A COST FRAMEWORK FOR EVALUATING
INTEGRATED RESTRUCTURING
OPTIMIZATIONS

by

Bharat Chandramouli

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

School of Computing
The University of Utah
May 2002
THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Bharat Chandramouli

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

__________________________
Chair: John B. Carter

__________________________
Wilson C. Hsieh

__________________________
Sally A. McKee
THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of The University of Utah:

I have read the thesis of Bharat Chandramouli in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to the Graduate School.

Date

John B. Carter
Chair, Supervisory Committee

Approved for the Major Department

Thomas Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School
ABSTRACT

Loop transformations and array restructuring usually improve the performance of applications by increasing their memory locality. They can also be harmful. For instance, loop and array restructuring can either complement or compete with one another. Previous research has proposed integrating loop and array restructuring, but it was not driven by an analytic framework for determining how best to combine the optimizations for a given program. Since the choice of which optimizations to apply, alone or in combination, is highly application- and input-dependent, a cost framework is needed if integrated restructuring is to be automated by an optimizing compiler. To this end, we develop a cost model that considers standard loop optimizations along with two potential forms of array restructuring: conventional copying-based restructuring and remapping-based restructuring that exploits a smart memory controller. We simulate eight applications on a variety of input sizes and with a variety of hand-applied restructuring optimizations. We find that employing a fixed strategy does not always deliver the best performance. Our cost model accurately predicts the best combination of restructuring optimizations among those we examine and yields performance within a geometric mean of 5% of the best combination across all benchmarks and input sizes.
To Amma and Appa
CONTENTS

ABSTRACT ........................................................................................................ iv

LIST OF FIGURES ............................................................................................ viii

LIST OF TABLES ............................................................................................... x

ACKNOWLEDGMENTS ......................................................................................... xi

CHAPTERS

1. INTRODUCTION ............................................................................................ 1
  1.1 Technology Trends .................................................................................. 1
  1.2 Problem Statement ............................................................................... 2
  1.3 Contributions ....................................................................................... 4
  1.4 Road Map .............................................................................................. 5

2. RESTRUCTURING OPTIMIZATIONS ............................................................... 6
  2.1 Loop Transformations ........................................................................... 6
  2.2 Copying-based Array Restructuring ..................................................... 8
  2.3 The Impulse Memory System .............................................................. 11
  2.4 Remapping-based Array Restructuring ............................................... 15
    2.4.1 Implementation Details ............................................................... 15

3. ANALYTIC FRAMEWORK .......................................................................... 18
  3.1 Modeling the Restructuring Strategies .............................................. 18
    3.1.1 Basic Model .................................................................................. 18
    3.1.2 Modeling Loop Transformation .................................................. 21
    3.1.3 Modeling Copying-based Array Restructuring ............................. 21
    3.1.4 Modeling Remapping-based Restructuring ................................ 22
    3.1.5 Estimating the Cost of Gathering Cache Lines ......................... 24
  3.2 Integrating the Restructuring Optimizations ....................................... 27

4. EVALUATION ................................................................................................. 29
  4.1 Benchmarks ........................................................................................... 29
  4.2 Results .................................................................................................... 31
    4.2.1 MATMUL ..................................................................................... 32
    4.2.2 SYR2K ....................................................................................... 34
    4.2.3 VPENTA ..................................................................................... 35
    4.2.4 BTRIX ....................................................................................... 36
4.2.5  CFFT2D ....................................................... 39
4.2.6  EX1 .......................................................... 40
4.2.7  IR KERNEI .................................................... 41
4.2.8  KERNEL6 ...................................................... 42
4.3  Summary ....................................................... 43
4.4  Caveats ......................................................... 47

5. RELATED WORK .................................................. 48
   5.1  Loop Transformations ........................................ 48
   5.2  Data Transformations ....................................... 50
   5.3  Combined Transformations .................................. 51
   5.4  Cost Models ................................................ 51

6. CONCLUSIONS AND FUTURE WORK ............................... 53
   6.1  Future Directions ........................................... 54

APPENDIX: EXAMPLE COST MODEL CALCULATION .................. 57

REFERENCES ........................................................ 64
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Matrix multiply kernel</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Direction of array accesses in the innermost loop with the original “ijk” loop order.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Direction of array accesses in the innermost loop with the optimized “ikj” loop order.</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Array restructuring example. We will refer to this code as <em>ir_kernel</em></td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Visualizing array restructuring of <em>ir_kernel</em></td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td><em>ir_kernel</em> optimized using copying-based array restructuring.</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Address mapping in an Impulse system.</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>Impulse architecture.</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td><em>ir_kernel</em> optimized using remapping-based array restructuring.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Example loop used to illustrate the cost model.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Microbenchmark used to compute $G_r$, the cost of gathering a remapped cache line (using the base stride mechanism) for various strides</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Graph showing the relationship between $G_r$ and the stride.</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Microbenchmark used to compute $G_c$, the cost of gathering a normal cache line.</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Graph showing the relationship between $G_c$ and the stride (in bytes).</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td><em>ir_kernel</em>, optimized by a combination of loop permutation and copying-based array restructuring.</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>MATMUL kernel</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>SYR2K kernel</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>BTRIX kernel</td>
<td>37</td>
</tr>
</tbody>
</table>
4.4 EXI kernel ................................................. 41
4.5 Kernel6 code ............................................. 42
LIST OF TABLES

2.1 Cost Model of McKinley et al. Applied to the Matrix Multiply Example . . 8
4.1 Benchmark Suite and Candidates for the Optimization Choices. . . . . . 30
4.2 Normalized Mean Speedups for Optimizations in each Benchmark . . . . 31
4.3 Performance Characteristics of the Optimizations for MATMULT . . . . . 33
4.4 Performance Characteristics of the Optimizations for SYR2K . . . . . . 34
4.5 Performance Characteristics of the Optimizations for VPENTA . . . . . . 36
4.6 Performance Characteristics of the Optimizations for BTRIX . . . . . . 38
4.7 Performance Characteristics of the Optimizations for CFFT2D . . . . . . 39
4.8 Performance Characteristics of the Optimizations for EX1 . . . . . . . . 41
4.9 Performance Characteristics of the Optimizations for IR KERNEL . . . . . 42
4.10 Performance Characteristics of the Optimizations for KERNEL6. . . . . . 43
4.11 Quantitative Metrics to Evaluate Cost Model . . . . . . . . . . . . . . . 46
A.1 Memory Costs for Individual Loop Nests . . . . . . . . . . . . . . . . . 62
A.2 Total Memory Costs for the Various Optimizations of BTRIX . . . . . . 63
ACKNOWLEDGMENTS

I would like to thank my thesis advisor, John Carter, for his guidance and encouragement. I learnt what "doing research" means from him. To be able to work for John was the reason I changed my focus from systems to computer architecture. John's classes have also been great fun, and I have always enjoyed his wacky sense of humor.

I am grateful to Wilson Hsieh and Sally McKee, members of my committee, for teaching me how to present research results effectively. Wilson's insights and suggestions especially helped give direction to this thesis. I also thank him for his moral support during my job search when things were looking bleak for a while. Sally has always been a pleasure to work with. I thank her for her guidance especially with the writing. I also thank her for offering to sponsor my trip to Barcelona and later for presenting my paper when I could not go.

I thank all my friends who made my life in Utah very enjoyable: Vidya, Lavanya, Ryan, Ashutosh, Mohan, Abhijeet, Yeshwant, Karra, and Binu. I would also like to thank Cinemark Theatres for their "dollar theater" in Sugarhouse – we've seen more movies there than anywhere else. Also, I thank the computer science department for providing us a wonderful "grad break room" – the venue for endless sessions of foosball games and weekend rental movies on a big screen.

I am deeply grateful to my parents for inculcating the love of reading in me. One of the reasons I chose to study computer science was the excitement I felt when I read the biographies of mathematicians and computer scientists. I also thank my mother for teaching me mathematics, and my father for teaching me not to take life too seriously.
CHAPTER 1

INTRODUCTION

The memory system has become a bottleneck that limits the performance of many applications. Applications need to use the memory hierarchy efficiently for good overall performance. Thus, compiler optimizations that improve locality are becoming increasingly important. Optimizations can be software- or hardware-based. Innovative hardware mechanisms have enabled new memory performance optimizations. There are myriad optimizations available, and often there is no single best optimization choice. Application and input characteristics highly influence the choice of optimizations. A model that can analytically capture the costs and benefits of these optimizations can be useful in driving decisions.

