

A Cost Model For Integrated Restructuring Optimizations

Bharat Chandramouli

Verplex Systems, Inc.
300 Montague Expressway #100
Milpitas, CA 95035

Wilson C. Hsieh and John B. Carter

School of Computing
University of Utah
Salt Lake City, UT 84112

Sally A. McKee

Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853

Compilers must make choices between different optimizations; in this paper we present an analytic cost model that compares several compile-time optimizations for memory-intensive, matrix-based codes. These optimizations increase the spatial locality of references to improve cache hierarchy performance. Specifically, we consider loop transformations, array restructuring, and address remapping, as well as combinations thereof. Our cost model chooses among these optimizations and gives a good decision about which optimization to use.

To evaluate the cost model and the decisions taken based on it, we simulate eight applications on a variety of input sizes and with a variety of manually applied restructuring optimizations. We find that always choosing a single, fixed strategy delivers suboptimal performance, and that it is necessary to adjust the chosen optimization to each code. Our model generally predicts the best combination of restructuring optimizations among those we examined. It chooses a set of application optimizations that yields performance within a geometric mean of 5% of the best combination of candidate optimizations, regardless of the benchmark or its input dataset size.

1 Introduction

Compilers use cost models to choose among optimization strategies. If cost models are not available, choices must be heuristic or ad hoc. In this paper, we present a cost model that helps choose among combinations of code and data restructuring optimizations to increase memory locality, thereby improving utilization of current multi-level memory subsystems. Better memory hierarchy performance translates into better execution times for applications that are latency and bandwidth-limited, particularly scientific computations containing complex loops that access large data arrays.

The optimizations we address focus on array-based applications, and they improve cache locality by transforming the iteration space [4, 15, 33] or by changing the data layout [10, 22]. These *loop transformations* and *array restructuring* techniques can be complementary and synergistic. Combining these optimizations is therefore often profitable, but exactly which set of transformations performs best depends on which of the variations has the minimum overall cost on a particular application. We use *integrated restructuring* to refer to the ability to combine legal, complementary code and data restructuring optimizations, as well as the ability to choose a good combination from the legal options for a given program and input set.

Other researchers integrate data restructuring with loop transformations [10, 15], but they only consider static array restructuring, and do not address profitability-analysis mechanisms for the various possible choices. Their efforts provide a foundation on which to base our research, but combine restructuring optimizations in an ad hoc manner. This paper explores the feasibility of using a cost model to guide the compiler's choices among various restructuring techniques. Specifically:

- We demonstrate via simulation that integrated restructuring is not always a win, and that it should be applied carefully.
- We introduce *remapping-based array restructuring*, an optimization that exploits hardware support for remapping from a smart memory controller.

This paper is an expanded version of a conference paper [7].

```

double A[N][N],B[N][N],C[N][N];
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      C[i][j] = C[i][j] + A[i][k]*B[k][j];

```

Figure 1. *matmult* Kernel

- We present a *cost model* that incorporates the costs/benefits of hardware support from a smart memory controller, which enables appropriate decisions as to whether hardware support should be used.
- We model the memory cost of whole applications, not just individual loop nests, and we have developed an *integrated analytic framework* for reasoning about hardware/software tradeoffs in restructuring optimizations.

Programs optimized based on the results of our model achieve performance within 5% of the best observed for our set of eight benchmarks. In contrast, the performance of any fixed optimization is at best 24% of the best combination.

In Section 2, we present an overview of existing program restructuring optimizations and describe remapping-based array restructuring, a new memory controller-based locality optimization. In Section 3, we develop the analytic framework to estimate costs and benefits for individual restructuring optimizations. We explain the assumptions built into our framework and discuss how they affect performance estimation. In Section 4, we describe our simulation methodology, benchmark suite, and experimental results. Details for individual benchmarks are described in the Appendix. In Section 5, we compare our cost-model driven optimizations to existing integration strategies, emphasizing the differences in strategies and the resulting tradeoffs. In Section 6, we discuss extending this initial cost model to address a broader class of applications with greater accuracy, and in Section 7, we summarize our conclusions.

2 Background

We briefly review the compiler restructuring optimizations on which this work is based: loop transformations, the standard copying-based form of array restructuring, and a new version of array restructuring based on hardware remapping. The loop transformations we consider are permutation, fusion, distribution, and reversal. Together with tiling, these are the most common examples of loop restructuring. Given the complexity of calculating cross-interference and self-interference misses in complex loop nests, we omit loop tiling from the analytic models developed in this study.

In addition to evaluating the costs and benefits of the traditional implementation of this optimization, we also consider implementing dynamic data restructuring with hardware support. Run-time changes in array layout have heretofore been implemented by physically moving data. Our work on the Impulse memory controller [5, 32, 36] shows how hardware support for address remapping can be used to change data structure layout. The tradeoffs involved in remapping optimizations differ from those for traditional copying-based array restructuring, and our model is able to reflect this.

2.1 Loop Transformations

Loop transformations improve performance by changing the execution order of loops in a nest, transforming the iteration space to increase the temporal and spatial locality of a majority of data accesses [4, 15, 33]. These compile-time transformations incur no run-time overhead, but they cannot always improve the locality of *all* arrays in a given 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 exists to 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 nests. Even when the optimization can legally be applied, it may fail to increase locality for enough data accesses in the nest.

To see a simple example of the potential impact of loop permutation, consider the matrix multiplication code shown in Figure 1. Array B is accessed with a stride of $8 \times N$ bytes, which yields low locality of reference: all but one element of B loaded with any cache line are evicted without being used. For a 32-Kbyte L1 cache with 32-byte lines and eight-byte array elements, only 25% of each cache line is referenced. In addition, cache performance degrades even more if the length of a column walk exceeds 1024. Similarly, this program fragment exhibits poor TLB performance, even for small values of N . To see why, assume a TLB of 128 entries, where each maps a four-Kbyte page. In one column

array references	innermost loop		
	i	j	k
A[i][k]	$N \times N^2$	$1 \times N^2$	$\frac{1}{16} N \times N^2$
B[k][j]	$1 \times N^2$	$\frac{1}{16} N \times N^2$	$N \times N^2$
C[i][j]	$N \times N^2$	$\frac{1}{16} N \times N^2$	$1 \times N^2$
total	$2N^3 + N^2$	$\frac{1}{8}N^3 + N^2$	$\frac{17}{16}N^3 + N^2$

Table 1. Carr, McKinley, and Tseng’s cost model applied to matrix multiply

walk, the code touches a memory span of $8 \times N^2$ bytes, or $\frac{8 \times N^2}{4096}$ pages. When $\frac{8 \times N^2}{4096} > 128$ (i.e., $N > 216$), a TLB miss is taken for every traversal of a column. Permuting the loops from ijk to ikj order yields zero or unit stride for all arrays. The row-major access patterns drastically reduce the TLB and cache footprints. For large arrays, the performance speedup can be significant.

Carr *et al.* [4] analyze loop transformations using a simple cost model based on the estimated number of cache lines accessed within a nest. They define a given *loop cost* to be the estimated number of cache lines accessed by all arrays when that loop is placed innermost in the nest. Evaluating all loop costs and ranking the loops in descending cost order (subject to legality constraints) yields a permutation with least cost. Consider the matrix multiply loop nest in Figure 1; calculating loop costs gives the results in Table 1. Descending cost order is $i > k > j$, and thus the recommended permutation is ikj . A , B , and C are now accessed sequentially. This model provides the basis for the model we develop in Section 3.

2.2 Traditional Array Restructuring

Array restructuring transforms the physical data layout to improve cache performance [22] either at compile time or at run time. Static (compile-time) array restructuring changes the array layout to match the direction in which the data are most often accessed; e.g., the compiler stores an array in column-major rather than row-major order when column walks dominate. If the array is accessed in the same manner throughout the program’s execution, such static restructuring can dramatically improve performance.

In this paper we do not compare against static array restructuring. If array access patterns change at run time, or if the patterns cannot be determined at compile time, static array restructuring is not feasible. Instead, dynamic restructuring can be used at run time to create new data structures that have better locality.² The “dynamic” part of the restructuring is the run-time change in the application’s view of an array. In the remainder of the paper, we shall use “array restructuring” to mean “dynamic array restructuring”.

Unlike loop restructuring, array restructuring affects only the locality of a single target array, as opposed to all arrays in a nest. Static restructuring incurs no run-time costs, but dynamic restructuring incurs run-time overhead costs; dynamic transformations are therefore only profitable when these costs are dwarfed by the improved latency of the high-locality, restructured accesses. Nonetheless, dynamic restructuring is more versatile, and more widely applicable, than loop restructuring.

Many scientific applications contain non-unit stride accesses to multi-dimensional arrays. Array restructuring can directly improve the array layouts in memory. Consider the example loop nest *ir.kernel* in Figure 2. With row-major storage, arrays U and V are accessed sequentially, where U enjoys spatial locality, and V enjoys temporal locality. Array W enjoys temporal locality in the innermost loop, but is accessed along columns in the outer loops. Array X is accessed diagonally, and will suffer poor cache and TLB performance. Array restructuring creates new arrays cX and cW , where cW is W transposed, and cX is a diagonal layout of X (as in Figure 3).

