restructuring optimizations to be evaluated.

3.1 Modeling the Restructuring Strategies

3.1.1 Basic Model

We need careful cost/benefit analysis to determine when compiler optimizations may be profitably applied because finding the best choice via detailed simulation or hardware measurements is time-consuming and expensive. We have developed an analytic model to estimate the memory costs of applications at compile time. Like McKinley, Carr and Tseng [11], we estimate the number of cache lines accessed in the loop nests.

The memory cost of a single array reference, say R_α, in a loop nest is directly proportional to the number of cache lines accessed by it. Suppose *cls* is the cache line size of the cache closest to memory, *stride* is the distance in elements between successive accesses to this array, and f is the fraction of the cache lines reused from a previous iteration of the innermost loop. We estimate the memory cost of a single array reference R_α to be:

$$MemoryCost(R_\alpha) = \frac{loopTripCount}{max(\frac{cls}{stride},1)} \times (1-f) \qquad (1)$$

We do not have a framework for estimating f accurately. We expect f to be very small for large working sets, which allows us to neglect the $(1-f)$ term without introducing significant inaccuracies (see Section 4.3 for a discussion of the limitations of our cost model). For the example in Figure 1(a), the estimated number of cache lines accessed by array X is $\frac{N^3}{max(\frac{cls}{N+1},1)}$, by array V is $\frac{N}{cls}$, and by array U is $\frac{N^3}{cls}$.

We estimate the memory cost of a loop nest to be the sum of the memory costs of the independent array references in the nest. We define two array references to be *independent* if they access different cache lines in each iteration. This characterization is necessary to avoid overcounting cache lines. Thus, if both $a[i][j]$ and $a[i][j+1]$ are present, we consider them as one. The cost of a loop nest depends on *loopTripCount* (the total number of iterations of the nest), the spatial and temporal locality of the array references, the stride of the arrays and the cache line size. We estimate the memory cost of the i^{th} loop nest in the program to be:

$$MemoryCost(L_i) = \sum_{\alpha:independentRef} MemoryCost(R_\alpha) \qquad (2)$$

The memory cost of the entire program is estimated as the sum of the memory costs of all the loop nests. If there are n loop nests in a program, then the memory cost of the program is:

$$MemoryCost(program) = \sum_{i=1}^{n} MemoryCost(L_i) \qquad (3)$$

For the example code in Figure 1(a), the total memory cost is $\left(N^3 \times (1+\frac{1}{cls}) + N^2 + \frac{N}{cls}\right)$.

The goal of the cost model is to choose the combination of array and loop restructuring optimizations that has minimum memory cost for the entire program. We assume that the relative order of memory costs determines the relative ranking of execution times. The above formulation of total memory cost of an application as the sum of the memory costs of individual loop nests makes an important assumption — the compiler will know the number of times each loop nest executes, and can calculate the loop bounds of each loop nest at compile-time. This assumption holds for many applications. In cases where it does not hold, we expect to be able to make empirical assumptions about the frequencies of loop nests. We also assume that the cache line size of the cache closest to memory is known to the compiler.

3.1.2 Modeling Loop Transformations

When we consider only loop transformations, the recommendations from our model are the same as those of the simpler model by McKinley *et al.* [11]. However, their approach does not consider array restructuring. We consider the total memory cost of the loop nests, which allows us to compare the cost of loop transformations with that of independent optimizations such as array restructuring.

3.1.3 Modeling Array Restructuring

The cost of array restructuring is the sum of the initial cost of creating the new array and that of the optimized loop nest. In addition, there is the cost of updating the original array if the new array was modified. The cost of setup is the sum of the memory costs of the original array and the new array in the setup loop.

$$MemoryCost(CopyingSetup) = OriginalArraySize \times (min(\frac{newArrayStride}{cls}, 1) + \frac{1}{cls}) \quad (4)$$

In Figure 1(C), the memory cost of creating array cX in the setup loop is $\frac{3N^2}{\max(\frac{cls}{N+1}, 1)}$ and the memory cost of array X in the setup loop is $\frac{3N^2}{cls}$. Thus, the setup cost of creating array cX is $3N^2 \times (1 + \frac{1}{cls})$ if $(N+1) > cls$, which is usually the case. The calculation for array cW is similar.