1.1 Technology Trends

Microprocessor performance has been improving exponentially according to Moore’s Law at 60% a year, while DRAM latencies are decreasing at only 7% per year. As a result, memory optimizations have become increasingly important. The performance of programs on modern systems depends on how effectively the memory system is utilized, not just on the processor features such as speed, or instruction issue width. The high latency of memory accesses, poor cache utilization, and small cache sizes relative to application working sets result in poor application performance. Increasing CPU parallelism is worsening the demands on the memory system. The traditional approach to attacking the memory system bottleneck has been to build deeper and more complex cache hierarchies. However, this approach is beginning to fail. Architectural trends suggest that caches will continue to be small
so as to keep pace with the ever-decreasing processor cycle time. McFarland [21] showed that for a feature size of 0.1 micron and 1-nsec cycle time, the L1 cache can be no bigger than 32 kilobytes to maintain a one-cycle access time, or 128 kilobytes for two. The twin goals of high capacity and low access latency seem to be in direct conflict with each other. Bridging the growing processor/memory performance gap will require innovative hardware and software solutions that use cache space and memory bandwidth more efficiently.

Software solutions for tackling the memory bottleneck have been studied widely. They include software prefetching [25], control and data restructuring techniques [5, 33, 19], cache-conscious structure layout [9], and cache-oblivious algorithms [12].

Hardware solutions for improving the memory subsystem performance include massive multithreading [1], moving processing power on to DRAM chips [28], building programmable stream buffers [22], developing configurable architectures [35], hardware-based prefetching [3] or building a smart memory controller [6].

1.2 Problem Statement

Software restructuring techniques that improve the locality of array-based applications are one important class of memory optimizations. They improve cache locality either by transforming the iteration space [5, 14, 33] or by changing the data layout [10, 19]. *Loop transformations*, an example of the former, improve performance by changing the execution order of a set of nested loops so that the temporal and spatial locality of a majority of array accesses are increased. This class of optimizations includes loop permutation, fusion, distribution, reversal and tiling [5, 33].

*Array restructuring* is a data layout transformation that improves cache performance by changing the physical layout of arrays that are accessed with poor locality [19]. Array restructuring can be static or dynamic. Static array restructuring changes the layout of an array at compile time to match the way in which
it is most often accessed; e.g., the compiler might store an array in column-major order rather than row-major order, if most accesses to the array are via column walks. Static restructuring is most useful when an array is accessed in the same way throughout the program. Dynamic restructuring creates a new array at run time, so that the new layout better matches how the data is accessed. This is most useful when access patterns change during execution, or when access patterns cannot be determined at compile time.

Dynamic array restructuring is more widely applicable than static, and we focus on it in this thesis. The run-time change in array layout is most often accomplished by copying, and we refer to this optimization as \textit{copying-based array restructuring}. We also consider the possibility of having smart memory hardware perform the restructuring [6]. Hardware mechanisms that support data remapping allow one to create array aliases that are optimal for a particular loop nest. We call this optimization \textit{remapping-based array restructuring}. The tradeoffs involved are different from those of traditional copying-based array restructuring, and thus this optimization is sometimes a useful alternative when copying is expensive. However, hardware support does not come for free, and thus there is a need to determine automatically whether hardware support should be relied upon.

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

Others have integrated data restructuring and loop transformations [10, 14], but their approaches only consider static array restructuring and do not provide any mechanisms to determine whether the integration will be profitable.

1.3 Contributions

The primary result of this research is the demonstration that compiler restructuring optimizations, hitherto combined in an ad hoc fashion, can be integrated profitably by relying on an analytical cost model to drive the decisions. The contributions of this thesis include:

- showing that remapping-based array restructuring, an optimization that exploits hardware support for remapping from a smart memory controller, is a viable alternative to copying-based array restructuring

- presenting a cost model that incorporates the costs/benefits of hardware support from a smart memory controller, which allows one to make accurate judgments about whether hardware support should be used

- demonstrating via simulation that integrated restructuring is not always a win, and that it should be applied carefully

- by modeling the memory costs of applications as a whole, we have developed an integrated analytical framework for reasoning about hardware/software trade-offs in restructuring optimizations

In this thesis, we present cost models that capture the cost/performance tradeoffs of loop transformations, copying-based array restructuring, and remapping-based restructuring. We use an integrated cost framework to decide which optimizations
to apply, either singly or in combination, for any given loop nest. Code optimized using our cost model achieves performance within 95% of the best observed for a set of eight benchmarks. In contrast, the performance of any fixed optimization is at best 76% of the best combination we observed.

1.4 Road Map

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of existing program restructuring optimizations and introduces a new hardware-based locality optimization, remapping-based array restructuring. In Chapter 3, we present an analytic framework to estimate the costs and benefits for individual restructuring optimizations. In Chapter 4, we describe our simulation methodology, our benchmark suite, and the results of our experiments. We explain the assumptions in our analytical cost framework, discuss how they affect actual estimation, and show how they influence the final decision of what optimizations(s) to employ. Chapter 5 compares our cost-model driven optimizations to existing integration strategies, with an emphasis on the difference in strategies and the tradeoffs that result. Chapter 6 presents the major conclusions of this thesis and directions for future work.
CHAPTER 2

RESTRUCTURING OPTIMIZATIONS

This chapter introduces restructuring optimizations, discusses the basic hardware mechanism that provides support for data remapping, and presents remapping-based array restructuring, a hardware-based locality optimization.

2.1 Loop Transformations

Loop transformations improve performance by changing the execution order of the individual loops in a loop nest. In other words, they transform the iteration space so that the temporal and spatial locality of a majority of array accesses in the loop nest are improved [5, 14, 33]. These optimizations include loop permutation, loop fusion, loop distribution, loop reversal and loop tiling. We do not study tiling because the analysis required for calculating cross-interference and self-interference misses is outside the scope of this research.

