Cost-Model Driven Integration of Restructuring Optimizations

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

Abstract

Loop transformation and array restructuring are important compiler optimizations that improve memory locality in complementary ways. Although previous researchers have proposed integrating the two techniques, there exists no analytical framework for determining how best to combine them for a given program. In this paper, we propose a cost model for choosing between all combinations of loop and array restructuring options for a given loop nest. Since the choice of which optimization to apply, alone or in combination, is highly application and/or input dependent, such a cost model is crucial if integrated restructuring is to be automated by an optimizing compiler. Our cost model considers two potential forms of array restructuring: conventional copying-based restructuring and remapping-based restructuring that exploits a smart memory controller. We simulate six benchmark programs on a variety of input sizes and with a variety of restructuring optimizations. We find that employing a fixed strategy, e.g., only loop transformations or only copying-based restructuring, does not always deliver the best performance. We further find that for the benchmarks we examine, our cost-model chooses the best combination of restructuring optimizations the vast majority of the time, and yields performance within an average of 10% of optimal across all benchmarks and input sizes.

1 Introduction

Processor and memory speeds have been diverging at the rate of almost 50% a year. At the same time, architectural trends suggest that cache sizes will remain small to keep pace with decreasing processor cycle times. McFarland [11] showed that for a feature size of 0.1 micron and 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. The high latency of memory accesses, increasing CPU parallelism, poor cache utilization and small cache sizes relative to the application working sets create a memory bottleneck that results in poor performance for applications with poor locality. Bridging the growing processor/memory performance gap will require innovative hardware and software solutions that use cache space and memory bandwidth more efficiently.

Iteration space transformations [1, 5, 17] and data layout transformations [9, 3] are two classes of compiler optimizations that improve memory locality of applications. *Loop transformation* is a common iteration space transformation. It improves 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 are increased. This class of optimizations includes loop permutation, fusion, distribution, reversal and tiling [1, 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 [9]. Static restructuring changes the layout of an array at compile time to match the way in which it is most often accessed, e.g., the compiler might store an array in column-major order rather than row-major order if most accesses to the 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 whose layout matches how it is accessed. It is most useful when the access patterns of an array differ throughout the program with no single layout providing optimal performance in all phases, or when the access pattern cannot be determined at compile time. This dynamic change in array layout is most often accomplished by copying, but we also consider the possibility of having smart memory hardware perform the restructuring on the fly [2]. Dynamic array restructuring is more widely applicable than static, and thus we focus on it for our study.

Loop transformation and data restructuring are complementary, and often synergistic, optimizations. Loop transformation, when applicable, improves memory locality with no runtime overhead. However, it is often not possible to improve the locality of all arrays in a particular loop 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 eliminate all accesses with poor locality. 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, data restructuring can always be applied, since it affects only the locality of the target data structure. However, all forms of data restructuring incur some overhead to perform the restructuring, and this overhead cost must be amortized across the accesses with improved locality to be profitable. The two classes of optimizations can be integrated and the best choice depends on which combination of loop and data transformations has the minimum overall cost.

Previous work on integrated restructuring has focused solely on combining static data restructuring and loop transformations [3, 5]. This previous work on integrated restructuring did not provide any means to determine the profitability of any given integrated optimization. In this paper, we present a cost model that captures the cost/performance tradeoffs of each of the above optimizations – loop transformation, traditional array restructuring, and remapping-based restructuring. We use this cost model to decide which

```
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]+W[j][i]+X[i+j+k][k];
      flush_cache(rX);
   }
```

(d) *remap restructured*

Figure 1: An example loop nest and three optimizations.

optimizations to apply, either singly or in combination, for any given loop nest. Code optimized using our cost model achieves performance within 91% of optimal performance for a set of eight benchmarks. In contrast, the performance of any single optimization is at best 68% of optimal.

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 upon our *ir_kernel* example.

2.1 Loop Transformations

Carr *et. al.*[1] choose loop transformations based on a simple cost model. 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*. Applying this cost model to our example loop gives the costs shown in Table 1.