Transformation matrices specify how new indices in cX and cW relate to original array indices. Leung and Zahorjan derive formalisms to create index transformation matrices [23]. In this case, 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 4 shows the code with copying-based array restructuring. To keep the example simple, we do not tile the copying phase, but tiling would improve performance. As illustrated by the shaded regions of the restructured array depicted in Figure 3, this optimization may waste storage. In this case, cX contains $4N^2 - 3N^2 = N^2$ extra, unused elements. Leung and Zahorjan show that

²The code to implement array restructuring optimizations—even those we call dynamic—can be generated at compile-time, or even at run-time by systems such as Dynamo [2].

```

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];

```

Figure 2. *ir_kernel* code

further transformations can reduce the unused memory. Their algorithm bounds the restructured array size to at most $N!$ times that of the original, with N being the array dimensionality [23].

The innermost loop now accesses cX with unit stride, and the outermost loops access cW with unit stride. Spatial locality of reference in the transformed code is thus much greater than in the original. Nonetheless, whether the transformation improves performance depends on the input parameters. If an array is large and rarely accessed, the cost of creating the new array might dominate the benefits of improved spatial locality. In Figure 4, it is unlikely that copying W is worthwhile, since no cache misses are saved; they are merely moved from the main loop to the copying loop. In Section 3.1.2, we develop a formula to decide when array restructuring can be used profitably.

2.3 Remapping Array Restructuring

With appropriate hardware support, array restructuring can be implemented without copying. Instead, a new alias array can be created *virtually* by remapping addresses. Remapping-based array restructuring is similar in spirit to copying-based array restructuring. The difference is in its implementation, which depends on the ability to remap physical memory.

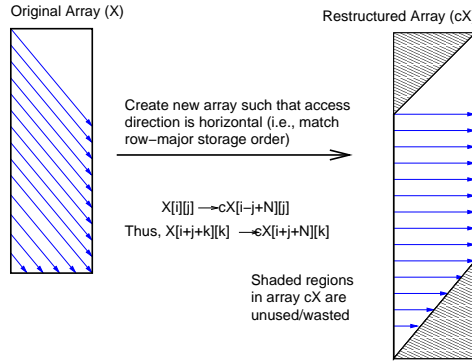


Figure 3. Visualizing copy restructuring of *ir_kernel*'s X array

```

double U[N], V[N], W[N][N], X[3N][N], cX[4N][N], cW[N][N];
// copy W into its transposed version cW
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        cW[j][i] = W[i][j];
// copy X into its transposed version cX
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];

```

Figure 4. *ir_kernel* optimized via copying-based restructuring

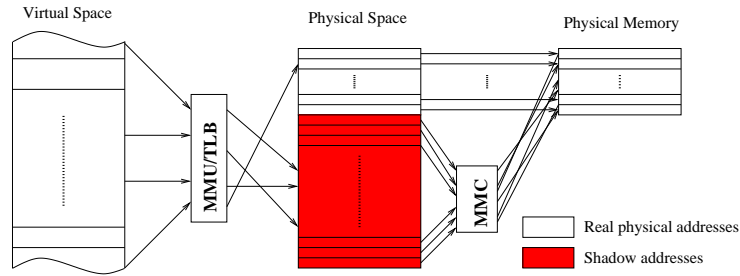


Figure 5. Address remapping in an Impulse system

Our study assumes that remapping-based data restructuring is implemented with an intelligent memory controller. The Impulse memory system expands the traditional virtual memory hierarchy by adding address translation hardware to the main memory controller [5, 32, 36]. Impulse takes physical addresses that would be unused in conventional systems and uses them as remapped aliases of real physical addresses. For instance, in a system with 32-bit physical addresses and one Gbyte of installed DRAM, physical addresses in the range $[0x40000000 - 0xFFFFFFFF]$ normally would be considered invalid. We refer to such otherwise unused physical addresses as *shadow addresses*.

Figure 5 shows how addresses are mapped in an Impulse system. The real physical address space is directly backed by physical memory; its size is exactly the size of installed physical memory. The shadow address space does not directly point to any real physical memory (thus the term *shadow*), and can be remapped to real physical addresses through the Impulse memory controller. This virtualization of unused physical addresses provides different views of data stored in physical memory. For the *ir_kernel* code, the Impulse controller can create pseudo-copies of W and X that function much the same as cW and cX . The difference is that the hardware fetches the correct elements from W and X when the pseudo-copies are accessed; no copying is performed. Gathering elements within the memory controller allows the data to be buffered so that elements are transmitted over the bus only when requested, which consumes less bus bandwidth and uses processor cache space more efficiently. The operating system manages all of the resources in the expanded memory hierarchy and provides an interface for the application to specify optimizations for particular data structures. The programmer or compiler inserts appropriate system calls into application code to configure the memory controller.

Consider the following example, in which the application creates a virtual transpose of a two-dimensional array A . The application makes a system call to allocate a contiguous range of virtual addresses large enough to map a transposed version of A , the virtual address of which is returned in rA . System call parameters include the original array’s starting virtual address, element size, and dimensions. The OS allocates a range of shadow physical addresses to contain the remapped data structure, i.e., the transposed array. It then configures Impulse to respond appropriately to accesses within this shadow physical address range.

Configuring Impulse takes two steps: setting up a dense page table and initializing a set of control registers. The OS uses I/O writes to these registers to indicate remapping-specific information such as what address range is being configured, what kind of remapping Impulse should perform (in this case, matrix transposition), the size of the elements being remapped, and the location of the corresponding page table. After configuring Impulse, the OS maps the shadow physical addresses to an unused portion of the virtual address space and places a pointer to it in rA . Finally, the user program flushes the original array from the cache to prevent improper aliasing.

When the user process accesses a remapped address in the virtual transpose array, the processor TLB or MMU converts this virtual address to the corresponding shadow physical address. Impulse translates the shadow address into the real physical addresses for the data that comprises that shadow cache line, and then passes these physical addresses to an optimized DRAM scheduler that orders and issues the reads, buffers the data, and sends the gathered data back to the processor. The controller contains an address calculation unit, a large cache to buffer prefetched data, and its own TLB, which caches entries from the dense page table created by the OS during the system call that sets up the remapping.

We use both transpose and base-stride remapping to implement the array restructuring in this study. Base-stride remapping is more general than transposition, and is used to map a strided data structure to a dense alias array. For instance, consider a set of diagonal accesses to be remapped, as with X in the loop in Figure 2. Each access has a fixed stride from the start of the diagonal, but for each diagonal, the base address differs. Before accessing each new diagonal, the application issues a system call to specify a new base address to Impulse.

```

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);
    }

```

Figure 6. *ir_kernel* optimized via remapping restructuring

Figure 6 shows how the memory controller is configured to support remapping-based array restructuring for *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 to reflect the new values of *i* and *j* — i.e., the new diagonal being traversed. Thus, $rX[k]$ 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.

After applying the remapping-based restructuring optimization, all accesses are in *array order*, i.e., their access patterns match their element storage order. The cost of setting up a remapping is small compared to that of copying. However, subsequent accesses to the remapped data structure are slower than accesses to an array restructured via copying, because the memory controller must translate addresses on-the-fly and gather data from disjoint regions of physical memory. The amount of data reuse in a loop nest directly influences the overhead of using remapping-based array restructuring. Remapping is preferable to copying when the cumulative recurring access costs of remapping are less than the one-time setup cost of copying. In general, a cost/benefit analysis is necessary to decide whether to apply data restructuring, and if so, whether to use remapping versus a copying-based implementation.

3 Analytic Framework

A compiler first chooses part of the program to optimize and a particular optimization to apply. Then it transforms the program and verifies that the transformation does not change the meaning of the program (or at least that it changes it in a way that is acceptable to the user). Bacon *et al.* [1] consider the first step a “black art”, since it is difficult and not well understood. In this section, we explore an efficient mechanism that lets the compiler choose the code portions to optimize for the domain of restructuring optimizations. We present cost models for each optimization and analyze tradeoffs among individual restructuring optimizations, both qualitatively and quantitatively. We show why it is beneficial to consider combining optimizations, and present a framework that aids in evaluating the costs and benefits of integrated restructuring optimizations. Our method does an exhaustive search of the optimization design space to find the optimal combination. This approach is feasible because we provide a cost model that is inexpensive to compute, and we limit the number of optimization-combinations that we consider.

3.1 Modeling Restructuring Strategies

Finding the best choice of optimizations to apply via detailed simulations or hardware measurements is slow and expensive. An analytic model that provides a sufficiently accurate estimate of the cost/benefit tradeoffs between various optimizations makes choosing the right strategy much easier. We have developed such an analytic model to estimate the memory cost of applications at compile time. Like Carr *et al.* [4], we estimate the number of cache lines accessed within loop nests, and then use this estimate to choose which optimization(s) to apply.

The memory cost of an array reference, R_α , in a loop nest is directly proportional to the number of cache lines it accesses. Consider the line that references *B* in the loop nest in Figure 7. The array is accessed sequentially *N* times. If the cache line size is 128 bytes and a “double” is eight bytes, then the number of lines accessed by that line is $\frac{N}{16}$, and the memory cost of the array references is thus proportional to this number.

```

double A[N+1], B[N], C[3N][N];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      A[i+1] = A[i] + B[i] + C[i+j+k][k];

```

Figure 7. Example loop to illustrate cost model

Let cls be the line size of the cache closest to memory, $stride$ be the distance between successive accesses of array reference R_α in a nest, $loopTripCount$ be the number of times that the innermost loop whose stride is non-zero is executed, and f be the fraction of cache lines reused from a previous iteration of that loop. The memory cost of array reference R_α in the loop nest is estimated as:

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

Unfortunately, we do not have a framework for accurately estimating f , but in most cases we expect f to be very small for large working sets. For the benchmarks that we examine here, the working sets are indeed large, and data reuse is very low. We therefore approximate f as zero, and we may neglect the $(1 - f)$ term without introducing significant inaccuracies (see the limitations discussed in Section 3.2). Using Equation 1, the memory cost of array reference C in Figure 7 is $\frac{N^3}{\max(\frac{cls}{N+1}, 1)}$. If we assume that $N > 16$, then the cost is N^3 .