A simple example where loop permutation is effective is matrix multiplication. Consider the matrix multiplication code shown in Figure 2.1. In this code, array $B$ is accessed with a stride of $8 \times N$ bytes. This results in very poor TLB and cache behavior. Cache performance degrades significantly when the column length of this array exceeds the L1 cache size. For example, for a 32 KB L1 cache with 32 byte

```c
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 2.1. Matrix multiply kernel
cache lines and 8 byte array elements, cache performance is affected when a single column walk’s length exceeds 1024. TLB performance degrades very rapidly even for small values of \(N\). To see why, assume a TLB of 128 entries where each entry maps a 4 kilobyte page. In one column walk (corresponding to one vertical line in Figure 2.2), the span of memory touched by the array \(B\) is \(8 \times N^2\) bytes, which is \(\frac{8 \times N^2}{1024}\) pages. When \(\frac{8 \times N^2}{1024} > 128\) (i.e., \(N > 216\)), the number of pages touched exceeds 128 and the TLB starts thrashing. Permuting the loops from “\(ijk\)” to “\(ikj\)” order results in zero or unit stride for all arrays. All the arrays now access memory in row-major fashion as shown in Figure 2.3, which reduces the TLB and the cache footprint significantly and consequently improves performance. For large arrays, the speedup can be quite significant.

For the simple case of matrix multiplication, loop permutation improves performance. However, there are loops for which loop restructuring is not feasible either because interchanging the loops would violate program dependencies or not improve the locality of all arrays in the loop nest. We say that an array is in \textit{loop order} if it has a unit or zero stride in the loop nest. Loop nests that have conflicting memory accesses may not be able to achieve loop order for all arrays in the loop

\[
\begin{array}{c}
C
\end{array}
= \begin{array}{c}
A
\end{array} \times \begin{array}{c}
B
\end{array}
\]

\textbf{Figure 2.2.} Direction of array accesses in the innermost loop with the original “\(ijk\)” loop order.

\[
\begin{array}{c}
C
\end{array}
= \begin{array}{c}
A
\end{array} \times \begin{array}{c}
B
\end{array}
\]

\textbf{Figure 2.3.} Direction of array accesses in the innermost loop with the optimized “\(ikj\)” loop order.
under any loop transformation.

Carr *et al.* [5] demonstrate how to analyze loop transformations using a simple cost model based on the estimated number of cache lines accessed by a loop nest. They define *loop cost* to be the estimated number of cache lines accessed by all arrays when a given loop is placed innermost in the nest. Evaluating the loop cost for each loop and ranking the loops in descending cost order (subject to legality constraints) yields a permutation with least cost. The resulting loop nest is said to be in *memory order.* Applying this cost model to the matrix multiply example loop in Figure 2.1 gives the costs shown in Table 2.1. Since the loop costs in descending order are $i > k > j$, the recommended loop permutation is $ikj$. Arrays $A$, $B$, and $C$ are now all accessed sequentially.

### 2.2 Copying-based Array Restructuring

Many scientific applications contain loop nests with accesses to multidimensional arrays with non-unit strides. Copying-based array restructuring directly improves the spatial locality of such array accesses by changing the way the array is stored in memory. Spatial locality determines how much reuse there is within a cache line once it is brought into the cache. Words within a cache line are stored in the order in which they are stored in memory. If an array is in loop order, then the way the array is stored in memory matches its access pattern. In such a case, every word in the cache line is used before the cache line is evicted, which leads to excellent

<table>
<thead>
<tr>
<th>array references</th>
<th>innermost loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[i][k]$</td>
<td>$N \times N^2$</td>
</tr>
<tr>
<td>$B[k][j]$</td>
<td>$1 \times N^2$</td>
</tr>
<tr>
<td>$C[i][j]$</td>
<td>$N \times N^2$</td>
</tr>
<tr>
<td>total</td>
<td>$2N^3 + N^2$</td>
</tr>
</tbody>
</table>

Table 2.1. Cost Model of McKinley *et al.* Applied to the Matrix Multiply Example
spatial locality. For example, one might store one array in row-major order if it is accessed row-by-row and another array in column-major order if it is accessed column-by-column. Laying out the arrays in this way enhances spatial locality. Array restructuring generalizes this idea by restructuring arrays so that they are stored in the direction of the regular strided accesses of the original array.

Consider the simple example loop nest \textit{ir\_kernel} in Figure 2.4. If we assume row-major storage, arrays \(U\) and \(V\) are accessed sequentially, with \(U\) having good spatial locality, and \(V\) having good temporal locality. Array \(W\) has good temporal locality in the innermost loop, but is accessed along its columns in the outer loops. Array \(X\) is accessed diagonally and will not enjoy good cache or TLB performance. Array restructuring can optimize this loop by creating new arrays \(cX\) and \(cW\) (copied versions of \(X\) and \(W\) respectively), where \(cW\) is the transpose of array \(W\) and \(cX\) intuitively is the array \(X\) laid out diagonally (see Figure 2.5). Figure 2.6 shows the optimized code after array restructuring is applied.

The creation of the new arrays, \(cX\) and \(cW\), are guided by index transformation matrices that specifies how the indices of the new array relate to that of the original array. Leung and Zahorjan derived the formalisms needed to create index transformation matrices [20]. In this case, the array \(X\) is transformed such that \(cX[i - j + N][j]\) maps to \(X[i][j]\) and array \(W\) such that \(cW[j][i]\) maps to \(W[i][j]\). Figure 2.6 shows the code optimized using copying-based array restructuring. This optimization may waste storage, as illustrated by the shaded regions of the re-
Create new array such that access direction is horizontal (i.e., match row-major storage order)

Array Restructuring

\[ X[i,j] \rightarrow cX[i-j+N][j] \]
Thus, \[ X[i+j+k][k] \rightarrow cX[i+j+N][k] \]

The shaded regions in the array \( cX \) are unused/wasted.

Figure 2.5. Visualizing array restructuring of \textit{ir\_kernel}.

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

Figure 2.6. \textit{ir\_kernel} optimized using copying-based array restructuring.
structured array in Figure 2.5. The amount of unused memory in array $cX$ is $4N^2 - 3N^2 = N^2$. Leung and Zahorjan show that using further transformations, one can reduce the amount of unused memory. In the worst case, their algorithm bounds the size of the restructured array to $n!$ times the size of the original array, where $n$ is the dimensionality of the array [20].

Array $cX$ has stride one accesses in the innermost loop and thus exhibits better spatial locality than array $X$ in the original code. Similarly, array $cW$ has improved locality because of stride one accesses in the outer loops. Whether the transformation improves performance depends on the input parameters. If the size of the array is much larger than the amount of use it has in the loop nest, then the setup costs of creating the new array might dominate the benefits of improved spatial locality of the restructured array. In Section 3.1.3, we will analytically formulate when array restructuring can be used profitably.

### 2.3 The Impulse Memory System

The Impulse adaptable memory system expands the traditional virtual memory hierarchy by adding address translation hardware to the main memory controller (MMC) [6, 32, 34]. Impulse uses physical addresses unused in conventional systems as remapped aliases of real physical addresses. For instance, in a system with 32-bit physical addresses and one gigabyte 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 2.7 shows how addresses are mapped in an Impulse system. The real physical address space is directly backed up 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 must be remapped to real physical addresses through the Impulse MMC. How the MMC interprets shadow addresses presented to it is configured by the operating system.
This virtualization of unused physical addresses can provide different views of data stored in physical memory to programs. For example, it can create cache-friendly data structures to improve the efficiency of the processor caches. 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 the compiler) inserts appropriate system calls into the application code to configure the memory controller.

To map a data item in the shadow address space to the physical memory, the Impulse MMC must first recover its virtual address. To avoid directly handling virtual addresses at the MMC, we require that the virtual address must be located inside a special virtual region. The OS creates a dense, flat page table in contiguous physical addresses for the special virtual region. We call this page table the memory controller page table. The OS then pins down this page table in main memory and sends its starting physical address to the memory controller so that the MMC can access it without interrupting the OS. Since data items in a shadow region are mapped to a special virtual region, the MMC only need compute offsets relative to the starting address of the virtual region. We call such an offset a pseudo-virtual address. For each shadow data item, the MMC first computes its pseudo-virtual address, then uses the memory controller page table to determine the data item’s real physical address. To speed up the translation from pseudo-virtual to physical
addresses, the MMC uses an TLB to store recently used translations. We call this TLB the MTLB.

Figure 2.8 shows a simplified block diagram of the Impulse memory system. The critical component of the Impulse MMC is the shadow engine, which processes all shadow accesses. The shadow engine contains a small scatter/gather SRAM buffer used as a place to scatter/gather cache lines in the shadow address space, some control registers to store remapping configuration information, an ALU unit (AddrCalc) to translate shadow addresses to pseudo-virtual addresses, and a Memory Controller Translation Lookaside Buffer (MTLB) to cache recently used translations from pseudo-virtual addresses to physical addresses. The control registers are split into eight different sets and are capable of saving configuration information for eight different mappings. However, all mappings share the same ALU unit and the same MTLB.

To understand Impulse’s address remapping functionality better, consider the following example where the application desires to create a virtual transpose of a two-dimensional array, say $A$. The application allocates a contiguous range of virtual addresses large enough to map the elements of $A$, and makes a system call to

![Figure 2.8. Impulse architecture.](image-url)
create the virtual transpose, \( rA \). It also passes along relevant control information such as the starting virtual address of the original array, the size of each element in the array, and the dimensions of the array to the OS. In response to the system call, the OS first allocates a sufficiently large range of shadow physical addresses to contain the remapped data structure, e.g., the transposed array. It then configures the Impulse MMC to respond to accesses within this shadow physical address range appropriately. This configuration takes two steps: setting up a dense page table and then initializing a set of configuration registers on the MMC. To configure the MMC, the OS performs IO writes to indicate remapping-specific information such as what address range is being configured, what kind of remapping the MMC should perform (e.g., matrix transposition), the size of the elements being remapped, and the location of the corresponding page table. Finally, after configuring the MMC, the OS maps the shadow physical addresses to an unused portion of the application’s virtual address space and returns a pointer to where this range starts to the user process. This address is \( rA \).

When the user process accesses an address corresponding to a remapped data structure, e.g., the virtual transpose array, the processor converts this virtual address to the corresponding shadow physical address using its MMU. When the Impulse MMC sees a read (or write) request to a valid shadow address, the AddrCalc unit of the Impulse MMC uses the remapping information stored in the shadow descriptor to translate the shadow address into a set of pseudo-virtual addresses. The MMC then uses the memory controller page table to determine the real physical addresses corresponding to the pseudo-virtual addresses and passes these physical addresses to an optimized DRAM scheduler. The DRAM scheduler orders and issues the reads, loads the data residing at these physical addresses into an output buffer, and sends the gathered data back to the processor to satisfy the outstanding read request. More detailed examples can be found elsewhere [6, 34].
In the next section, we introduce a new data restructuring optimization that is implemented using the Impulse remapping mechanism.

2.4 Remapping-based Array Restructuring

Array restructuring is particularly well-suited for the Impulse memory system. Instead of copying the original array elements to form a new array, a new array can be created virtually using the additional level of indirection that Impulse provides. We call this optimization remapping-based array restructuring.

Remapping yields several performance benefits. The processor achieves a higher cache hit rate, because several useful (originally noncontiguous) elements are loaded into the caches at once. Furthermore, sending only needed elements over the bus consumes less bus bandwidth and results in more efficient use of cache space. The TLB hit rate also increases because the now sequential accesses span fewer pages. Remapping is not always beneficial, though, and this is discussed at the end of this section.

Remapping-based array restructuring is similar in spirit to copying-based array restructuring, in that the goal is to create a new array in such a way that the noncontiguous accesses of the original array correspond to sequential accesses in this array. The difference is in its implementation.

2.4.1 Implementation Details

We use two kinds of remapping mechanisms, transpose and base-stride, to implement remapping-based array restructuring. The transpose remapping is used when a virtual transpose of a two dimensional array is needed. The base-stride remapping is more general and is used to map a dense alias array to a strided data structure. Consider a set of diagonal accesses (such as array $X$'s accesses in the loop in Figure 2.4) that need to be remapped. Each access in the diagonal is with a fixed stride from the start of the diagonal. For each diagonal, the base is different.
Thus, before accessing each new diagonal, we need to specify a new base address to the memory controller. By calculating the base as an offset from the starting of the array, we can change the base by changing the offset.

Figure 2.9 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 \( X+offset+k \times \text{stride} \), where stride is \( N+1 \). The offset, which is \( (i+j) \times N \), 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 \( X+(i+j) \times N+k \times (N+1) \), which is simply the original reference \( X[i+j+k][k] \). We flush \( rX \) from the cache to maintain coherence when the offset changes.

After applying the remapping-based restructuring optimization, all accesses are in array order. The cost of setting up a remapping is quite small compared to copying. However, subsequent accesses to the remapped data structure are slower than accesses to an array restructured via copying, because the memory controller must retranslate addresses on the fly and “gather” cache lines from