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.

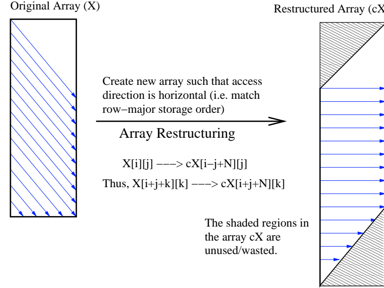

Figure 2: Restructuring an array with a skewed access pattern.

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. Obviously, 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 the new array relate to those of the original [10]. 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.

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 of 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 $X + \mathit{offset} + k * \mathit{stride}$, where *stride* is $N + 1$. The *offset*, which is $(i + j) * N$, is updated each iteration of

the j loop to reflect 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 Integrated Restructuring

In this section, we analyze the tradeoffs both qualitatively and quantitatively to show why it is beneficial to combine the individual restructuring opimizations considered in the previous section. We present an analytic cost model to evaluate the optimizations, and show how it drives an integrated restructuring algorithm.

3.1 A Case for Integrated Restructuring

Loop transformation incurs no run-time costs, and thus it is the optimization of choice when it succeeds in rendering all references in array order. In the presence of conflicting arrays, loop transformations cannot improve the locality of all arrays. In this case, 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 the which array(s) are cheaper to restructure. Array restructuring should be used if the overhead is less than the benefits derived from improved locality, and remapping-based restructuring should be used when the setup cost of creating the array through copying is expected to be larger than the cumulative cost of the remapped accesses.