The cost of the optimized array reference in the loop nest is:

$$MemoryCost(restructuredReference) = loopTripCount \times \frac{1}{cls} \quad (5)$$

The cost of the loop nest with optimized references, cX and cW, is $\frac{1}{cls} \times (2N^3 + N^2 + N)$. Array restructuring is expected to be profitable if:

$$\begin{aligned} & MemoryCost(copyingSetup) + \\ & MemoryCost(restructuredReference) \\ < \quad & MemoryCost(originalReference) \end{aligned} \quad (6)$$

For this example, the total cost of the array-restructured program (assuming $cls = 16$) is $(\frac{N^3}{8} + \frac{69N^2}{16} + \frac{N}{16})$, while the cost of the original program is $(\frac{17N^3}{16} + \frac{N}{16} + N^2)$. The latter is larger for almost all N, and so array restructuring is assumed to be always profitable for this particular loop nest. Simulation results bear this decision out.

3.1.4 Modeling Remapping-based Restructuring

When using remapping-based hardware support, we can no longer model the memory costs of an application as being directly proportional to the number of cache lines accessed, since all cache line fills no longer incur the same cost. Cache line fills to remapped addresses undergo a further level of translation at the memory controller, as explained in Section 2.3. After translation, the corresponding physical addresses need not be sequential, and thus the cost of gathering a remapped cache line depends on the stride of the array, the cache line size of the cache closest to memory, and the efficiency of the DRAM scheduler. To accommodate this variance in cache line gathering costs, we model the total memory cost of an application as proportional to the number of cache lines gathered times the cost of gathering the cache line. The cost of gathering a normal cache line, G_c, is fixed, and the cost of gathering a remapped cache line, G_r, is fixed for a given stride. We used a series of microbenchmarks to compute a table of G_r values indexed by stride that we consult to determine the cost of gathering a remapped cache line. Thus, if a program accesses n_c normal cache lines, n_1 remapped cache lines remapped with stride s_1, n_2 remapped cache lines remapped with stride s_2, and so on, then the memory cost of the program is modeled as:

$$MemoryCost(program) = n_c \times G_c + \sum_i n_i \times G_r(s_i) \quad (7)$$

The overhead costs involved in remapping-based array restructuring include the cost of remapping setup, the cost of updating the memory controller, and the cost of cache flushes. The initial cost of setting up a remapping through the *map_shadow* call is dominated by the cost of setting up the page table to cache virtual-to-physical mappings. The size of the page table depends on the number of elements that are to be remapped. We model this cost as $K_1 \times \#elementsToBeRemapped$. Updating the remapping information prior to entering the innermost loop every time using the *remap_shadow* system call incurs a fixed cost, which we model as K_2. We also model the costs of flushing as being proportional to the number of cachelines flushed where the constant of proportionality is K_3. We have empirically estimated these constants using microbenchmarks.

The memory cost of an array reference optimized with remapping support is:

$$MemoryCost(remappedReference) = \frac{loopTripCount}{max(\frac{cls}{stride}, 1)} \times (1 - f) \times G_r(stride) \quad (8)$$

$$MemoryCost(normalReference) = \frac{loopTripCount}{max(\frac{cls}{stride},1)} \times (1-f) \times G_c \quad (9)$$

In summary, multiplying the number of cache lines gathered by the cost of gathering the cache lines makes remapping-based array restructuring comparable with pure software transformations such as copying-based array restructuring and loop transformations.

Returning to our example in Figure 1(d), the memory cost of array rX is $\frac{1}{16} \times N^3 \times G_r$, where G_r is based on the stride $(N+1)$. The total cost of remapping-based restructuring is $(\frac{1}{16} \times (N^3+2N^2) * G_c + \frac{1}{16} \times N^3 \times G_r + K_1 \times 3N^2 + K_2 \times N^2 + K_3 \times N^3)$. This is less than the original program's memory cost and thus remapping-based restructuring is correctly assumed to be profitable for this example.