```c
double U[N], V[N], W[N][N], X[3N][N];
double *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 2.9. ir_kernel optimized using remapping-based array restructuring.
disjoint regions of physical memory. Thus, the amount of use there is in a loop
nest directly influences the overhead of using remapping-based array restructuring.
Often, when copying-based array restructuring is not viable, remapping-based array
restructuring can be worthwhile. In such a case, 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 remapping or copying-based
array restructuring should be used, if at all. In the next chapter, we present a
framework that analytically captures these tradeoffs.
CHAPTER 3

ANALYTIC FRAMEWORK

A compiler performs two steps to optimize a program. First, it chooses a part of the program to optimize and a particular transformation to apply to it. Second, it transforms the program and verifies that the transformation does not change the meaning of the program or changes it in a way that is acceptable to the user. Bacon et al. [2] compare the first step to a black art, because it is difficult and poorly understood. In this section, we explore how to choose what part of a program to optimize for the domain of restructuring optimizations. We present analytic cost models for each of the optimizations and analyze tradeoffs in individual restructuring optimizations, both qualitatively and quantitatively. We show why it is beneficial to consider them in an integrated manner, and present a cost framework that allows integrated restructuring optimizations to be evaluated.

3.1 Modeling the Restructuring Strategies

3.1.1 Basic Model

We need careful cost/benefit analysis to determine when restructuring optimizations may be applied profitably. Finding the best choice of optimizations to apply via detailed simulation or hardware measurements is time-consuming and expensive. An analytical model that provides a fairly accurate estimate of the cost/benefit tradeoffs between various optimizations is therefore desirable. We have developed such an analytic model to estimate the memory costs of applications at compile time. Like Carr, McKinley and Tseng [5], we estimate the number of cache lines
accessed in the loop nests and use this estimate to choose which optimization(s) to apply.

The memory cost of an array reference, say $R_{\alpha}$, in a loop nest is directly proportional to the number of cache lines accessed by it. Consider the array reference $B$ in the loop nest in Figure 3.1. This array is accessed $N$ times in the loop nest in a sequential fashion. If this loop nest is to be executed on a machine with a cache whose line size is 128 bytes, then the number of cache lines accessed by the reference $B$ is $\frac{N}{128}$ (assuming that the storage requirement for a “double” is 8 bytes). The memory cost of array reference $B$ is proportional to this number.

In general, if $cls$ is the line size of the cache closest to memory, $stride$ is the distance between successive accesses of array reference $R_{\alpha}$ in a loop nest, and $f$ is the fraction of the cache lines reused from a previous iteration of the innermost loop, then the memory cost of array reference $R_{\alpha}$ in the loop nest is estimated to be:

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

(3.1)

We do not have a framework for estimating $f$ accurately. We expect $f$ to be very small for large working sets, which allows us to neglect the $(1 - f)$ term without introducing significant inaccuracies (see the limitations discussed in Section 4.4).

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++)

Figure 3.1. Example loop used to illustrate the cost model.
Using Equation 3.1, we see that the memory cost of array reference $C$ in the loop nest in Figure 3.1 is $N^3 \max(\frac{1}{\text{I}}, 1)$.

We estimate the memory cost of a loop nest to be the sum of the memory costs of the independent array references in the nest. We define two array references to be independent if they access different cache lines in each iteration. This characterization is necessary to avoid over-counting cache lines. For example, references $A[i]$ and $A[i + 1]$ are not independent as we assume that they access the same cache line. However, 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 (the total number of iterations of the nest), the spatial and temporal locality of the array references, the stride of the arrays and the cache line size. We estimate the memory cost of the $i^{th}$ loop nest in the program to be:

$$\text{MemoryCost}(L_i) = \sum_{\alpha:\text{independentRef}} \text{MemoryCost}(R_\alpha) \quad (3.2)$$

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

$$\text{MemoryCost(program)} = \sum_{i=1}^{n} \text{MemoryCost}(L_i) \quad (3.3)$$

Using the above equations, we see that the total memory cost of the loop in Figure 3.1 is $(N^3 \times (1 + \frac{1}{\text{I}}) + 2 \times \frac{N}{\text{I}})$.

The goal of the cost model is to let us compute a metric for choosing the combination of array and loop restructuring optimizations that has the 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 will know the number
of times each loop nest executes and also 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, we expect to be able to make empirical assumptions about the
frequencies of loop nests. We also assume that the cache line size of the cache
closest to memory is known to the compiler.

3.1.2 Modeling Loop Transformation

When we consider only loop transformations, the recommendations from our
model are usually the same as those of the simpler model by McKinley et al. [23].
However, their model offers no guidance to drive data restructuring or integrated
restructuring. We consider the total memory cost of the loop nests, which allows
us to compare the cost of loop transformations with that of independent optimiza-
tions, e.g., copying or remapping-based array restructuring, or any combination of
restructuring optimizations.

3.1.3 Modeling Copying-based Array Restructuring

The cost of copying-based array restructuring is the sum of the cost of creating
the new array and the cost of executing the optimized loop nest. There is the
additional cost of copying data back to the original array if the new array was
modified. In our cost model, all costs are modeled in terms of memory costs as
described in Section 3.1.1. The cost of setup is the sum of the memory costs of the
original array and the new array in the setup loop.

\[
MemoryCost(copyingSetup) = \text{originalArraySize} \times (min\left(\frac{\text{newArrayStride}}{\text{cls}}, 1\right) + \frac{1}{\text{cls}})
\]  
(3.4)
For the code shown in Figure 2.6, the memory cost of creating array \( cX \) in the setup loop is \( \frac{3N^2}{\text{max}(\frac{cls}{N+1})} \) and the memory cost of array \( X \) in the setup loop is \( \frac{3N^2}{cls} \). Thus, the setup cost of creating array \( cX \) is \( 3N^2 \times (1 + \frac{1}{cls}) \) if \( (N+1) > cls \), which is usually the case. The calculation for array \( cW \) is similar.