We estimate the memory cost of the whole 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 distinction is necessary to avoid counting some lines twice. For example, references $A[i]$ and $A[i+1]$ are not independent, because we assume that they access the same cache line. In contrast, references $A[i]$ and $B[i]$ are independent, because we assume they access different cache lines. The cost of a loop nest depends on the *loop trip count* (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:

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

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

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

Using these equations, the total memory cost of the loop in Figure 7 is $\left(N^3 + 2 \times \frac{N}{cls} \right)$.

The cost model's goal is to compute a metric for choosing the combination of array and loop restructuring optimizations with 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 – that the compiler knows the number of times each loop nest executes, and that it can calculate the loop bounds of each loop nest at compile time. This assumption holds for many applications. In cases where this does not hold, other means must be used to estimate the frequencies of loop nests (e.g., profile-directed feedback). We also assume that the line size of the cache closest to memory is known to the compiler.

3.1.1 Modeling Loop Transformation

When we consider only loop transformations, the recommendations from our model usually match those of the simpler model by McKinley *et al.* [24]. However, their model offers no guidance to drive data restructuring or integrated restructuring. We consider the total memory cost of the loop nests, allowing us to compare the costs of loop transformations with those of independent array restructuring optimizations as well as of combinations of restructuring optimizations.

3.1.2 Modeling Copying-based Array Restructuring

The cost of copying-based array restructuring is the sum of the cost of creating the new array, the cost of executing the optimized loop nest, and the cost of copying modified data back to the original array. The setup cost equals the sum of the memory costs of the original and new arrays in the setup loop.

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

In the setup loop of the code shown in Figure 4, the memory cost for writing cX is $\frac{3N^2}{\max(\frac{\text{cls}}{N+1}, 1)}$, and the memory cost for reading X is $\frac{3N^2}{\text{cls}}$. The total setup cost is $3N^2 \times (1 + \frac{1}{\text{cls}})$ if $(N + 1) > \text{cls}$ (the usual case). The calculation for cW is similar.

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

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

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

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

For this example, the total cost of the array-restructured program (assuming $\text{cls} = 16$) 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 , so our cost model will estimate that array restructuring will always be profitable for this particular loop nest. Simulation results bear out this decision for arrays exceeding the $L2$ cache size.

3.1.3 Modeling Remapping-based Array Restructuring

When using remapping-based hardware support, we can no longer model the memory cost 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 this cache line. The cost of gathering a normal cache line, G_c , is assumed to be fixed, and the cost of gathering a remapped cache line, G_r , is assumed to be fixed for a given stride. We use a series of simulated microbenchmarks with the Impulse memory system to build a table of G_r values indexed by stride. This table, a pictorial view of which is given in Figure 8, determines the cost of gathering a particular 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:

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

The overhead costs involved in remapping-based array restructuring include the cost of remapping setup, the costs of updating the memory controller, and the costs of flushing the cache. The initial overhead 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 to be remapped. We model this cost as $K_1 \times \#elementsToBeRemapped$. Updating the remapping information via the `remap_shadow` system call prior to entering the innermost loop incurs a fixed cost, which we model as K_2 . We model the flushing costs as proportional to the number of cache lines evicted, where the constant of proportionality is K_3 . We estimate these constants via the simulation of microbenchmarks.

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

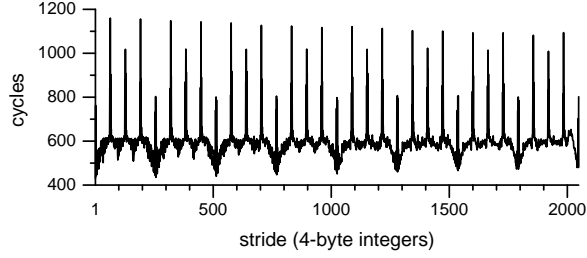


Figure 8. Relationship between G_r and stride

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

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

Returning to the example in Figure 6, the memory cost of array rX is $\frac{1}{16} \times N^3 \times G_r$, where G_r is based on the stride, which here is $(N + 1)$. The total cost of the remapping-based array-restructured loop is $(\frac{1}{16}(N^3 + N) * G_c + \frac{1}{16}N^3 \times G_r(N + 1) + K_1 \times 3N^2 + K_2 \times N^2 + K_3 \times N \times N^2)$. This cost is less than the original program’s memory cost, and so our model deems remapping-based array restructuring to be profitable.

The cost of gathering a remapped cache line, G_r , is influenced by two factors: the DRAM bank organization and the hit rate in the memory controller’s TLB. See the first author’s thesis [6] for details of the microbenchmark used to estimate G_r .

Figure 8 shows how G_r varies with the stride (in bytes). The spikes occur whenever the stride is a power of two greater than 32, where the remapped accesses all reside in the same memory bank. We calculate G_r for strides ranging from four to 8192 bytes. The average value of G_r is 596 cycles, with the range spanning 429 to 1160 cycles. Since G_r varies significantly with stride, our cost model uses a table to look up G_r for each array.

3.2 Caveats

Our cost model has a number of known inaccuracies; improving its accuracy is part of ongoing 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 is 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 subsystem have less impact on program performance than cache misses that are distributed in time, but our model gives them equal weight. Third, we do not consider cache reuse (i.e., we estimate 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 overestimates the memory costs and recommends an unnecessary optimization. Similarly, not modeling cache interference can result in the model’s underestimating memory costs if there are significant conflict misses in the optimized applications. It should be noted, though, that the goal of this cost model is not to accurately predict the exact performance of the code under investigation, but is rather to provide a fast basis for deciding which optimizations are likely to result in performance improvements. It is therefore an open question to which degree a more accurate modeling is necessary for this task.

In general, selecting the optimal set of transformations is an extremely difficult problem. Finding the combination of loop fusion optimizations alone for optimal temporal locality has been shown to be NP-hard [18]. The problem of finding the optimal data layout between different phases of a program has also been proved to be NP-complete [19]. Researchers therefore use heuristics to make integrated restructuring tractable. Our analytic cost framework can help evaluate various optimizations, and optimization strategies driven by our cost model come closer to the best possible performance across a range of benchmarks than any fixed optimization strategy.

	L1 cache	L2 cache
size of cache	32 Kbytes	512 Kbytes
cache line size	32 bytes	128 bytes
associativity	2-way	2-way
access latency	1 cycle	8 cycle
index/tagged	virtual/physical	physical/physical
mode	write-back	write-back
	non-blocking	non-blocking

Table 2. Configuration of the memory subsystem

4 Evaluation

In this section, we briefly describe our experimental framework, including the simulation platform and benchmarks used. The Appendix contains details on the composition and behavior of each benchmark; we summarize salient features here. We then discuss the relative performance of the various combinations of optimizations, and define several metrics to evaluate how well our cost model recommends appropriate choices.

4.1 Simulation Platform

We use URSIM [34] to evaluate the effectiveness of our cost models. This execution-driven simulator (derived from RSIM [29]) models a microprocessor similar to the MIPS R10000 [26], with a split-transaction MIPS R10000 bus and snoopy coherence protocol. The processor is superscalar (four-way issue) and out-of-order, with a 64-entry instruction window. This base system has been extensively validated against real hardware [35] and has demonstrated high accuracy.

The memory system itself can be adjusted to model various target systems. The cache configuration used for this study, shown in Table 2, models that of a typical SGI workstation. The instruction cache is assumed to be perfect, but this assumption has little bearing on our study of data optimization techniques. The TLB maps both instructions and data, has 128 entries, and is single-cycle, fully associative, and software-managed.

The eight-byte bus multiplexes addresses and data, 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 implements a critical-word-first policy, returning the critical quad-word for a normal cache-line load 16 bus cycles after the corresponding L2 cache miss. The main memory system models eight banks, pairs of which share an eight-byte bus between DRAM and the the Impulse memory controller.

4.2 Benchmark Suite

To evaluate our cost-model driven integration, we study eight benchmarks used in previous loop or data restructuring studies. Four of our benchmarks—*matmult*, *syr2k*, *ex1*, and the *ir_kernel* discussed in Section 2.2—have been studied in the context of data restructuring. The former two are used by Leung *et al.* [22], and the latter two by Kandemir *et al.* [15]. Three other applications—*btrix*, *vpenta*, *cfft2d*—are NAS kernels used to evaluate data and loop restructuring in isolation [4, 16, 22]. Finally, *kernel6* is the sixth Livermore Fortran kernel, a traditional microbenchmark within the scientific computing community.