We analyze the example loop nest in Figure 1(a) to see whether integrated restructuring yields better 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 optimization achieves array order for all references and incurs only the setup cost of creating cX (W is already in array order after loop transformation). In this case, integrated restructuring delivers slightly better performance than either loop or array restructuring alone (see Table 3).

```
double U[N],V[N],W[N][N],X[3N][N];
double cX[4N][N],cW[N][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: The example loop nest, *ir_kernel*, is optimized by a combination of loop transformation and copying-based array restructuring.

The same loop nest can be optimized using loop transformation with remapping-based restructuring, 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, integrated restructuring can lead to new choices of transformations. It is clear that an integrated restructuring strategy can be beneficial, but the questions of what choice to make and when to make the choice have not been addressed. In the next section, we show how the costs and benefits of these restructuring strategies can be modeled analytically, and how this drives the integrated restructuring algorithm.

3.2 Modeling the Restructuring Strategies

3.2.1 Basic Model

We need careful cost/benefit analysis to determine when compiler optimizations may be profitably applied. Finding the best choice via detailed simulation or hardware measurements is time-consuming and expensive. A model that provides a fairly accurate estimate of the cost/benefit tradeoffs between various optimizations is therefore desirable. We have developed an analytic model to estimate the memory costs of applications at compile time. Like Carr, McKinley and Tseng [1], 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 lines 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, then we estimate the memory cost of a single array reference R_α to be:

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

We do not have a framework for estimating f accurately, and so we approximate it to be zero (see the limitations discussed in Section 4.3). For the example in Figure 1(a), the estimated number of cache lines accessed by array X is $\frac{N^3}{max(\frac{cls}{N+1},1)}$, and by array V is $\frac{N}{cls}$.

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 over-counting 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 the the *loop trip count* (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 ith loop nest in the program to be:

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

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:

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

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

3.2.2 Modeling Loop Transformations

The recommendations from our model are the same as those of the model by Carr et al. [1], even though they do not model the total cost of the loop nest. Their approach is adequate if only loop transformations

are to be considered, but it does not work in the presence of array restructuring. We consider the total memory cost of the loop nests, and this allows us to compare the cost of loop transformations with that of independent optimizations such as array restructuring, or of any combination of the optimizations.

3.2.3 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. There is the additional 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.

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

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:

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

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:

$$MemoryCost(copyingSetup) + MemoryCost(restructuredReference) < MemoryCost(originalReference) \quad (6)$$

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 shows that array restructuring is estimated to be always profitable for this particular loop nest. Simulation results bear this out.

3.2.4 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 cache line fills no longer incur the same cost. Cache line fills to remapped (shadow) addresses undergo a further level of translation at

the memory controller, as explained in Section 2.3. After translation, the actual 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 sum of $\#normalCacheLines \times G_c$ and $\#remappedCacheLines \times G_r(stride)$. G_c, the cost of gathering a normal cache line, is fixed, and G_r, the cost of gathering a remapped cache line, is fixed for a given stride.

The cost of remapping-based restructuring is the sum of the cost of setup and the cost of recurring remapped accesses. The initial cost of setting up a remapping through the *map_shadow* call is primarily that of setting up the pagetable to cache virtual-to-physical mappings. The size of the pagetable depends on the number of elements that are to be remapped. We model this cost as $K_1 \times \#elementsToBeRemapped$. We remap one complete run of the innermost loop using the base-stride mechanism. The initial setup cost is typically low but there is the penalty of recurring access costs. We update the remapping information prior to entering the innermost loop every time using the *remap_shadow* system call. This cost, K_2 is usually fixed. We also model the costs of flushing as being K_3 times the number of cachelines flushed. We have empirically estimated these constants using micro-benchmarks. The memory cost of an array reference optimized with remapping support is:

$$MemoryCost(remappedReference) = \left(\frac{loopTripCount}{max(\frac{cls}{stride}, 1)} \times (1 - f) \right) \times G_r(stride) \quad (7)$$

$$MemoryCost(normalReference) = \left(\frac{loopTripCount}{max(\frac{cls}{stride}, 1)} \times (1 - f) \right) \times G_c \quad (8)$$

In particular, to make remapping-based restructuring, array restructuring and loop transformations comparable, we multiply the the memory cost of programs not relying on remapping by a factor of G_c.

We estimated G_r for each stride, and have created a reference database that we consult to determine the cost of gathering a remapped cache line. The memory cost of array rX is $\frac{1}{16}N^3 \times G_r$, where G_r is based on the stride, which here is $(N + 1)$. The total cost of remapping-based restructuring is $(\frac{1}{16}(N^3 + 2N^2) * G_c + \frac{1}{16}N^3 \times G_r + K_1 \times 3N^2 + K_2 \times N^2 + K_3 \times N \times N^2)$. This is again less than the original program's memory cost and thus remapping-based restructuring is profitable.