3.2 Integrated Restructuring

By *integrated restructuring* we mean two things. First, the individual restructuring optimizations can be combined to optimize an application in complementary ways. For example, we could combine loop permutation, loop fusion, and copying-based array restructuring to optimize one single loop nest. Second, we should be able to select a good combination of restructuring optimizations from the legal options given the application and its inputs. Researchers have provided heuristics-driven algorithms for combining loop and array restructuring [3, 5] but there has not been any work on providing a framework for choosing the right set of optimizations among restructuring optimizations. Our cost model framework for selecting the right combination of restructuring optimizations is based on Equations $1-9$. These equations allow us to decide which optimization to choose for a given application and input size.

Loop transformation incurs no run-time costs and thus is the optimization of choice when it succeeds in rendering all references in array order. In the presence of conflicting array access patterns, however, loop transformations cannot improve the locality of all arrays. In such cases, array restructuring can be applied to individual references that lack the desired locality. The choice of what loop transformation to use might therefore depend on which array(s) are cheaper to restructure.

We analyze the example loop nest in Figure 1(a) to see whether integrated restructuring improves performance. Recall that loop transformation could not improve the locality of array X, and array restructuring incurred the costs of creating the arrays cX and cW through copying. Consider the integrated restructuring optimization that permutes the loop nest into jik order and uses array restructuring to improve the locality of X. This combination of optimizations achieves array order for all references and incurs only the setup cost of creating cX (W is already in array order after loop transformation). Integrated restructuring delivers slightly better performance than either loop or array restructuring alone for this example (see Table 3).

```
double U[N],V[N],W[N][N],X[3N][N];
double cX[4N][N];
for(i=0;i<3*N;i++)
   for(j=0;j<N;j++)
      cX[i-j+N][j]=X[i][j];
for(j=0;j<N;j++)
   for(i=0;i<N;i++)
      for(k=0;k<N;k++)
         U[k]+=V[i]+W[j][i]+cX[i+j+N][k];
```

Figure 3: *ir_kernel*, optimized by a combination of loop permutation and copying-based array restructuring.

The same loop nest can be optimized using loop transformation with remapping-based restructuring (**L+R**), in which case the loop transformation would bring array W into array order, and array X would be remapped to rX, as in Figure 1(d).

In general, selecting the optimal set of transformations is a non-trivial problem. Finding the combination of loop fusion optimizations alone for optimal temporal locality has been shown to be NP-hard [6]. The problem of finding the optimal data layout between different phases of a program has also been proven to be NP-complete [7]. As a result, researchers have used heuristics to make integrated restructuring tractable. Our analytical cost framework can help evaluate various optimizations, and we show later in our results that the cost model driven optimizations comes closer to best possible performance than any fixed optimization strategy.

4 Evaluation

To perform our studies, we used URSIM, an execution-driven simulator derived from RSIM [13]. URSIM models in detail a microprocessor similar to a MIPS R10000, a split-transaction MIPS R10000 bus that supports a snoopy coherence protocol, and the Impulse memory controller [12]. The processor modeled is four-way issue, out-of-order, and superscalar with a 64-entry instruction window. The L1 data cache is 32KB, non-blocking, write-back, virtually indexed, physically tagged, two-way associative, with 32-byte lines and has a one-cycle latency. The L2 data cache is 512KB non-blocking, write-back, physically indexed, physically tagged, two-way set associative, with 128-byte lines and has an eight-cycle latency. The instruction cache is assumed to be perfect. The TLB maps both instructions and data, has 128 entries, and is single-cycle, fully associative, and software-

managed. The bus multiplexes addresses and data, is eight bytes wide, and has a three-cycle arbitration delay and a one-cycle turn-around time. The system bus, memory controller, and DRAMs all run at one-third the CPU clock rate. The memory supports critical word first and returns the critical quad-word for a load request 16 bus cycles after the corresponding L2 cache miss. The memory system models eight banks, pairs of which share an eight-byte wide bus between DRAM and the MMC.