Table 3 shows which optimizations are suitable for each benchmark. The candidates include copying-based array restructuring (**ac**), remapping-based restructuring (**ar**), loop transformations (**lx**), a combination of loop and copying-based restructuring (**lc**), and a combination of loop and remapping-based restructuring (**lr**). A checkmark \checkmark indicates that the optimization is possible, *N* indicates that the optimization is not needed, and *I* indicates that the optimization is either illegal or inapplicable. In this study, we hand code all optimizations, but automating them via a compiler [14], binary rewriting system [21], or run-time optimization system [2] would be straightforward. We run each benchmark over a range of input sizes, with the smallest input size typically just fitting into the L2 cache. Whenever there are several choices for a given restructuring strategy, we choose the best option. In other words, the results we report for **lx** represent the best loop transformation choice (loop permutation, fusion, distribution or reversal) among the ones we evaluate. Similarly, the results for **lr** are for the best measured combination of optimizations. The Appendix

Application	ac	ar	lx	lc	lr
<i>matmult</i>	✓	✓	✓	N	N
<i>syr2k</i>	✓	✓	N	N	✓
<i>vpenta</i>	✓	✓	✓	N	✓
<i>btrix</i>	✓	I	✓	✓	✓
<i>cfft2d</i>	✓	✓	I	I	I
<i>ex1</i>	✓	✓	✓	✓	✓
<i>ir_kernel</i>	✓	✓	✓	✓	✓
<i>kernel6</i>	✓	✓	I	I	I

✓ = Optimization possible
I = Optimization illegal/inapplicable
N = Optimization not needed

Table 3. Benchmark suite and candidates for optimizations

gives more details of the experiments conducted using each benchmark and includes the relevant source code for each baseline version.

4.3 Cost Model Results for *btrix*

We now generate the individual cost formulas for each benchmark and for each optimization or combination thereof. Table 4 shows these formulas for one example code, *btrix*.³ They model both the memory access costs and the setup costs for the data transformations, and hence provide a complete assessment of all estimated memory-related costs for the complete benchmark execution.

Figure 9 plots these cost estimates for various strides, which gives an overview of the decision space. All graphed values represent relative speedups compared to the original benchmark. The graph clearly shows the differences in expected performance of the various optimizations, and illustrates the constant relationship among the individual optimizations for large strides. As N grows, the ratio between the cubic terms dominates all other terms. Loop restructuring (**lx**) is therefore expected to outperform all other optimization combinations. In the following section, we validate this against simulated *btrix* performance.

Note that the absolute values of the memory costs represented in in the graph do not reflect the real, whole-program speedups to be expected, but they correctly indicate the compiler’s desired ordering of the various optimizations.

4.4 Simulation Results

We summarize the results of our experiments in Figure 10. For each input size, we simulate the original benchmark and the best version for each candidate optimization (**ac** for array copying, **ar** for array remapping, **lx** for loop restructuring, and **lc** or **lr** for integrating loop restructuring and array restructuring via copying or remapping, respectively). We also compute which version the cost model **cm** selects for a given input size. The cost model data are generated using the appropriate values of G_r (depicted in Figure 8) to choose among optimization options. Our primary performance metric is the geometric mean speedup obtained for each optimized version over the range of input sizes. We compare the speedups of each version with the post-facto best set of candidate optimizations **b** for a given benchmark/input combination.

The results in Figure 10 show that using our cost model delivers an average of 95% of the best observed speedups (1.68 versus 1.77), whereas the best single optimization, copying-based array restructuring, obtains only 77% (1.35

³Details on how to compute these formulas are included in the appendix of the first author’s thesis [6].

Optimization Candidate	Total Memory Cost of Program
base	$(157.78N^3 + 2.34N^2)G_c$
lx	$(43.90N^3 + 84N^2)G_c$
ac	$(43.40N^3 + 111.375N^2)G_c$
lc	$(23.62N^3 + 85N^2)G_c$
lr	$(40.03N^3 + 84N^2)G_c + 107N^2 + 0.156N^3G_r(N^2)$
lx'	$(14.84N^3 + 84N^2)G_c$
lr'	$(36.75N^3 + 84N^2)G_c + 107N^3 + 0.156N^3G_r(N^3)$

Table 4. Total memory costs for the various *btrix* optimizations

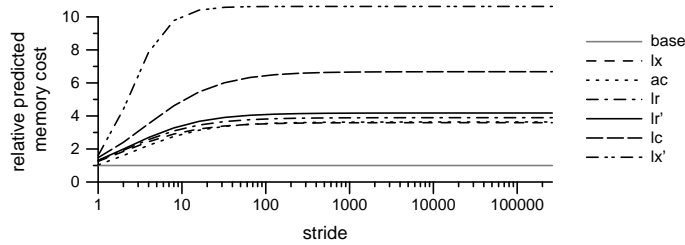


Figure 9. Computed relative memory costs for increasing stride

versus 1.77). Even within the same benchmark, the best optimization choice is dependent on input data size. The cost-model strategy therefore performs well because it adapts its choices to each application/input. For example, in *syr2k*, the cost-model is able to choose between **ac** and **lr** for different inputs, achieving a higher speedup (1.88) than using either **ac** or **lr** for all runs (i.e., it picks **ac** when **lr** performs poorly, and vice-versa). Overall, our cost model is usually able to select the correct strategy, and when it fails to pick the best strategy, the choice is still generally very close to **b**. To better understand why the cost model works well in most cases, albeit poorly in a few, we examine each benchmark in turn in the Appendix.

No single optimization performs best for all benchmarks. Examining which optimization is the most beneficial for the maximum number of benchmarks reveals a four-way tie between **ac**, **ar**, **lx** and **lr** (*syr2k* and *ex1* vote for **ac**, *cfft2d* and *kernel6* for **ar**, *matmult* and *btrix* for **lx**, and *vpenta* and *ir_kernel* for **lr**). Thus, it makes sense to choose optimizations based on benchmark/input combinations, rather than to blindly apply a fixed set of optimizations.

Combining loop and array restructuring can result in better performance than either individual optimization achieves. For example, in *vpenta*, loop permutation optimizes the two most expensive loop nests, but the five remaining nests still contain strided accesses that loop transformations alone cannot optimize. Combining loop permutation with remapping-based array restructuring results in a speedup higher than either optimization applied alone. The cost model predicts this benefit correctly.

Nonetheless, combining restructuring optimizations is not always desirable, even when they do not conflict. For example, in *btrix*, loop restructuring is able to bring three out of four four-dimensional arrays into loop order. The remaining array can be optimized using either remapping-based or copying-based array restructuring. Unfortunately, adding the array restructuring optimization degrades performance, because the overhead of array restructuring is higher than its benefits. The cost model correctly recognizes this and chooses pure loop restructuring.

Within a single benchmark, the choice of what optimization to apply differs depending on the loop and array characteristics. For example, in *kernel6* remapping and copying are both beneficial and resulting in good speedups. However, the cost model achieves an even higher speedup than either of these by choosing remapping when copying is expensive and vice-versa.

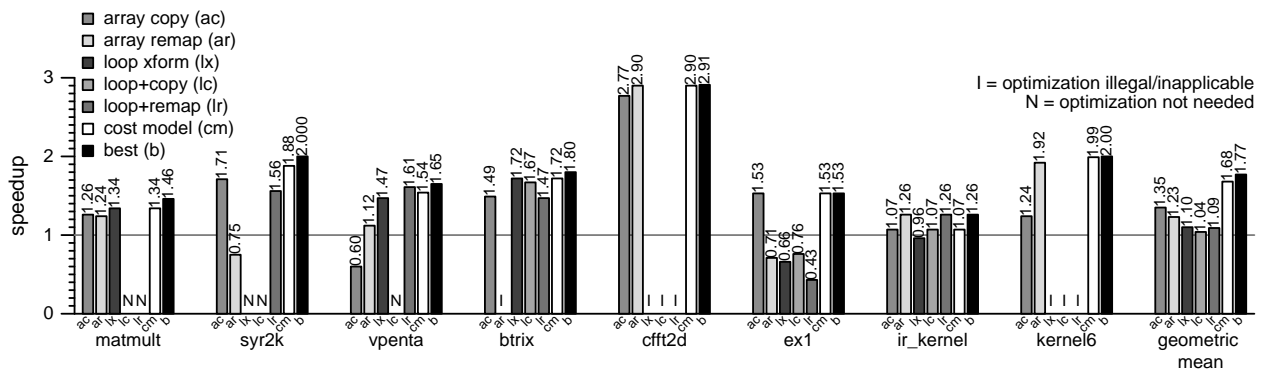


Figure 10. Mean optimization performance

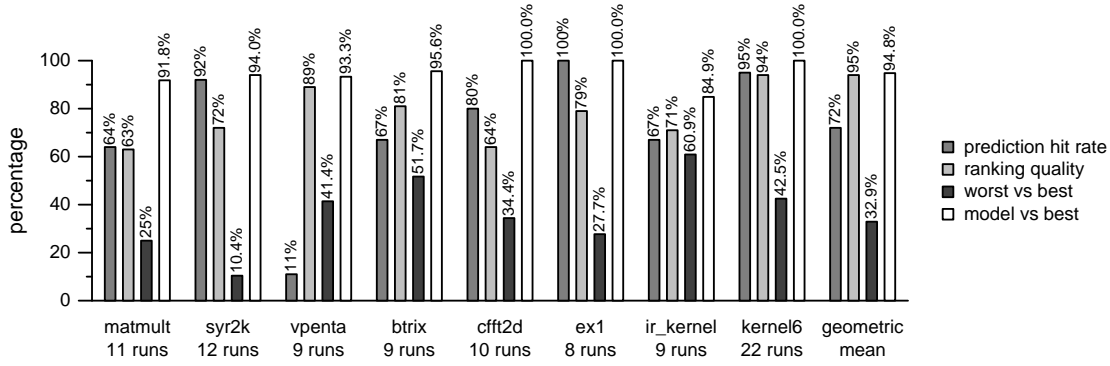


Figure 11. Evaluating the cost model

4.5 Model Evaluation

To evaluate the accuracy of our cost model quantitatively, we employ four metrics:

- *Best prediction hit rate.* This metric calculates the cost model’s success in predicting the best optimization from among the given choices over a large range of different input data set sizes.