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

\[
MemoryCost(\text{restructuredReference}) = \text{loop Trip Count} \times \frac{1}{cls}
\]

(3.5)

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

\[
MemoryCost(\text{copyingSetup}) + MemoryCost(\text{restructuredReference}) < MemoryCost(\text{originalReference})
\]

(3.6)

For this example, the total cost of the array-restructured program (assuming \( cls = 16 \)) is \((\frac{N^3}{8} + \frac{6N^2}{16} + \frac{N}{16})\), 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 this decision out when array sizes are larger than the \( L2 \) cache.

### 3.1.4 Modeling Remapping-based Restructuring

When using remapping-based hardware support, we can no longer model the memory costs of an application as being directly proportional to the number of cache lines accessed, since all cache line fills no longer incur the same cost. Cache line fills to remapped addresses undergo a further level of translation at the memory controller, as explained in Section 2.4. After translation, the corresponding physical addresses need not be sequential, and thus the cost of gathering a remapped cache line depends on the stride of the array, the cache line size of the cache closest to memory, and the efficiency of the DRAM scheduler. To accommodate this variance in cache line gathering costs, we model the total memory cost of an application
as proportional to the number of cache lines gathered times the cost of gathering the cache line. The cost of gathering a normal cache line, $G_c$, is fixed, and the cost of gathering a remapped cache line, $G_r$, is fixed for a given stride. We used a series of microbenchmarks to compute a table of $G_r$ values indexed by stride that we consult to determine the cost of gathering a remapped cache line. Thus, if a program accesses $n_c$ normal cache lines, $n_1$ remapped cache lines remapped with stride $s_1$, $n_2$ remapped cache lines remapped with stride $s_2$, and so on, then the memory cost of the program is modeled as:

$$MemoryCost(program) = n_c \times G_c + \sum_{i} n_i \times G_r(s_i)$$  \hspace{1cm} (3.7)$$

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 cache flushes. 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 $K_1 \times \#elementsToBeRemapped$. Repeatedly 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 also model the costs of flushing as being proportional to the number of cache lines flushed where the constant of proportionality is $K_3$. We have empirically estimated these constants using microbenchmarks.

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

$$MemoryCost(remappedReference) = \left( \frac{loopTripCount}{\max(\frac{ds}{\text{stride}}, 1)} \times (1 - f) \right) \times G_r(\text{stride})$$ \hspace{1cm} (3.8)$$
\begin{align*}
    MemoryCost(\text{normalReference}) &= \left( \frac{\text{loopTripCount}}{\max\left( \frac{\text{cls}}{\text{stride}}, 1 \right)} \times (1 - f) \right) \times G_c \quad (3.9)
\end{align*}

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

Returning to our example in Figure 2.9, 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 restructuring optimized loop is \( \frac{1}{16}(N^3 + N) \times G_c + \frac{1}{16}N^3 \times G_r(N + 1) + K_1 \times 3N^2 + K_2 \times N^2 + K_3 \times N \times N^2 \). This is less than the original program’s memory cost, and thus remapping-based array restructuring is assumed to be profitable.

### 3.1.5 Estimating the Cost of Gathering Cache Lines

The cost of gathering a remapped cache line, \( G_r \), is mainly influenced by two factors: the DRAM banks organization and the MTLB hit rate. We use the microbenchmark illustrated in Figure 3.2 to calculate the cost of gathering a cache line, remapped using the base-stride mechanism, for all strides between 1 and 2048 of an integer array.

Figure 3.3 shows how \( G_r \) varies with the stride (in bytes). The spikes occur whenever the stride is a power of 2 greater than 32. In these cases, the remapped accesses all hit the same memory bank causing poor performance. We calculate \( G_r \) for strides ranging from 4 to 8192 bytes. The average value of \( G_r \) is 596 cycles and the range was 429 cycles to 1160 cycles. Since \( G_r \) varies significantly with the stride, our cost model uses a reference database to lookup \( G_r \) for each stride.
bs_args.newaddr = (unsigned *) &rA;
bs_args.vaddr = A;
bs_args.count = N*STRIDE;
bs_args.objsize = sizeof(int);
bs_args.stride = STRIDE*sizeof(int);
ams_mapshadow(AMS_TYPE_BASESTRIDE, &bs_args)
for(gather_remapped=0, i=0;i<50;i++){
    t1 = read_clock();
    temp = rA[65*i];
    t2 = read_clock();
    sum += temp;
    gather_remapped += t2 - t1;
}
gather_remapped = gather_remapped/i;

Figure 3.2. Microbenchmark used to compute $G_r$, the cost of gathering a remapped cache line (using the base stride mechanism) for various strides

Figure 3.3. Graph showing the relationship between $G_r$ and the stride.
Figure 3.4 shows the microbenchmark for calculating the cost of gathering a normal (i.e., unremapped) cacheline. We chose a stride of 65 so that we could be sure that each of the 50 accesses would miss in the cache and go to memory (65*4 bytes < cache line size). Figure 3.5 shows how $G_c$ varies with the stride (in bytes). $G_c$ is a constant (48 cycles) for any stride. There is no variance because every access has the same memory bank behavior. Bank parallelism is maximum because each DRAM bank supplies an equal number of elements to the gathered cache line.

```
for(gather_normal=0, i=0; i<50; i++){
    t1 = read_clock();
    temp = A[65*i];
    t2 = read_clock();
    sum += temp;
    gather_normal += t2 - t1;
}
gather_normal = gather_normal/i;
```

Figure 3.4. Microbenchmark used to compute $G_c$, the cost of gathering a normal cacheline.

![Fig 3.4](image)

Figure 3.5. Graph showing the relationship between $G_c$ and the stride (in bytes).
3.2 Integrating the Restructuring Optimizations

By *integrated restructuring* we mean two things. First, the individual restructuring optimizations can be combined to optimize an application in complementary ways. For example, we could combine loop permutation, loop fusion, and copying-based array restructuring to optimize one single loop nest. Second, we should be able to select a good combination of restructuring optimizations from the legal options given the application and its inputs. Researchers have provided heuristics-driven algorithms for combining loop and array restructuring [10, 14] but there has not been any work on providing a framework for choosing the right set of optimizations among restructuring optimizations. Our cost model framework for selecting the right combination of restructuring optimizations is based on Equations 3.1 – 3.9. These equations allow us to decide which optimization to choose for a given application and input size.

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

We analyze the example loop nest in Figure 2.4 to see whether integrated restructuring improves performance. Recall that loop transformation could not improve the locality of array $X$, and array restructuring incurred the costs of creating the arrays $cX$ and $cW$ through copying. Consider the integrated restructuring optimization that permutes the loop nest into $jik$ order and uses array restructuring to improve the locality of $X$. Figure 3.6 shows the optimized code. This combination of optimizations achieves array order for all references and incurs
double U[N], V[N], W[N][N], X[3N][N];
double cX[4N] [N];
for (i=0; i<3*N; i++)
    for (j=0; j<N; j++)
        cX[i+j*N][j]=X[i][j];
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        for (k=0; k<N; k++)
            U[k]+= V[i]+ W[j][i]+cX[i+j*N][k];

Figure 3.6. \textit{ir-kernel}, optimized by a combination of loop permutation and copying-based array restructuring.

only the setup cost of creating \( cX \), since \( W \) is already in array order after loop transformation. As we will see in the following chapter, integrated restructuring delivers slightly better performance than either loop or array restructuring alone for this example.

The same loop nest can be optimized using loop transformation with remapping-based restructuring (\( \text{L}+\text{R} \)), in which case the loop transformation would bring array \( W \) into array order, and array \( X \) would be remapped to \( rX \), as in Figure 2.9.