4.1 Benchmarks

To evaluate the performance of our cost-model driven integration strategy, we studied eight benchmarks used in previous studies of loop and/or data restructuring. Four of the benchmarks that we considered (*matmult, syr2k, ex1*, and the *ir_kernel* we discussed earlier in Section 2) have been studied previously in the context of data restructuring – the former two by Leung *et al.* [8] and the latter two by Kandemir *et al.* [5]. Three other applications – *btrix, vpenta, cfft2d* – are NAS kernels that have been evaluated in the context of both data and loop restructuring in isolation [5, 8, 11]. Finally, *kernel6* is the sixth Livermore kernel.

Table 2 shows which optimizations we considered for each benchmark. The optimization candidates are copying-based array restructuring(**C**), remapping-based restructuring(**R**), loop transformations(**L**), a combination of loop and copying-based restructuring(**L+C**), and a combination of loop and remapping-based restructuring(**L+R**). $\surd$ indicates that the optimization was possible, *N* indicates that the optimization was not needed, and *I* indicates that the optimization was either illegal or inapplicable. In our study, we hand-coded all optimizations; work is ongoing to add our cost model to the Scale compiler [16] to automate the transformations. We ran each benchmark for ten different input sizes, with the smallest input size typically just fitting into the L2 cache. Whenever there were several choices for a single restructuring strategy, we chose the best option. In other words, the results that we report for **L** is for the best loop transformation (loop permutation, fusion, distribution or reversal) among the ones we implemented. Similarly, the results for **L+R** is for the best combination of loop and remapping-based array restructuring that we implemented.

4.2 Results

In this section, we briefly analyze each benchmark and validate the effectiveness of our cost model. We report the results of our experiments in Table 3. For each input size, we simulate the performance of the base (unoptimized) benchmark and the best optimized version for each candidate optimization. Our primary performance metric is the geometric mean speedup obtained for each optimization compared to the baseline benchmark over the range of input sizes. In addition to the performance obtained by applying only a static optimization per benchmark, we also present the results obtained when our cost model is used to select dynamically which optimization to perform for a given input size (**CM-driven**) and the post-facto best optimization selection for each input size (**Best**). We refer to best performance as that resulting from making the best choices among the optimizations we consider.

Application	C	R	L	L+C	L+R
matmult	$\surd$	$\surd$	$\surd$	N	N
syr2k	$\surd$	$\surd$	N	N	$\surd$
vpenta	$\surd$	$\surd$	$\surd$	N	$\surd$
btrix	$\surd$	I	$\surd$	$\surd$	$\surd$
cfft2d	$\surd$	$\surd$	I	I	I
ex1	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
ir_kernel	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
kernel6	$\surd$	$\surd$	I	I	I

Table 2: Benchmark suite and candidates for the optimization choices.

As can be seen from Table 3, CM-driven optimization obtains an average of 94.9% of the best possible speedup (1.68 versus 1.77), whereas the best single optimization (in this case, copying-based array restructuring) obtained only 77.0% of the best possible speedup (1.35 versus 1.77). The reason for the good performance of cost model driven optimization is that the best optimization strategy is highly application and input dependent. Even within the same benchmark, the best choice is dependent on the size of the input data. For example, in *syr2k*, the cost-model was able to pick between **C** and **L+R** and got a higher speedup (1.88) than either **C** or **L+R** (i.e. it picked **C** when **L+R** was bad, and vice-versa). Overall, for most of the benchmarks, our cost model was usually able to select the correct strategy to employ, and when it failed to pick the best strategy, the choice it made was generally very close to the post-facto best choice among the restructuring optimizations. To better understand why the cost model worked well in most cases, and poorly in a few, we will discuss each benchmark program in turn.

matmult: *matmult* involves multiplying a *N* by *M* matrix by an *M* by *L* matrix to get the product matrix. One array is walked by columns and these strided accesses incur high TLB and cache misses. Remapping-based or copying-based array restructuring can be used to get unit strided access for this array. This is the only application for which loop permutation alone is sufficient to achieve unit stride for all arrays in the loop nest. While array restructuring improves the performance of *matmult*, loop restructuring can do so with lower setup costs. Our cost