The prediction success rate for the cost model across all benchmarks is 72%, on average. This is impressive, considering that if choices were made completely at random, the prediction success rate would only be 22%. Although this metric is fairly indicative of the accuracy of the cost model, it is somewhat conservative. It fails to distinguish between “close second” and “dead last”. For example, in *vpenta*, although the cost model’s choice of **lx** was incorrect 89% of the time, **lx** is slower than the best choice of **lr** by only 5%. In contrast, **ac** is slower by 58%, but either choice yields the same prediction hit rate.

- *Quality of ranking.* This metric quantifies the accuracy of the relative ranking of optimization choices and therefore is able to compensate for the shortcoming of the *best prediction hit rate* introduced above.

Assume we have N optimization choices. opt_1, \dots, opt_N , and have observed and estimated speedups for each. We define P_{ij} as

$$P((opt_i \text{ obs.} > opt_j \text{ obs.} | opt_i \text{ est.} > opt_j \text{ est.}) \text{ or } (opt_i \text{ obs.} < opt_j \text{ obs.} | opt_i \text{ est.} < opt_j \text{ est.}))$$

for all i, j between 1 and N , inclusive. We define *quality of ranking* for the cost model as the arithmetic mean of all these probabilities. For example, consider a case where there are four possible optimizations choices (opt_1, opt_2, opt_3 and opt_4) for an application, and where three experiments have been run for this application. Suppose the ranking of the estimated costs and observed costs for the first experiment is estimated as $opt_1, opt_3, opt_4, opt_2$ and observed to be $opt_4, opt_3, opt_1, opt_2$; for the second, estimated as $opt_2, opt_3, opt_4, opt_1$ and observed to be $opt_2, opt_1, opt_3, opt_4$; for the third, estimated as $opt_1, opt_3, opt_4, opt_2$ and observed to be $opt_1, opt_3, opt_4, opt_2$. Then, $P_{12} = 1, P_{13} = 1/3, P_{14} = 1/3, P_{23} = 1, P_{24} = 1$, and $P_{34} = 2/3$. The cost model’s *quality of ranking* for this example, based on these three ranking pairs, is the arithmetic mean of 1, 0.33, 0.33, 1, 1, 0.67, which is 0.72 (72%).

The cost model’s *quality of ranking* across all benchmarks is 77%. For *vpenta*, the cost model’s *quality of ranking* is 89%, which shows that the cost model accurately ranks all optimizations correctly except the first and second (**lr** and **lx** respectively, which are very close).

- *Worst vs. best.* This metric tells us what the performance would be if the worst optimization choice were made for each experiment, giving us a lower bound by which to evaluate the cost model.

For example, applying the worst choice of optimizations to /em syr2k results in a version that runs 10 times slower than one to which the best optimizations are applied. For the benchmarks studied, the average performance of the worst optimized versions were 32.9% of the best versions **b**. The cost model’s performance far exceeds this low-water mark.

- *Model vs. best.* This metric measures how close the cost model’s choice of optimizations is to that of **b**.

It is the primary metric for describing the accuracy of the cost model. In this metric, the cost model’s accuracy is penalized only to the extent that the model’s predictions deviate from the best possible. For the benchmarks under study, this metric is 94.8%, indicating that the cost model’s predictions are close to optimal for all benchmarks.

Figure 11 shows the accuracy of the cost model using these quantitative metrics. We find that quantifying overheads and access costs of the various restructuring optimizations allows us to make good decisions about which optimizations or mix of optimizations to apply and in what order. There is no single best order of application, and one optimization can permit, prevent, or change the effectiveness of another. The recommendations made by our cost model result in a mean overall speedup of 95% of optimal (for the combinations of candidate optimizations we consider). In contrast, the best performance from any single optimization choice is 77% of optimal in our experiments. These results indicate that our cost model is useful and effective in driving restructuring optimizations.

5 Related Work

Software solutions for tackling the memory bottleneck have been studied widely. They include software prefetching [27], control and data restructuring techniques [4, 22, 33], cache-conscious structure layout [9], and cache-oblivious algorithms [12]. We focus our comparison on program restructuring techniques for improving locality of arrays in loop nests, and on analytical models for program and memory performance.

Loop Transformations McKinley, Carr and Tseng [24] present a loop transformation framework for data locality optimization that guides loop permutation, fusion, distribution and reversal. Their framework is based on a simple, yet effective, model for estimating the cost of a executing a given loop nest in terms of the number of cache lines it accesses. They use the model to find the compound loop transformation that has the fewest accesses to main memory. The key idea in their memory cost model is that if a loop l accesses fewer cache lines than l' when both are considered as innermost loops, then l will promote more reuse than l' at any outer loop position. Thus, to obtain the loop permutation with the least cost, they simply rank the loops in descending order of their memory costs from the outermost to the innermost, subject to legality constraints. They also present heuristics for applying loop fusion and distribution using the same memory cost metric. The cost model that we propose is based on this cost model. However, we model the memory costs of the whole program and not just a single loop in a loop nest. The advantage of our approach is that we can compare loop transformations with other independent restructuring optimizations, such as array restructuring, and determine which is more profitable. Also, we can predict whether the optimized portions of the program will contribute significantly to the overall execution time or not using our cost model. However, if loop transformations alone are considered, the framework developed by McKinley *et al.* is adequate. The disadvantage of our approach is its increased complexity and computation time.

Wolf and Lam [33] define four different types of reuse—*self-spatial*, *self-temporal*, *group-spatial* and *group-temporal*—and present a mathematical formulation of each type of reuse using vector spaces. They introduce the concept of a *reuse vector space* to capture the potential of data locality optimizations for a given loop nest. They say that there is reuse if the same data item is used in multiple iterations of a loop nest. This is an inherent characteristic of the computation and is independent of the actual locality, which depends on the cache parameters. They also define the *localized iteration space* as the iterations that can exploit reuse. They present a data locality optimizing algorithm to choose a compound loop transformation involving permutation, skewing, reversal, and tiling that minimizes a locality cost metric. They represent loop dependences as vectors in the iteration space, and thus loop transformations can be represented as matrix transformations. Their algorithm’s execution time is exponential in the number of loops in the worst case; they prune their search space by considering only the loops that carry reuse. They use the number of memory accesses per iteration of the innermost loop as their locality cost metric. They calculate the locality cost metric by intersecting the reuse vector space with the localized vector space. Their metric is potentially more accurate than the one derived by McKinley *et al.* as it directly calculates reuse across outer loops. This cost metric can form the basis for extension in our cost model. Whether such a model would be more effective than the one derived by McKinley *et al.* remains to be seen.

While both of the above techniques have proved to be very useful in improving application performance through a combination of loop transformations, they do not consider data transformations that can be used in a complementary

fashion to get still better performance. As described in earlier sections, loop transformations alone are not sufficient for achieving good data locality in many cases.

Data Transformations Leung and Zahorjan [22] introduced *array restructuring*, a technique to improve the spatial locality of arrays in loop nests. This technique is very effective in improving the performance of certain applications. In general, though, array restructuring needs to be carefully applied because of its high overheads. In their work, Leung and Zahorjan perform no profitability analysis to determine when array restructuring should be applied. Instead, they use a simple heuristic that considers array restructuring to be profitable if the *access matrix* of the array has more non-zero columns at the right than its rank. This heuristic is simplistic because it 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. In *btrix*, their heuristic-based approach recommended that copying-based array restructuring not be done at all. In contrast, our cost model estimated that copying-based array restructuring would be beneficial. The observed performance validated our approach as we obtained a 49% speedup despite the overhead of copying.

Callahan *et al.* [3] introduced *scalar replacement*. When an array element is invariant within the innermost loop or loops, it can be loaded into a scalar before the inner loop. If the scalar is modified, it can be stored after the inner loop. This optimization decreases the number of memory accesses since the scalar can typically be stored in a register during the complete run of the inner loop, and eliminates unnecessary subscript calculations. We assume that the optimization is done wherever possible throughout this paper. This assumption also simplifies the calculation of the cost model.

Combined Transformations Cierniak and Li [10] propose a linear algebraic framework for locality optimization by unifying data and control transformations. Their framework is designed to reduce false sharing in distributed shared memory machines. The data transformations they use is restricted to index permutations of arrays. They also do not consider multiple loop nests.

Kandemir *et al.* [15] extend Li's work by considering a wider set of possible loop transformations. They handle multiple loop nests by fixing the memory layout of arrays as determined in previous loops. They make an *ad hoc* decision to always optimize a loop nest for temporal locality, irrespective of whether the optimization causes poor spatial locality for the remaining arrays. The remaining arrays are optimized for spatial locality by changing the data layout, if possible. An implicit assumption in this algorithm is that the array sizes and loop trip counts are sufficiently large that such a transformation is always profitable.

Both Li and Kandemir consider only static data transformations. Once an array is optimized in a particular way in a loop nest, the array has a fixed layout throughout the program. The rest of the loop nests do not have a say in the locality optimization of the array in question. Our integrated optimization strategy considers dynamic data restructuring in which an array access is replaced by a locally-optimal alias or copy. Neither Li nor Kandemir do any cost/benefit analysis to decide when to apply their optimizations, and simply assume that they are always profitable. We employ profitability tests for all our optimizations.