In general, selecting the optimal set of transformations is a nontrivial problem. Finding the combination of loop fusion optimizations alone for optimal temporal locality has been shown to be NP-hard [16]. The problem of finding the optimal data layout between different phases of a program has also been proven to be NP-complete [17]. As a result, researchers have used heuristics to make integrated restructuring tractable. Our analytical cost framework can help evaluate various optimizations, and we show later that cost model driven optimizations comes closer to best possible performance than any fixed optimization strategy.
CHAPTER 4

EVALUATION

To evaluate the effectiveness of our cost models, we used URSIM, an execution-driven simulator derived from RSIM [27]. URSIM models in detail a microprocessor similar to a MIPS R10000, a split-transaction MIPS R10000 bus that supports a snoopy coherence protocol, and the Impulse memory controller [24]. The processor modeled is four-way issue, out-of-order, and superscalar with a 64-entry instruction window. The L1 data cache is 32KB, nonblocking, write-back, virtually indexed, physically tagged, two-way associative, with 32-byte lines and has a one-cycle latency. The L2 data cache is 512KB nonblocking, write-back, physically indexed, physically tagged, two-way set associative, with 128-byte lines and has an eight-cycle latency. The instruction cache is assumed to be perfect. The TLB maps both instructions and data, has 128 entries, and is single-cycle, fully associative, and software-managed. The bus multiplexes addresses and data, is eight bytes wide, and has a three-cycle arbitration delay and a one-cycle turn-around time. The system bus, memory controller, and DRAMs all run at one-third the CPU clock rate. The memory supports critical word first and returns the critical quad-word for a load request 16 bus cycles after the corresponding L2 cache miss. The memory system models eight banks, pairs of which share an eight-byte wide bus between DRAM and the MMC.

4.1 Benchmarks

To evaluate the performance of our cost-model driven integration strategy, we studied eight benchmarks used in previous studies of loop and/or data restructuring.
Four of the benchmarks that we considered (matmult, syr2k, ex1, and the ir_kernel discussed in Section 2.2) have been studied previously in the context of data restructuring – the former two by Leung et al. [19] and the latter two by Kandemir et al. [14]. Three other applications – btrix, vpenta, cfft2d – are NAS kernels that have been evaluated in the context of both data and loop restructuring in isolation [5, 15, 19]. Finally, kernel6 is the sixth Livermore Fortran kernel.

Table 4.1 shows which optimizations we considered for each benchmark. The optimization candidates are copying-based array restructuring(C), remapping-based restructuring(R), loop transformations(L), a combination of loop and copying-based restructuring(L+C), and a combination of loop and remapping-based restructuring(L+R). √ indicates that the optimization was possible, N indicates that the optimization was not needed, and I indicates that the optimization was either illegal or inapplicable. In our study, we hand-coded all optimizations; work is ongoing to add our cost model to the Scale compiler [31] to automate the transformations. We ran each benchmark with a range of input sizes, with the smallest input size typically just fitting into the L2 cache. Whenever there were several choices for a single restructuring strategy, we chose the best option. In other words, the results that we report for L is for the best loop transformation (loop permutation, fusion,

<table>
<thead>
<tr>
<th>Application</th>
<th>C</th>
<th>R</th>
<th>L</th>
<th>L+C</th>
<th>L+R</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmult</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>syr2k</td>
<td>√</td>
<td>√</td>
<td>N</td>
<td>N</td>
<td>√</td>
</tr>
<tr>
<td>vpenta</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>N</td>
<td>√</td>
</tr>
<tr>
<td>btrix</td>
<td>√</td>
<td>I</td>
<td>√</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>cfft2d</td>
<td>√</td>
<td>√</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>ex1</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>ir_kernel</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>kernel6</td>
<td>√</td>
<td>√</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
</tbody>
</table>
distribution or reversal) among the ones we implemented. Similarly, the results for \( L+R \) is for the best combination of loop and remapping-based array restructuring that we implemented.

### 4.2 Results

In this section, we briefly analyze each benchmark and validate the effectiveness of our cost model. We report the results of our experiments in Table 4.2. For each input size, we simulate the performance of the base (unoptimized) benchmark and the best optimized version for each candidate optimization. Our primary performance metric is the geometric mean speedup obtained for each optimization compared to the baseline benchmark over the range of input sizes. We also present the results obtained when our cost model is used to select dynamically which optimization to perform for a given input size (CM-driven). We compare the CM-driven optimization with BEST, which is the post-facto best optimization. We refer to best performance as that resulting from making the best choices among the optimizations we consider.

As can be seen from Table 4.2, CM-driven optimization achieves an average of 95% of BEST's speedup (1.68 versus 1.77) whereas the best single optimization,

<table>
<thead>
<tr>
<th>Application</th>
<th>C</th>
<th>R</th>
<th>L</th>
<th>L+C</th>
<th>L+R</th>
<th>CM-driven</th>
<th>BEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmult</td>
<td>1.26</td>
<td>1.24</td>
<td>1.34</td>
<td>-</td>
<td>-</td>
<td>1.34</td>
<td>1.46</td>
</tr>
<tr>
<td>syr2k</td>
<td>1.71</td>
<td>0.75</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.56</td>
<td>1.88</td>
</tr>
<tr>
<td>vpenta</td>
<td>0.60</td>
<td>1.12</td>
<td>1.47</td>
<td>-</td>
<td>1.61</td>
<td>1.54</td>
<td>1.65</td>
</tr>
<tr>
<td>btrix</td>
<td>1.49</td>
<td>-</td>
<td>1.72</td>
<td>1.67</td>
<td>1.47</td>
<td>1.72</td>
<td>1.80</td>
</tr>
<tr>
<td>cfft2d</td>
<td>2.77</td>
<td>2.90</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.90</td>
<td>2.91</td>
</tr>
<tr>
<td>ex1</td>
<td>1.53</td>
<td>0.71</td>
<td>0.66</td>
<td>0.76</td>
<td>0.43</td>
<td>1.53</td>
<td>1.53</td>
</tr>
<tr>
<td>ir_kernel</td>
<td>1.07</td>
<td>1.26</td>
<td>0.96</td>
<td>1.07</td>
<td>1.26</td>
<td>1.07</td>
<td>1.26</td>
</tr>
<tr>
<td>kernel6</td>
<td>1.24</td>
<td>1.92</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.99</td>
<td>2.00</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>1.35</strong></td>
<td><strong>1.23</strong></td>
<td><strong>1.10</strong></td>
<td><strong>1.04</strong></td>
<td><strong>1.09</strong></td>
<td><strong>1.68</strong></td>
<td><strong>1.77</strong></td>
</tr>
</tbody>
</table>
copying-based array restructuring, obtains only 77% of **BEST**'s speedup (1.35 versus 1.77). The reason for the good performance of cost model driven optimization is that the best optimization strategy is highly application and input dependent. Even within the same benchmark, the best choice is dependent on the size of the input data. For example, in *syr2k*, the cost-model was able to choose between **C** and **L+R** and achieved a higher speedup (1.88) than either **C** or **L+R** (i.e., it picked **C** when **L+R** performed poorly, and vice-versa). Overall, our cost model was usually able to select the correct strategy to employ, and when it failed to pick the best strategy, the choice it made was generally very close to **BEST**. To better understand why the cost model worked well in most cases, and poorly in a few, we will discuss each benchmark program in turn.

### 4.2.1 MATMULT

*MATMULT* involves multiplying a *N* by *M* matrix, *B[N][M]*, by an *M* by *L* matrix, *A[M][L]*, to get the product matrix, *C[N][L]* as shown in Figure 4.1. This code performs poorly with large arrays due to high TLB and cache miss rates. Remapping-based or copying-based array restructuring get unit stride access for all of these arrays. This is the only application for which loop permutation alone is sufficient to achieve unit stride for all arrays in the loop nest. While array restructuring improves the performance of *MATMULT*, loop restructuring can do so with lower setup costs. Our cost model recognized this situation and recommended loop permutation over the other choices. Though loop permutation was the best

```c

double A[N][M], 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[k][i]*B[k][j];

Figure 4.1. MATMULT kernel
```
choice for a majority of cases, conflict misses caused remapping to be better for some data sizes. Since conflict misses are not modeled, the cost model incorrectly recommended loop permutation for such cases, and achieved a lower speedup than the best possible one. Table 4.3 shows the characteristics of each choice in our experiments.