Application	C	R	L	L+C	L+R	CM–driven	Best
matmult	1.26	1.24	1.34	-	-	1.34	1.46
syr2k	1.71	0.75	-	-	1.56	1.88	2.00
vpenta	0.60	1.12	1.47	-	1.61	1.54	1.65
btrix	1.49	-	1.72	1.67	1.47	1.72	1.80
cfft2d	2.77	2.90	-	-	-	2.90	2.91
ex1	1.53	0.71	0.66	0.76	0.43	1.53	1.53
ir_kernel	1.07	1.26	0.96	1.07	1.26	1.07	1.26
kernel6	1.24	1.92	-	-	-	1.99	2.00
Overall	**1.35**	**1.23**	**1.10**	**1.04**	**1.09**	**1.68**	**1.77**

Table 3: Mean speedup obtained when each of these choices were held fixed for all inputs, the cost–model driven speedup, and the best possible speedup.

model recognized this situation and recommended loop permutation over the other choices. Though loop permutation was the best choice for a majority of cases, conflict misses caused remapping to be better for some data sizes. Since conflict misses are not modeled, the cost model incorrectly recommended loop permutation for such cases, and achieved a lower speedup than the best possible one.

syr2k: The *syr2k* subroutine from the BLAS library is a banded matrix calculation that computes $C = \alpha A^T B + \alpha B^T A + C$. The arrays are $N \times N$ matrices and the width of the band in the matrix is b. The core loop references four array elements from different rows and columns during each iteration of the innermost loop, which results in poor cache and TLB hit rates for the baseline program. Data dependences negate the possibility of using loop permutation. We combine loop distribution and remapping-based array restructuring (**L+R**) to optimize this subroutine. Copying-based array restructuring (**C**) is also very effective at creating sequential access patterns. The number of accesses to the elements in the band is $O(Nb^2)$. Thus copying performs well for large bands, where the fixed setup cost of copying ($O(N^2)$) is amortized by the subsequent reuse, and remapping performs well for small bands. The cost model correctly identified this behavior in most, but not all, cases. By correctly choosing when to apply copying (**C**) and when to choose combined loop and remapping-based restructuring (**L+R**), the cost model driven optimizations achieved better performance than either in isolation.

vpenta: This benchmark accessed eight two-dimensional arrays in seven loop nests with large strides. Loop permutation was used to optimize the two most expensive loop nests. The remaining loop nests had strided accesses that loop transformations could not optimize. For the remaining array references with strided accesses, we considered remapping-based restructuring. The cost model recommendations performed 4% worse than a fixed choice of **L+R** and 9% worse than best. The model did not account for a "side effect" of remapping, whereby for input sizes that are a power-of-2, remapping eliminates a significant number of conflict misses. Our cost model does not account for cache conflict effects, and thus in this case it underestimates the potential benefits that remapping can achieve. A more sophisiticated cost model that employed cache miss equations [4] or a similar mechanism might be able to handle this case more effectively.

btrix: The innermost loop was written to be vectorizable, and involves strided accesses across four four-dimensional arrays. The access pattern of one array conflicted with that of the other three arrays. As a result, loop permutation alone could not bring all these arrays into array order. There were six non-obvious optimization candidates (two choices of **L+R**, two **L**, one each for **L+C**, **C**) involving loop fusion, permutation and array restructuring. Our cost model recommended copying-based array restructuring (**C**). In contrast, Leung's heuristics-based decision model recommended that copying-based array restructuring not be done [8]. Our experiments validated our cost model, as we obtained was a speedup of 49% despite the overhead of copying.

Though the cost model framework correctly predicted that array restructuring would be beneficial, it correctly predicted that loop transformations alone were the best choice for all experiments but one. The single misprediction was when the array dimensions were a power-of-2 which induced many conflict misses. Combining loop transformations with array restructuring was not as beneficial because the overhead of creating the restructured array was higher than the benefit derived.

cfft2d: This application implements two dimensional FFT. Previous work on loop transformations did not optimize this loop nest or reported no performance improvement [11]. Dependence constraints and imperfect loop nests make loop transformations infeasible, but did not prevent copying- or remapping-based array restructuring.