Cost Models Saavedra *et al.* [31] develop a uniprocessor, machine-independent model (the *Abstract Machine Model*) of program execution to characterize machine and application performance, and the effectiveness of compiler optimization. It can predict with high accuracy the running time of a given benchmark on a given machine. Their model, however, omits any consideration of cache and TLB misses. Since they encountered low cache and TLB misses in the SPEC92 and Perfect benchmarks, the lack of a memory cost model did not pose a problem; however, modeling cache and TLB misses caused their model to underestimate the running time of applications with high miss rates. In later work, Saavedra and Smith [30] deal with the issue of locality and incorporate memory costs into their model. They calculate cache and TLB parameters, miss penalty, associativity, and cache line size using a series of microbenchmarks that exercised every level of the memory hierarchy. However, they do not model or estimate the cache hit rates of applications. Rather, they used published cache and TLB miss ratios for the SPEC benchmarks to compute the additional execution time caused due to poor TLB and cache performance. The prediction errors in most of the benchmarks decreased with the new parameters and model. We use a similar methodology to calculate the various architectural cost parameters in our model.

Ghosh *et al.* [13] introduce Cache Miss Equations (CMEs), a precise analytical representation of cache misses in a loop nest. CMEs, however, have some deficiencies. They cannot determine if the cache is warm or cold before entering a loop nest. They also cannot handle imperfect or multiple loop nests. However, they represent a promising

step towards being able to accurately model the cache behavior of regular array accesses. Their work is complementary to ours and can be used to further enhance the accuracy of our model. Specifically, we would like to be able to estimate f , the degree of cache reuse across iterations of the innermost loop, accurately. Such an estimate would allow us to improve the accuracy of our cost model’s predictions, and also improve our results on small dataset sizes.

6 Future Work

The results of this initial study are promising, but are by no means definitive. The cost model currently covers several basic, regular cases with sufficiently large working set sizes. The most significant addition would be a mechanism to estimate cache line reuse within a loop nest. The current model will overestimate cache misses in codes that use a neighborhood of matrix points during each innermost iteration, such as stencil methods.

Cache Miss Equations (CMEs) [13] represent the most exact means (known to date) of calculating this cache line reuse. They create a precise description of which cache lines are reused based on an exact model of all references within the particular loop nest. However, CMEs quickly become complicated, especially for complex loop constructs. A less precise model probably suffices for the cost model discussed here.

One such possible (faster) approach would use the stencil footprint of each iteration as the basis for the cost model. If the stencil is assumed to be applied at each element of the matrix, the stencil width characterizes the potential reuse: for instance, a stencil that is three elements wide has the potential to reuse data from the two previous iterations. This observation, combined with knowledge of the relative sizes of the matrix and the cache, should provide the required information to estimate overall cache line reuse within the entire loop nest. Such an approach should also be able to take limited cache sizes and limited working set sizes into account. Tiling could be modeled in a similar way.

Our cost model could incorporate information about inter-nest locality optimizations. For instance, Kandemir et al. [17] demonstrate that optimizing different nests to take advantage of the cache contents left by earlier nests may be beneficial. For two of the four benchmarks that they study, taking advantage of inter-nest locality results in better performance than loop and data restructuring; in the other two, it results in worse performance. Integrating an analysis of this tradeoff into our cost model would be a valuable generalization.

We did not evaluate the combination of using both remapping-based restructuring and copying-based restructuring in the same program. A compiler could combine remapping with copying for one loop: for example, it could use remapping to speed up the copying loop itself. It would be interesting to evaluate whether such a combination would be valuable. Given that the number of optimization combinations grows exponentially in the number of optimizations, we would probably need to implement the model in a compiler to evaluate the space of optimizations more fully.

Finally, our cost model could be made more accurate by modeling more architectural features, such as the TLB, a two-level cache hierarchy, and memory controller-based prefetching. Optimizing for both the TLB and the cache simultaneously is an open problem, at least with respect to blocking optimizations. Future work in this direction also involves extending the cost model to include more types of remappings.

7 Conclusions

Program restructuring optimizations can be effective, but they do not always deliver equal benefits for all applications. Previous work has integrated different restructuring optimizations; to our knowledge, our work is the first to analytically model the costs and benefits of combining restructuring strategies. This paper demonstrates that the memory costs of applications can be modeled to allow the qualitative comparison of multiple locality optimizations — alone or in combination. The accuracy of our cost model is encouraging, given its simplicity. This model should provide the basis for a wider integration of locality strategies, including tiling, blocking, and other loop transformations.

This paper also describes how hardware support for address remapping enables a new data restructuring optimization that has different costs and benefits from copying. Such hardware support enables more diverse data restructuring techniques. We contend that combining the benefits of software-based and hardware-based restructuring optimizations is a valuable research direction, and we provide a framework for reasoning about the combined effects of such optimizations.

Our analysis and simulations demonstrate the following points about the optimizations we study:

- Copying-based array restructuring, although useful in improving locality for multi-dimensional array accesses, is not widely applicable, because the overhead of copying can overwhelm the benefit derived from improved locality. Applicability depends largely on input parameters and reuse of data resident in cache.

- Loop transformations can improve memory locality, but they are not sufficient for many applications. They can be used in conjunction with array restructuring to achieve increased locality. These complementary optimizations can yield improved locality, but not necessarily better performance. Thus, integrated restructuring may be useful, but must be done carefully.
- Restructuring does not always deliver better performance. Our model helps to predict when which combination of restructuring optimizations is likely to be useful.

On average, our initial cost model chooses optimizations that deliver good speedups – within 5% of the geometric mean of the best possible choice among the optimizations we consider. This efficient model helps compilers or programmers to generate faster code for array-based applications, and it provides the foundation for a more comprehensive model that will cover more types of codes, addressing multilevel memory hierarchies and the data reuse within them.

Acknowledgments

We gratefully thank Martin Schulz for providing us with a great deal of feedback and editorial assistance with this paper, and with suggesting possibilities for future improvements of our model.

References

- [1] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Computing Surveys* 26, 4 (Dec. 1994), 345–420.
- [2] BALA, V., E.DUESTERWALD, AND BANERJIA, S. Dynamo: A transparent runtime optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2000), pp. 1–12.
- [3] CALLAHAN, D., CARR, S., AND KENNEDY, K. Improving register allocation for subscripted variables. *ACM SIGPLAN Notices* 25, 6 (June 1990), 53–65.
- [4] CARR, S., MCKINLEY, K., AND TSENG, C.-W. Compiler optimizations for improving data locality. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 252–262.
- [5] CARTER, J., HSIEH, W., STOLLER, L., SWANSON, M., ZHANG, L., BRUNVAND, E., DAVIS, A., KUO, C.-C., KURAMKOTE, R., PARKER, M., SCHAELOCKE, L., AND TATEYAMA, T. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture* (Jan. 1999), pp. 70–79.
- [6] CHANDRAMOULI, B. A cost framework for evaluating integrated restructuring optimizations. Master’s thesis, University of Utah School of Computing, May 2002.
- [7] CHANDRAMOULI, B., CARTER, J., HSIEH, W., AND MCKEE, S. A cost framework for evaluating integrated restructuring optimizations. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001), pp. 98–107.
- [8] CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., MUNDHRA, S., AND THOTTETHODI, M. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 Conference on Supercomputing* (June 20–25 1999), pp. 444–453.
- [9] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1999).
- [10] CIERNIAK, M., AND LI, W. Unifying data and control transformations for distributed shared memory machines. Tech. Rep. TR-542, University of Rochester, Nov. 1994.
- [11] FRENS, J. D., AND WISE, D. S. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP-97)* (New York, June 18–21 1997), vol. 32, 7 of *ACM SIGPLAN Notices*, ACM Press, pp. 206–216.
- [12] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York*, (1999), IEEE, Ed., IEEE Computer Society Press, pp. 285–297.
- [13] GHOSH, S., MARTONOSI, M., AND MALIK, S. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct. 3–7, 1998), pp. 228–239.