3.2.5 Integrated Restructuring Algorithm

We now have quantitative performance measures for each of the optimizations. The equations presented in the previous section allow us to decide which optimization wins when, and what the break-even points are. In the absence of hardware support, our comparison is based on the total number of cache lines accessed, since in this case the cost of gathering each cache line is fixed. If hardware support is also considered, then to make the cost comparisons meaningful, we take into account the cost of gathering each cache line. Such a formulation helps us compare the optimizations made with and/or without hardware support.

We present an algorithm to determine the choice between loop transformation, remapping-based restructuring, copying-based array restructuring or a combination of the three. Our algorithm to choose among the possible combinations of transformations: (1) eliminates obviously bad choices and (2) uses the equations of the analytic cost model to choose among potential candidates. Obviously bad choices are those optimizations that render none of the array references in array order: such a transformation will most likely result in a slowdown. There may be many candidates from each of the optimizations: loop transformations(L), array restructuring(C), remapping-based restructuring(R), loop plus array restructuring($L + C$) and loop plus remapping-based restructuring($L + R$). For example, in one of the benchmarks evaluated (*btrix*), there were seven choices for optimization (two choices of L, two of $L + R$, two of $L + C$, and one choice of C).

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 [7]. The problem of finding the optimal data layout between different phases of a program has been proved to be NP-complete [8]. As a result, we need to resort to heuristics or an exhaustive search to choose the best set of transformations to apply. We use heuristics to eliminate obviously bad choices, and then use the mathematical equations derived using the cost model to choose the optimization that has the minimum memory cost. The algorithm does not guarantee that it will find the optimal combination, but we show later in our results that it comes closer to best possible performance than any single 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 [12], a split-transaction MIPS R10000 bus that supports a snoopy coherence protocol, and the Impulse memory controller. 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

For our experiments, we studied eight benchmarks used in previous studies of loop and/or data restructuring. Four of the benchmarks that we considered (*matmult*, a matrix multiplication routine, *syr2k*, a subroutine from the BLAS library, *ex1*, and the *ir_kernel* we have used as an example throughout this paper) have been studied previously in the context of data restructuring – the former two by Leung *et. al.* [9] 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 [1, 6, 9]. Finally, *kernel6* is the sixth Livermore Fortran 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**). *X* indicates the optimizations that were considered, *N* indicates those that were

Application	C	R	L	L+C	L+R
matmult	X	X	X	N	N
syr2k	X	X	I	I	I
vpenta	X	X	X	N	X
btrix	X	I	X	X	X
cfft2d	X	X	I	I	I
ex1	X	X	X	X	X
ir_kernel	X	X	X	X	X
kernel6	X	X	I	I	I

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

not useful, and *I* indicates optimizations that were either illegal or inapplicable, and thus not considered. 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 possible for each input size, and the results for **L+R** is for the optimal combination of loop and remapping-based array restructuring.

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 mix of optimizations to perform for a given input size (**CM-driven**) and the post-facto optimal optimization selection for each input size (**Best**). We refer to optimal performance as that resulting from making the best choices among the optimizations we consider.

As can be seen from Table 3, CM-driven optimization obtains an average of 91.9% of the best possible speedup (2.86 versus 2.98), whereas the best single optimization (in this case, remapping-based array re-

Application	C	R	L	L+C	L+R	CM–driven	Best
matmult	1.56	1.21	2.11	-	-	2.11	2.11
syr2k	1.74	1.77	-	-	-	2.09	2.56
vpenta	0.65	2.13	2.37	-	2.43	2.37	2.66
btrix	2.92	-	4.76	2.90	3.13	4.76	4.76
cfft2d	1.42	1.59	-	-	-	1.61	1.71
ex1	2.79	3.23	1.06	2.41	2.15	3.12	3.90
ir_kernel	3.50	4.87	0.92	3.59	5.27	4.93	5.27
kernel6	1.39	3.46	-	-	-	3.68	3.73
Overall	**1.77**	**2.11**	**1.48**	**1.50**	**1.75**	**2.86**	**3.11**

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.

structuring) obtained only 68.4% of the best possible speedup (2.04 versus 2.98). The reason for the good performance of cost model driven optimization is that the optimal 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 *cfft2d*, the cost-model was able to pick between **R** and **C** and got a higher speedup (2.09) than either **R** or **C** (i.e. it picked **R** when **C** 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 optimal strategy, the choice it made was generally very close to the post-facto optimal 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* was the only benchmark for which loop transformation alone was able to bring all references into array order. While data restructuring can improve the performance of *matmult*, loop restructuring can do so with lower setup costs. Our cost model correctly recognized this situation for all input sizes.

syr2k: The *syr2k* subroutine from the BLAS library computes $C = \alpha A^T B + \alpha B^T A + C$. This banded matrix calculation 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 restructuring, but array restructuring can be quite effective at creating sequential access patterns. The number of accesses to the remapped data is an order of magnitude higher than the size of the arrays. Thus remapping performs well for small bands, where the fixed setup cost of

copying overwhelms the potential benefit, whereas copying performs well for large bands. The cost model correctly identified this behavior in most, but not all, cases. By correctly choosing when to apply copying (**C**) versus remapping (**R**), the cost model driven results achieved better performance than either in isolation.