Like *syr2k*, the relative performance of array restructuring and remapping-based restructuring is input size dependent for *cfft2d*. The array sizes were $O(N^2)$**)**, but each element was touched $O(N^2 logN)$ times. Thus, copying-based restructuring performs best for large data sets where the higher one-time cost of setup ($O(N^2)$) is amortized by the lower cost per access ($O(N^2 logN)$). Conversely, remapping is preferable for small data sets. Our cost model correctly predicted this tradeoff for most cases and obtained a speedup very close to the best possible.

ex1: In this kernel, only two out of the six possible loop permutations were legal. Loop permutation by itself yielded almost no benefit, but it enabled data transformations that are beneficial. We considered five optimization candidates for this kernel, **C**, **R**, **L**, **L+R**, **L+C**. Eight experiments with different values of N (ranging from N=200 to N=500) were performed. Copying-based array restructuring obtains the best speedup in each case. Remapping has high latency of gathering cache lines, and caused it to not perform well. The cost model successfully predicted the best optimization in all cases.

ir_kernel: As predicted from our earlier discussion, a combination of loop transformation and data restructuring results in the best performance for the *ir_kernel*. **L+C** outperforms **C**, and **L+R** outperforms **R** in all cases. Between **L+R** and **L+C**, the former performed better, but the cost model predicted otherwise. The main reason for this misprediction is again the high incidence of conflict misses in **L+C** compared to **L+R**. The penalty for this incorrect prediction was high and causes the cost model's recommendations to be only 85% of **Best**.

kernel6: This kernel is a general linear recurrence equation solver. The recurrence relation makes loop permutation illegal. We used remapping-based and copying-based array restructuring to optimize this kernel. The choice of whether **R** or **C** is better depended on the amount of reuse in the loop nest. We ran 22 experiments with various loop and data set sizes. Our cost model accurately chose the best optimization in 21 cases, and even in the single misprediction the performance was very close. When we used the cost model to select the optimizations to perform, we achieved an overall geometric mean speedup of 1.99, whereas the best overall mean speedup was 2.00. In contrast, applying remapping (**R**) or copying (**C**) exclusively results in an average speedup of only 1.92 and 1.24, respectively.

In summary, we find that quantifying the overheads and accesses costs of the various optimizations allowed us to make good decisions about which optimization(s) to choose. The recommendations made by our cost model resulting in a mean overall speedup within 5% of the best combination of optimizations that we considered, whereas the best performance from any single optimization choice was 23% less than the best combination.

4.3 Caveats

Our cost framework has a number of known inaccuracies; improving its accuracy is an area for future work. First, we do not consider the impact of any optimization on TLB performance. For some input sizes, TLB effects can dwarf cache effects, so adding a TLB model in an interesting open issue. Second, we do not consider the impact of latency-tolerating features of modern processors, such as hit-under-miss caches and out-of-order issue instruction pipelines in our cost model. This may lead us to overestimate the impact of cache misses on execution time. For example, multiple simultaneous cache misses that can be pipelined in the memory system have less impact on program performance than spread out cache misses, but our model gives them equal weight. Third, we do not consider cache reuse (i.e., we estimated f to be zero) or cache interference. We assume that the costs of different loop nests are independent, and thus additive. If there is significant inter-nest reuse, our model will overestimate the memory costs and recommend an unnecessary optimization. We do not have a framework for calculating f, and would benefit from a framework such as cache miss equations proposed by Ghosh *et al.* [4]. Similarly, not modeling cache interference could result in the model underestimating the memory costs if there are significant conflict misses in the optimized applications.

5 Related Work

Wolf and Lam [17] present definitions of different types of reuse and propose an algorithm for improving data locality based on unimodular transformations. McKinley, Carr and Tseng [11] present a loop transformation framework that combines loop permutation, loop fission and loop distribution. These computation-reordering optimizations generally reorder iterations of a loop nest based on a model of likely cache misses for various configurations of the loop nests. However, neither of these consider data transformations.