- [14] HUANG, X., WANG, Z., AND MCKINLEY, K. Compiling for an impulse memory controller. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001), pp. 141–150.
- [15] KANDEMIR, M., CHOUDHARY, A., RAMANUJAM, J., AND BANERJEE, P. Improving locality using loop and data transformations in an integrated approach. In *Proceedings of IEEE/ACM 31st International Symposium on Microarchitecture* (Dec. 1998).
- [16] KANDEMIR, M., CHOUDHARY, A., SHENOY, N., BANERJEE, P., AND RAMANUJAM, J. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proceedings of the International Conference on Supercomputing (ICS-98)* (New York, July 13–17 1998), ACM press, pp. 69–76.
- [17] KANDEMIR, M., KADAYIF, I., CHOUDHARY, A., AND ZAMBRENO, J. Optimizing inter-nest data locality. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Oct. 2002).
- [18] KENNEDY, K., AND MCKINLEY, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing* (Aug. 12–14, 1993), Lecture Notes in Computer Science, Springer-Verlag, pp. 301–320.
- [19] KREMER, U. NP-Completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers* (Delft, The Netherlands, 1993). (also available as CRPC-TR93330-S).
- [20] LAM, M., ROTHBERG, E., AND WOLF, M. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991), pp. 63–74.
- [21] LARUS, J., AND SCHNARR, E. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* (June 1995).
- [22] LEUNG, S. *Array Restructuring for Cache Locality*. PhD thesis, University of Washington, Aug. 1996.
- [23] LEUNG, S., AND ZAHORJAN, J. Optimizing data locality by array restructuring. Tech. Rep. UW-CSE-95-09-01, University of Washington Dept. of Computer Science and Engineering, Sept. 1995.
- [24] MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18, 4 (July 1996), 424–453.
- [25] MCMAHON, F. The Livermore Fortran kernels: A computer test of the numerical performance range. Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.
- [26] MIPS TECHNOLOGIES INC. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.
- [27] MOWRY, T., LAM, M. S., , AND GUPTA, A. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct. 1992), pp. 62–73.
- [28] M.TALLURI, AND HILL, M. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 171–182.
- [29] PAI, V., RANGANATHAN, P., AND ADVE, S. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter* (Fall 1997).
- [30] SAAVEDRA, R. H., AND SMITH, A. J. Measuring cache and TLB performance and their effect on benchmark run times. *IEEE Trans. on Computers C-44*, 10 (Oct. 1995), 1223–1235.
- [31] SAAVEDRA-BARRERA, R. H. Machine characterization and benchmark performance prediction. Technical Report CSD-88-437, University of California, Berkeley, June 1988.
- [32] SWANSON, M., STOLLER, L., AND CARTER, J. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (June 1998), pp. 204–213.
- [33] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. *ACM SIGPLAN Notices* 26, 6 (June 1991), 30–44.
- [34] ZHANG, L. URSIM reference manual. Tech. Rep. UUCS-00-015, University of Utah, Aug. 2000.
- [35] ZHANG, L. *Efficient Remapping Mechanisms For An Adaptable Memory System*. PhD thesis, University of Utah, Jan. 2002.
- [36] ZHANG, L., CARTER, J., HSIEH, W., AND MCKEE, S. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques* (Oct. 1999), pp. 98–107.

```

double A[M][L],B[N][M],C[N][L];
for (k=0;k<N;k++)
  for (i=0;i<L;i++)
    for (j=0;j<M;j++)
      C[k][i] = C[k][i] + A[j][i]*B[k][j];

```

Figure 12. *Matmult* kernel

Appendix

This appendix summarizes the behavior of each benchmark, examining how a code’s particular characteristics interact with our cost model’s ability to choose appropriate optimizations.

Matmult Matrix multiplication is among the most studied codes in scientific computing. There are many advanced techniques to improve its performance: loop tiling [20], non-linear array layouts [8], recursive matrix multiplication [11], and using superpages to map the entire array with a single TLB entry [28]. We select it because it is used in so many similar optimization studies, and thus it allows us to relate our results to those of others.

Matmult multiplies a $N \times M$ matrix, $B[N][M]$, by an $M \times L$ matrix, $A[M][L]$, to generate product matrix $C[N][L]$ (see Figure 12). For large arrays, this code suffers many TLB and cache misses, but remapping-based and copying-based array restructuring generate unit-stride accesses for all of these arrays (explained in the example from Section 2). Of our benchmarks, 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 *matmult* performance, loop restructuring usually achieves the same effect with lower setup costs. Our cost model therefore recommends loop permutation over other options. Nonetheless, conflict misses in the loop-restructured code may cause remapping to perform better for some input data sizes. Since conflict misses are not represented in our cost model, it incorrectly recommends loop permutation for all cases, and achieves a lower speedup than the best possible. Likewise, the model does not consider possible cache reuse between iterations of the outer loops, and thus it overestimates the original code’s memory costs for small data sets. The performance penalty induced by this limitation in the cost model is low, though, because main memory performance is not a significant performance bottleneck for codes exhibiting high locality. When array sizes greatly exceed cache size, the cost model correctly prefers **lx** to the baseline. Figure 10 shows that had the best choice had been made for each data set, mean speedup would have been 46%. Our cost model achieves a mean speedup of 34% over the original program (91.8% of optimal for our candidate set). Table 5 shows the characteristics of each choice in our experiments.

Syr2k The *syr2k* subroutine comes from the BLAS library [11] and computes the banded matrix calculation $C = \alpha A^T B + \alpha B^T A + C$. The arrays are $N \times N$ matrices, and the width of the band is b . Figure 13 shows the banded matrix calculation’s unoptimized core kernel. The loop references four array elements from different rows and columns during each iteration of the innermost loop, yielding poor cache and TLB hit rates. The number of accesses to band elements is $O(Nb^2)$. Data dependences rule out loop permutation, so our model combines loop distribution and remapping-based array restructuring (**lr**) to improve this kernel. Copying-based array restructuring (**ac**) can be used effectively for large bands, where the fixed setup cost of copying ($O(N^2)$) is amortized over the subsequent data reuse, but remapping performs better for small bands. The cost model correctly identifies this behavior in most cases. By correctly choosing when to apply copying (**ac**) and when to choose combined loop and remapping-based restructuring

<i>matmult</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	54	5	11	3 2-D	90.9	6.8	2.25	98.9	1.00
ar	72	5	11	1 1-D 3 2-D	98.8	0.2	1.05	99.9	1.24
ac	54	6	13	4 2-D	95.2	3.8	0.98	99.9	1.26
lx	53	5	11	3 2-D	95.3	3.7	0.96	99.9	1.34

Table 5. Performance characteristics of *matmult* optimizations (mean *best* speedup = 1.46)

```

int A[N][N],B[N][N],C[N][N];
for (i=1;i<N;i++)
  for (j=i;j<min(i+2*b-2,N);j++)
    for (k=max(i-b+1,j-b+1,1);k<min(i+b-1,j+b-1,N);k++)
      C[j-i+1][i] += A[i-k+b][k]*B[j-k+b][k] + A[j-k+b][k] * B[i-k+b][k];

```

Figure 13. *Syr2k* kernel

(**lr**), the cost model achieves better performance than either in isolation. Table 6 shows the characteristics of each choice in our experiments.

Vpenta The *vpenta* subroutine performs large-stride accesses to eight two-dimensional arrays in seven loop nests. Loop permutation can optimize the two most expensive loop nests. The remaining nests have strided accesses that loop transformations cannot optimize. For the remaining array references with strided accesses, we consider remapping-based restructuring. This combination of loop transformation (**lx**) and remapping-based array restructuring (**lr**) results in the best overall performance. Additional array restructuring has little impact, since the five loop nests where remapping is legal only account for a small portion of overall execution time. The cost model recommends **lx**, which performs 4% worse than a fixed choice of **lr** and 9% worse than **b**. For input sizes that are a power of two, remapping eliminates a significant number of conflict misses, but since our cost model does not account for cache conflict effects, it underestimates the potential benefits that remapping achieves. A more sophisticated cost model leveraging cache miss equations [13] or similar mechanisms might recognize these cases, but experimenting with more sophisticated models is beyond the scope of this work.

Btrix *Btrix* solves a block tridiagonal matrix equation along one dimension of a four-dimensional array. The innermost loop is vectorizable and involves strided accesses across four four-dimensional arrays. Each of these arrays shares the same innermost loop index in the first dimension, giving rise to huge strides between accesses. Figure 14 shows a portion of the code from the original program. We optimize this program with loop fusion and permutation. Moving the innermost loop l to the outermost loop position and fusing all subnests in the first loop nest yields an order of j, l, m, n for subnest 1.1 and j, l, n for subloop nest 1.4. Interchanging the outermost loops, j and l , brings arrays a , b and c into array order in the first loop nest. Finally, loop nest 2 can be permuted in two different ways, representing loop transformation strategies **lx** and **lx'**.

Loop transformations alone cannot bring all four arrays a , b , c , and s into array order, because the access pattern of array s conflicts with the others. We further optimize this nest using copying- and remapping-based array restructuring to generate four more candidates (**lr**, **lr'**, **lc**, **lc'**). None of these four choices performs better than either **lx** or **lx'** alone because of the high overhead in restructuring the array s . Our cost model correctly calculates these costs and benefits. It predicts that **lx'** is the best optimization; in eight of nine experiments, **lx'** performs best. The single misprediction occurs when the array dimensions are a power of two, which induces many conflict misses in **lx'**. For this case, **lc'** performs best, because all arrays have unit stride and there are fewer conflict misses. Again, since we do not model conflict misses, the cost model erroneously chooses **lx'** over **lc'**, which leads to a performance penalty of 48%. The overall speedup for the cost model's recommendations is 1.72, while the best overall speedup is 1.80 (i.e., the cost model delivers 95.6% of **b**).