btrix: The innermost loop was written to be vectorizable, and involves strided accesses across four four-dimensional arrays. There were seven non-obvious optimization candidates (two choices of L+R, two L, one each for L+C, R, C) involving loop fusion, permutation and data restructuring. Since one array had an access pattern conflicting with the other three arrays, loop transformations alone could not convert all references to array order. However, neither array nor remapping-based restructuring was able to improve performance beyond that attainable by pure loop transformation, because the overhead of creating the restructured array was higher than the benefit derived. The cost model successfully predicted the right choice for all the experiments. This was another benchmark in which previous work on array restructuring alone overestimated the potential benefit of such restructuring, by not considering loop transformation.

vpenta: This benchmark accessed eight two-dimensional arrays in seven loop nests with large strides. Loop transformation was useful to optimize the two most expensive loop nests. Even after applying loop transformation, the remaining loop nests had strided accesses. We considered remapping-based restructuring for the remaining array references with strided accesses. In this case, our cost model did not always choose the optimal combination of optimizations. As a result, the CM-driven optimized benchmark performed 2% worse than a fixed choice of **L+R** and 12% worse than the best possible performance. Our cost 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 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.

cfft2d: 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 $O(N^3)$ 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 ($O(N^3)$). Conversely, remapping is preferable for small

data sets. Our cost model correctly predicted this tradeoff, and thus obtained speedups closer to *Best* than either copying- or remapping-based restructuring considered in isolation. Loop-carried dependences precluded us from considering loop transformations for this benchmark.

ex1: In this kernel, only two out of the six possible loop permutations were legal. Loop transformation by itself yielded almost no benefit, but it enabled data transformations that yielded significant speedup. For this benchmark, each of the five optimization candidates was optimal for some subset of the input sizes. Our cost model picked **C** as the best choice while **R** actually had the best performance. The misprediction happens because we do not model the TLB costs. **C** has a high TLB miss rate of 2.31% whereas **R** has a TLB miss rate of 0.02%. The poor TLB usage happens mainly in the setup loop. Remapping incurs no TLB penalty during setup. The cost-model driven optimizations had a slowdown of 3% compared to the best single choice (**R**), which was 80% of the best possible overall performance.

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 as predicted for all cases except one, as the overhead of creating the restructuring was lower in the remapping case. The optimal in this case is the uniformly **L+R** for all data sizes. The misprediction for one case where **L+C** was predicted instead of **L+R** caused a performance penalty of 7% compared to the optimal.

kernel6: This kernel is a general linear recurrence equation solver. The recurrence relation makes loop transformation illegal. For the ten inputs we considered, our cost model correctly predicted the best choice in nine cases: five correct choices of R and four correct choices of C. For the one case, the cost model incorrectly chose C, when R performed better. The performance of array restructuring fluctuated significantly depending on the size and the loop parameters. The CM-driven optimizations decisions were extremely good (within 1% of optimal) compared to array restructuring (37% of optimal) or remapping-based restructuring(7% of optimal).

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 9% of optimal, whereas the best performance from any single optimization choice was 32% less than optimal.

4.3 Caveats

Our cost model 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. 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. Carr, McKinley and Tseng [1] 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 develop a profitability analysis to determine when the optimization should be applied.

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 applicability 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. This work is complementary to ours 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. Restructuring optimizations are effective, but do not have the same mileage in every application. There have also been past studies that have attempted to integrate the restructuring optimizations. Our work is the first attempt to analytically model the costs and benefits of the integration. This paper demonstrates that modeling the memory costs of applications as a whole allows us to compare various locality optimizations in the same framework. The accuracy of the 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 enables a new data restructuring optimization. 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] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. of the 6th ASPLOS*, pp. 252–262, Oct. 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 *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, Oct. 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[5] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated approach. In *Proc. of IEEE/ACM 31st International Symposium on Microarchitecture*, Dec. 1998.

[6] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. of the ICS (ICS-98)*, pp. 69–76, New York, July 13–17 1998. ACM press.

[7] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 301–320, Portland, Oregon, Aug. 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

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

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

[10] 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.

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

[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/index.html`.

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