Leung and Zahorjan [9] introduce array restructuring, a technique to improve locality of array-based scientific applications. They do not propose a profitability analysis framework for determining when the optimization should be applied. Instead, they use a simple heuristic that takes neither the size of the array nor the loop bounds into account. Consequently, their decisions are always fixed for a particular application, regardless of input.

Cierniak and Li [3] propose a unified framework for optimizing locality combining data and control transformations. Kandemir *et al.* [5] extend Li's work by considering a wider set of possible loop transformations. These studies consider only static data transformations, and neither performs any cost/benefit analysis to decide the appli-

cability of their optimizations. Our integrated optimization strategy considers dynamic data restructuring and presents a general framework for profitability analysis.

Saavedra *et al.* [15] develop a uniprocessor, machine-independent model (the *Abstract Machine Model*) of program execution to characterize machine and application performance, and can predict with good accuracy the running time of a given benchmark on a given machine. However, this work does not consider the effects of the memory subsystem. Saavedra and Smith [14] model memory and TLB costs, but they use published miss ratio data rather than estimating cache miss rates. Ghosh *et al.* [4] introduce Cache Miss Equations (CMEs), a precise analytical representation of cache misses in a loop nest. CMEs have some drawbacks as they also cannot handle multiple loop nests. However, they represent a promising step towards being able to accurately model the cache behaviour of regular array accesses and can be used to further enhance the accuracy of our model.

6 Conclusions and Future Work

The widening processor-memory performance gap makes locality optimizations increasingly important. Optimizations such as loop transformations and array restructuring are effective, but do not have the same mileage in every application. Past studies have attempted to integrate the restructuring optimizations but our work is the first attempt to analytically model the costs and benefits of integration. This paper demonstrates that modeling the memory costs of applications as a whole allows us to compare multiple locality optimizations in the same framework. The accuracy of our cost model is encouraging, given its simplicity. This model can be used as the basis for a wider integration of locality strategies, including tiling, blocking, and other loop transformations.

We also show how hardware support for remapping from an adaptable memory controller can be used to support data restructuring. Such hardware support enables more diverse kinds of data restructuring techniques. In general, we make the case for combining the benefits of software and hardware-based restructuring optimizations in the best possible manner, and provide a framework for reasoning about the combined effects of optimizations.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[2] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proc. of the Fifth HPCA*, pp. 70–79, Jan. 1999.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. TR TR-542, University of Rochester, November 1994.

[4] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Architectural Support for Programming Languages and Operating Systems*, pp. 228–239, 1998.

[5] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *International Symposium on Microarchitecture*, pp. 285–297, 1998.

[6] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pp. 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.

[7] U. Kremer. NP-Completeness of dynamic remapping. In *Proc. of the Fourth Workshop on Compilers for Parallel Computers*, pp. 135–141, Delft, The Netherlands, 1993. (also available as CRPC-TR93330-S).

[8] S. Leung. *Array Restructuring for Cache Locality*. PhD thesis, University of Washington, Aug. 1996.

[9] S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. TR UW-CSE-95-09-01, University of Washington Dept. of Computer Science and Engineering, Sept. 1995.

[10] G. McFarland. *CMOS Technology Scaling and Its impact on cache delay*. PhD thesis, Department of Electrical Engineering, Stanford University, 1997.

[11] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.

[12] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.

[13] V. Pai, P. Ranganathan, and S. Adve. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.

[14] R. H. Saavedra and A. J. Smith. Measuring cache and TLB performance and their effect on benchmark run times. *IEEE Trans. on Computers*, C-44(10):1223–1235, Oct. 1995.

[15] R. H. Saavedra-Barrera. Machine characterization and benchmark performance prediction. TR CSD-88-437, University of California, Berkeley, June 1988.

[16] Scale Compiler Group, Dept. of Computer Science, University of Massachusetts, Amherst. Scale: A scalable compiler for analytical experiments. `http://celestial.cs.umass.edu/Scale/`.

[17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30–44, June 1991.