Our cost model correctly predicts that copying-based array restructuring (**ac**) is beneficial. In contrast, Leung's heuristics recommend against copying-based array restructuring [22]. We do not use pure remapping-based array

<i>Syr2k</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	54	3	7	3 2-D	79.1	16.8	4.1	99.1	1.00
ar	105	3	7	4 1-D 3 2-D	89.4	10.0	0.6	99.9	0.75
ac	63	4	9	5 2-D	91.7	7.3	1.0	99.5	1.71
lr	105	3	7	6 2-D	96.0	3.6	0.4	99.9	1.56

Table 6. Performance characteristics of *syr2k* optimizations (mean *best* speedup = 2.00)

<i>Vpenta</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	382	13	19	7 2-D 2 3-D	92.0	3.3	4.8	99.2	1.00
ac	427	15	23	14 2-D 4 3-D	88.0	5.0	7.0	98.6	0.60
ar	698	13	19	8 1-D 7 2-D 2 3-D	89.7	8.9	1.4	99.8	1.11
lx	409	13	19	7 2-D 2 3-D	92.5	5.5	2.0	99.9	1.47
lr	631	13	19	8 1-D 7 2-D 2 3-D	92.7	6.1	1.2	99.9	1.61

Table 7. Performance characteristics of *vpenta* optimizations (mean *best* speedup = 1.65)

```

int s[6][N][N][N], a[N][N][6][6], b[N][N][6][6], c[N][N][6][6];
for (j=1; j<N; j++) /* loop nest 1 */
{
    ...
    for (m = 1; m<=5; m++) /* subloop nest 1.1 */
        for (n = 1; n<=5; n++)
            for (l=1; l<N; l++)
                b[l][j][n][m] = b[l][j][n][m]
                - a[l][j][1][m] * b[l][j-1][n][1]
                - a[l][j][2][m] * b[l][j-1][n][2]
                - a[l][j][3][m] * b[l][j-1][n][3]
                - a[l][j][4][m] * b[l][j-1][n][4]
                - a[l][j][5][m] * b[l][j-1][n][5];
    ...
    for (m=1; m<=5; m++) /* subloop nest 1.4 */
        for (l=1; l<N; l++)
            s[m][l][k][j] = s[m][l][k][j]
            - a[l][j][1][m] * s[1][l][k][j - 1]
            - a[l][j][2][m] * s[2][l][k][j - 1]
            - a[l][j][3][m] * s[3][l][k][j - 1]
            - a[l][j][4][m] * s[4][l][k][j - 1]
            - a[l][j][5][m] * s[5][l][k][j - 1];
    ...
}
for (j=N-2; j>=1; j--) /* loop nest 2 */
{
    for (m = 1; m<=5; m++)
        for (l=1; l<N; l++)
            s[m][l][k][j] = s[m][l][k][j]
            - b[l][j][1][m] * s[1][l][k][j+1]
            - b[l][j][2][m] * s[2][l][k][j+1]
            - b[l][j][3][m] * s[3][l][k][j+1]
            - b[l][j][4][m] * s[4][l][k][j+1]
            - b[l][j][5][m] * s[5][l][k][j+1]
}
...

```

Figure 14. *Btrix* kernel

<i>Btrix</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	184	10	24	25 1-D 4 4-D	86.7	11.2	2.0	96.6	1.00
ac	218	14	40	25 1-D 7 4-D	95.7	3.1	1.2	99.9	1.49
lx	173	9	23	25 1-D 4 4-D	95.2	3.6	1.2	98.4	1.36
lx'	173	9	19	25 1-D 4 4-D	96.8	2.4	0.8	99.9	1.72
lc	188	10	27	25 1-D 5 4-D	96.5	3.2	1.3	99.9	1.43
lc'	188	10	27	25 1-D 5 4-D	96.5	2.8	0.7	99.9	1.67
lr +	246	9	24	26 1-D 4 4-D	96.2	2.7	1.1	98.6	1.24
lr'	246	9	23	26 1-D 4 4-D	97.3	1.9	0.8	99.9	1.47

Table 8. Performance characteristics of *btrix* optimizations (mean *best* speedup = 1.80)

restructuring (**ar**) for this benchmark, because doing so requires more than eight remappings, the maximum number that the current Impulse implementation supports. Loop transformations, however, enable remapping-based array restructuring by reducing the number of arrays that needed to be restructured.

Cfft2d *Cfft2d* implements a two-dimensional FFT. It consists of two subroutines that perform FFTs along the first and second dimensions of the array, respectively, thereby taking advantage of the parallel structure of the algorithm. Previous work on loop transformations either do not optimize this loop nest or report no performance improvement [24]. Dependence constraints and imperfect loop nests make loop transformations infeasible, but do not prevent copying- or remapping-based array restructuring. Both the latter optimizations prove effective for this benchmark. Like *syr2k*, the relative performance of copying-based restructuring versus remapping-based restructuring is input-size dependent. The array sizes are $O(N^2)$, but each element is touched $O(N^2 \log N)$ times. Thus, copying outperforms remapping for large data sets where the high, one-time setup cost ($O(N^2)$) is amortized by the lower cost per access over many accesses. Conversely, remapping is preferable for small data sets. Our cost model correctly predicts this tradeoff for most cases, and obtains a speedup very close to the best possible.

Ex1 Figure 15 shows the *ex1* loop nest. Array *U* has good spatial locality. Array *V*, however, not only accesses a new cache line but also a new page in every iteration of the innermost *k* loop for large values of *N*. In this kernel, only three of the six possible loop permutations are legal. Loop permutation by itself hurts performance, but it enables beneficial data transformations. Figure 10 shows that, on average, **lc** performs worse than the original code. However, **lc** outperforms the base version for the four large data sets.

The main cause for poor performance for large data set sizes is high TLB miss rates. For example, when $N = 500$, the TLB hit rate is 87.2% and TLB misses account for 42% of overall execution time. In contrast, for $N = 200$, the TLB hit rate is 99.9%, and TLB misses represent only 2% of overall execution time. We consider five optimization choices for this kernel – **ac**, **ar**, **lx**, **lr**, **lc** – and perform eight experiments with different values of *N* (ranging from $N = 200$ to $N = 500$). Copying-based array restructuring delivers the best speedup in each case. Remapping suffers from high latencies while gathering cache lines, and does not perform well. The cost model successfully predicts the

<i>Cfft2d</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	212	11	23	1 2-D, 3 1-D	88.0	5.9	6.1	98.7	1.00
ac	239	13	27	2 2-D, 3 1-D	91.7	6.3	1.9	98.9	2.77
ar	238	13	23	2 2-D, 3 1-D	94.1	4.4	1.5	99.9	2.90

Table 9. Performance characteristics of *cfft2d* optimizations (mean *best* speedup = 2.91)

```

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

```

Figure 15. *Ex1* kernel

best optimization in all cases.

Ir_kernel The optimization choices for *ir_kernel* (see Figure 2) are **ac**, **ar**, **lx**, **lr** and **lc**. As expected, a combination of loop transformation and data restructuring results in the best performance for *ir_kernel*. **lc** outperforms **ac**, and **lr** outperforms **ar** in all cases. Between **lr** and **lc**, the former performs better, but the cost model predicts otherwise. The main reason for this misprediction is the high incidence of conflict misses in **lc** compared to **lr**. The penalty for this incorrect prediction is high and causes the cost model’s recommendations to reach only 85% of **b**.

Kernel6 *Kernel6* is a general linear recurrence equation solver from the Lawrence Livermore National Laboratories benchmark suite of loop kernels (also known as the Livermore Fortran Kernels [25]). The loop’s recurrence relation makes permutation illegal (see Figure 16). We use remapping-based and copying-based array restructuring instead. Choosing **ar** over **ac** depends on the amount of reuse in the loop nest. Over 22 experiments with various loop and data set sizes, our cost model accurately chooses the best optimization in 21 cases. Even in the single misprediction case the performance is very close to the best choice. When we use the cost model to select the optimizations to perform, we achieve an overall speedup of 1.99, whereas the best overall mean speedup is 2.00. In contrast, applying remapping (**ar**) or copying (**ac**) exclusively results in an average speedup of only 1.92 or 1.24, respectively.

<i>Ex1</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	51	3	7	2 2-D	90.5	8.0	1.5	93.5	1.00
ac	58	4	9	3 2-D	91.3	7.1	1.6	99.7	1.53
ar	75	3	7	2 2-D, 1 1-D	98.0	0.5	1.5	99.9	0.74
lx	51	3	7	2 2-D	82.8	13.8	3.4	94.6	0.66
lc	62	5	11	3 2-D	78.1	16.8	5.1	99.9	0.76
lr	71	3	7	2 2-D, 1 1-D	85.4	11.0	3.6	99.9	0.43

Table 10. Performance characteristics of *ex1* optimizations (mean *best* speedup = 1.53)

<i>Ir_kernel</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	111	2	4	2 1-D, 2 2-D	96.4	3.1	0.5	99.9	1.00
ac	120	4	8	2 1-D, 4 2-D	97.6	2.0	0.4	99.9	1.07
ar	154	2	4	3 1-D, 2 2-D	98.9	0.5	0.6	99.9	1.26
lx	111	2	4	2 1-D, 2 2-D	94.7	4.8	0.5	99.9	0.96
lc	117	3	6	2 1-D, 3 2-D	97.6	2.0	0.4	99.9	1.07
lr	135	2	4	3 1-D, 2 2-D	99.0	0.5	0.6	99.9	1.26

Table 11. Performance characteristics of *ir_kernel* optimizations (mean *best* speedup = 1.26)

<i>Kernel6</i>	# of Lines	# of Nests	# of Loops	# of Arrays	L1 Hit	L2 Hit	Miss Rate	TLB Hit	Mean Speedup
base	43	2	5	1 2-D, 1 1-D	87.4	8.8	3.8	95.6	1.00
ac	49	3	7	2 2-D, 1 1-D	90.9	6.2	2.9	98.3	1.24
ar	60	2	5	2 2-D, 1 1-D	95.1	3.7	1.2	99.9	1.92

Table 12. Performance characteristics of *kernel6* optimizations (mean *best* speedup = 2.00)

```

double w[N],B[N][N];
for (l=1; l<=loop; l++)
  for (i=1; i<N; i++)
  {
    for (k=0; k<i; k++)
      w[i] += B[k][i] * w[(i-k)-1];
    sum=sum+w[i];
  }

```

Figure 16. *Kernel6* code