The cost model does not achieve the best possible performance because it overestimates the original code's memory costs for small data sets. It does not consider cache reuse between iterations of the outer loops. This example illustrates a source of error for our cost model. The performance penalty induced by this limitation in our cost model is low because the memory performance is not a significant factor when there is high locality. When the array sizes are much larger than the cache, the cost model correctly prefers L to O. From Table 4.2, we see that if the optimal choice had been made for each data set, the speedup would have been 46%, while the cost model achieves a mean-speedup of 34% over the original program, which is 91.8% of optimal.

Matrix multiplication code has historically been among the most studied codes in scientific computing. There are more advanced techniques to solve this problem: loop tiling [18], nonlinear array layouts [8], recursive matrix multiplication [11], and using superpages to map the entire array with a single TLB entry [26].

Table 4.3. Performance Characteristics of the Optimizations for MATMULT

<table>
<thead>
<tr>
<th>MATMULT</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>54</td>
<td>5</td>
<td>11</td>
<td>3 2-D</td>
<td>90.9</td>
<td>6.8</td>
<td>2.25</td>
<td>98.9</td>
<td>1.00</td>
</tr>
<tr>
<td>remap</td>
<td>72</td>
<td>5</td>
<td>11</td>
<td>1 1-D 3 2-D</td>
<td>98.8</td>
<td>0.2</td>
<td>1.05</td>
<td>99.9</td>
<td>1.24</td>
</tr>
<tr>
<td>copy</td>
<td>54</td>
<td>6</td>
<td>13</td>
<td>4 2-D</td>
<td>95.2</td>
<td>3.8</td>
<td>0.98</td>
<td>99.9</td>
<td>1.26</td>
</tr>
<tr>
<td>pureloop</td>
<td>53</td>
<td>5</td>
<td>11</td>
<td>3 2-D</td>
<td>95.3</td>
<td>3.7</td>
<td>0.96</td>
<td>99.9</td>
<td>1.34</td>
</tr>
</tbody>
</table>
4.2.2 SYR2K

The syr2k subroutine from the BLAS library is a banded matrix calculation that computes $C = \alpha A^T B + \alpha B^T A + C$. The arrays are $N \times N$ matrices and the width of the band in the matrix is $b$. Figure 4.2 shows the core kernel of the banded matrix calculation. The core loop references four array elements from different rows and columns during each iteration of the innermost loop, which results in poor cache and TLB hit rates for the baseline program. Data dependences negate the possibility of using loop permutation. We combine loop distribution and remapping-based array restructuring ($L+R$) to optimize this subroutine. Copying-based array restructuring ($C$) is also very effective at creating sequential access patterns. Table 4.4 shows the characteristics of each optimization.

The number of accesses to the elements in the band is $O(Nb^2)$. Thus copying performs well for large bands, where the fixed setup cost of copying ($O(N^2)$) is

```c
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 4.2. SYR2K kernel

<table>
<thead>
<tr>
<th>SYR2K</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>54</td>
<td>3</td>
<td>7</td>
<td>3 2-D</td>
<td>79.1</td>
<td>16.8</td>
<td>4.1</td>
<td>99.1</td>
<td>1.00</td>
</tr>
<tr>
<td>remap</td>
<td>105</td>
<td>3</td>
<td>7</td>
<td>4 1-D</td>
<td>89.4</td>
<td>10.0</td>
<td>0.6</td>
<td>99.9</td>
<td>0.75</td>
</tr>
<tr>
<td>copy</td>
<td>63</td>
<td>4</td>
<td>9</td>
<td>5 2-D</td>
<td>91.7</td>
<td>7.3</td>
<td>1.0</td>
<td>99.5</td>
<td>1.71</td>
</tr>
<tr>
<td>loop + remap</td>
<td>105</td>
<td>3</td>
<td>7</td>
<td>6 2-D</td>
<td>96.0</td>
<td>3.6</td>
<td>0.4</td>
<td>99.9</td>
<td>1.56</td>
</tr>
</tbody>
</table>

Table 4.4. Performance Characteristics of the Optimizations for SYR2K
amortized by the subsequent reuse, and remapping performs well for small bands. The cost model correctly identified this behavior in most, but not all, cases. By correctly choosing when to apply copying (C) and when to choose combined loop and remapping-based restructuring (L+R), the cost model driven optimizations achieved better performance than either in isolation.

4.2.3 VPENTA

The VPENTA subroutine accesses eight two-dimensional arrays in seven loop nests with large strides. Loop permutation can be used to optimize the two most expensive loop nests. The remaining loop 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 and remapping-based array restructuring (L+R) results in the best overall performance. The impact of additional array restructuring is not very high because the five loop nests where remapping was done constitute only a small portion of the overall execution time. Table 4.5 shows the performance characteristics of the various optimizations.

The cost model recommended L which performed 4% worse than a fixed choice of L+R and 9% worse than BEST. The model did not account for a “side effect” of remapping, whereby for input sizes that are a power-of-2, remapping eliminates a significant number of conflict misses. Since our cost model does not account for cache conflict effects, it underestimates the potential benefits that remapping can achieve in this case. A more sophisticated cost model that employs cache miss equations [13] or a similar mechanism might be able to handle this case more effectively.
Table 4.5. Performance Characteristics of the Optimizations for VPENTA

<table>
<thead>
<tr>
<th>VPENTA</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>382</td>
<td>13</td>
<td>19</td>
<td>7 2-D</td>
<td>92.0</td>
<td>3.3</td>
<td>4.8</td>
<td>99.2</td>
<td>1.00</td>
</tr>
<tr>
<td>copy</td>
<td>427</td>
<td>15</td>
<td>23</td>
<td>14 2-D</td>
<td>88.0</td>
<td>5.0</td>
<td>7.0</td>
<td>98.6</td>
<td>0.60</td>
</tr>
<tr>
<td>remap</td>
<td>698</td>
<td>13</td>
<td>19</td>
<td>8 1-D</td>
<td>89.7</td>
<td>8.9</td>
<td>1.4</td>
<td>99.8</td>
<td>1.11</td>
</tr>
<tr>
<td>pureloop</td>
<td>409</td>
<td>13</td>
<td>19</td>
<td>7 2-D</td>
<td>92.5</td>
<td>5.5</td>
<td>2.0</td>
<td>99.9</td>
<td>1.47</td>
</tr>
<tr>
<td>loop + remap</td>
<td>631</td>
<td>13</td>
<td>19</td>
<td>8 1-D</td>
<td>92.7</td>
<td>6.1</td>
<td>1.2</td>
<td>99.9</td>
<td>1.61</td>
</tr>
</tbody>
</table>

4.2.4 BTRIX

BTRIX solves a block tridiagonal matrix equation along one dimension of a four-dimensional array. The innermost loop is written to be vectorizable and involves strided accesses across four four-dimensional arrays. Figure 4.3 shows a portion of the code from the original program. We present the complete code and show the memory cost calculation for this program in the Appendix. Each of the four four-dimensional arrays has the innermost loop index, \( l \), in the first dimension, which results in huge strides in every access. We optimize this program by using loop fusion and permutation. First, we move the innermost loop \( l \) to the outermost loop position and fuse all the sub-nests in the first loop nest, to get an order \( j, l, m, n \) for subnest 1.1 and \( j, l, n \) for subloop nest 1.4. We then interchange the outermost loops \( j \) and \( l \) to bring arrays \( a, b \) and \( c \) into array order in the first loop nest. Then we optimize loop nest 2 in two different ways by using permutation. This represents the loop transformations (L and L').

However, loop transformations alone cannot bring all four arrays \( a, b, c, \) and \( s \)
int s[6][W][W][W], a[W][W][6][6], b[W][W][6][6], c[W][W][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[l][k][j-1]
      - a[l][j][2][m] * s[l][k][j-1]
      - a[l][j][3][m] * s[l][k][j-1]
      - a[l][j][4][m] * s[l][k][j-1]
      - a[l][j][5][m] * s[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[l][k][j+1]
      - b[l][j][2][m] * s[l][k][j+1]
      - b[l][j][3][m] * s[l][k][j+1]
      - b[l][j][4][m] * s[l][k][j+1]
      - b[l][j][5][m] * s[l][k][j+1]
    }
  ...

Figure 4.3. BTRIX kernel
into array order since the access pattern of array \( s \) conflicts with that of the others. As a result, array \( s \) still does not have unit stride. So, we further optimize this loop nest using copying- and remapping-based array restructuring to get four more choices (\( L+R, L+R', L+C, L+C' \)). It turns out that none of these four choices do better than either \( L \) or \( L' \) alone because of the high overhead in restructuring array \( s \). Table 4.6 shows the characteristics of the different choices.

Our cost model correctly calculated the costs and benefits of the various choices. It predicted that \( L' \) would be the best performing optimization, and indeed in eight of nine experiments we performed, \( L' \) was the best. The single misprediction occurred when the array dimensions were a power-of-2, which induced many conflict misses in \( L' \). For this case, \( L+C' \) was the best because all its arrays have unit stride resulting in fewer conflict misses. Since we do not model conflict misses, the cost model erroneously chose \( L' \) over \( L+C' \) incurring a penalty of 48%. The

<table>
<thead>
<tr>
<th>Optimizations</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>184</td>
<td>10</td>
<td>24</td>
<td>25 1-D 4 4-D</td>
<td>86.7</td>
<td>11.2</td>
<td>2.0</td>
<td>96.6</td>
<td>1.00</td>
</tr>
<tr>
<td>copy</td>
<td>218</td>
<td>14</td>
<td>40</td>
<td>25 1-D 7 4-D</td>
<td>95.7</td>
<td>3.1</td>
<td>1.2</td>
<td>99.9</td>
<td>1.49</td>
</tr>
<tr>
<td>pureloop</td>
<td>173</td>
<td>9</td>
<td>23</td>
<td>25 1-D 4 4-D</td>
<td>95.2</td>
<td>3.6</td>
<td>1.2</td>
<td>98.4</td>
<td>1.36</td>
</tr>
<tr>
<td>pureloop'</td>
<td>173</td>
<td>9</td>
<td>19</td>
<td>25 1-D 4 4-D</td>
<td>96.8</td>
<td>2.4</td>
<td>0.8</td>
<td>99.9</td>
<td>1.72</td>
</tr>
<tr>
<td>loop + copy</td>
<td>188</td>
<td>10</td>
<td>27</td>
<td>25 1-D 5 4-D</td>
<td>96.5</td>
<td>3.2</td>
<td>1.3</td>
<td>99.9</td>
<td>1.43</td>
</tr>
<tr>
<td>loop + copy'</td>
<td>188</td>
<td>10</td>
<td>27</td>
<td>25 1-D 5 4-D</td>
<td>96.5</td>
<td>2.8</td>
<td>0.7</td>
<td>99.9</td>
<td>1.67</td>
</tr>
<tr>
<td>loop + remap</td>
<td>246</td>
<td>9</td>
<td>24</td>
<td>26 1-D 4 4-D</td>
<td>96.2</td>
<td>2.7</td>
<td>1.1</td>
<td>98.6</td>
<td>1.24</td>
</tr>
<tr>
<td>loop + remap'</td>
<td>246</td>
<td>9</td>
<td>23</td>
<td>26 1-D 4 4-D</td>
<td>97.3</td>
<td>1.9</td>
<td>0.8</td>
<td>99.9</td>
<td>1.47</td>
</tr>
</tbody>
</table>
overall speedup obtained by the cost model’s recommendations is 1.72 while the
best possible overall speedup is 1.80 (i.e., cost model is 95.6% of \textbf{BEST}).

Also, our cost model correctly predicted that copying-based array restructuring
(C) is beneficial. In contrast, Leung's heuristics-based decision model recommended
that copying-based array restructuring not be done [19]. We did not use pure
remapping-based array restructuring (R) for this benchmark, because doing so
requires more than eight shadow descriptors. Loop transformations, however,
enabled remapping-based array restructuring by reducing the number of arrays
that needed to be optimized by restructuring.

\subsection{CFFT2D}

CFFT2D implements two-dimensional FFT. It consists of two subroutines that
perform FFTs along the first and second dimension of the array, respectively,
taking advantage of the parallel structure of the algorithm. Previous work on
loop transformations did not optimize this loop nest or reported no performance
improvement [23]. Dependence constraints and imperfect loop nests make loop
transformations infeasible, but do not prevent copying- or remapping-based array
restructuring. Both copying-based and remapping-based array restructuring are
effective for this benchmark. Table 4.7 shows the performance characteristics of
these optimization choices.

\begin{table}
\caption{Performance Characteristics of the Optimizations for CFFT2D}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{CFFT2D} & \textbf{# of} & \textbf{# of} & \textbf{# of} & \textbf{# of} & \textbf{L1} & \textbf{L2} & \textbf{Miss} & \textbf{TLB} & \textbf{Speedup} \\
\textbf{} & \textbf{Lines} & \textbf{Nests} & \textbf{Loops} & \textbf{Arrays} & \textbf{Hit} & \textbf{Hit} & \textbf{Rate} & \textbf{Hit} & \textbf{} \\
\hline
\textbf{original} & 212 & 11 & 23 & 1 2-D, 3 1-D & 88.0 & 5.9 & 6.1 & 98.7 & 1.00 \\
\textbf{copy} & 239 & 13 & 27 & 2 2-D, 3 1-D & 91.7 & 6.3 & 1.9 & 98.9 & 2.77 \\
\textbf{remap} & 238 & 13 & 23 & 2 2-D, 3 1-D & 94.1 & 4.4 & 1.5 & 99.9 & 2.90 \\
\hline
\end{tabular}
\end{table}
Like $syk\text{r2}k$, the relative performance of copying-based restructuring vs. remapping-based restructuring is input size dependent for $cfft2d$. The array sizes are $O(N^2)$, but each element is touched $O(N^2 \log N)$ times. Thus, copying-based restructuring outperforms remapping-based restructuring for large data sets where the high one-time cost of setup ($O(N^2)$) is amortized by the lower cost per access. Conversely, remapping is preferable for small data sets. Our cost model correctly predicted this tradeoff for most cases and obtained a speedup very close to the best possible speedup.

### 4.2.6 EX1

Figure 4.4 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 out of the six possible loop permutations are legal. $^1$ Loop permutation by itself yields almost no benefit, but it enables data transformations that are beneficial. Table 4.2 shows that on average $L + C$ performs worse than the original code. However, $L + C$ outperformed the base version for the four large data sets.

The main cause for poor performance in this kernel for large data set sizes is low TLB hit 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 considered five optimization choices for this kernel, $C, R, L, L+R, L+C$. The characteristics of the different choices are shown in Table 4.8.

Eight experiments with different values of $N$ (ranging from $N=200$ to $N=500$) were performed. Copying-based array restructuring obtains the best speedup in

---

$^1$In our paper [7], we mention that only two of six possible loop permutations are legal which is an error.
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 4.4. EX1 kernel

Table 4.8. Performance Characteristics of the Optimizations for EX1

<table>
<thead>
<tr>
<th>EX1</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>51</td>
<td>3</td>
<td>7</td>
<td>2 2-D</td>
<td>90.5</td>
<td>8.0</td>
<td>1.5</td>
<td>93.5</td>
<td>1.00</td>
</tr>
<tr>
<td>copy</td>
<td>58</td>
<td>4</td>
<td>9</td>
<td>3 2-D</td>
<td>91.3</td>
<td>7.1</td>
<td>1.6</td>
<td>99.7</td>
<td>1.53</td>
</tr>
<tr>
<td>remap remap</td>
<td>75</td>
<td>3</td>
<td>7</td>
<td>2 2-D 1 1-D</td>
<td>98.0</td>
<td>0.5</td>
<td>1.5</td>
<td>99.9</td>
<td>0.74</td>
</tr>
<tr>
<td>pureloop</td>
<td>51</td>
<td>3</td>
<td>7</td>
<td>2 2-D</td>
<td>82.8</td>
<td>13.8</td>
<td>3.4</td>
<td>94.6</td>
<td>0.66</td>
</tr>
<tr>
<td>loop+ copy</td>
<td>62</td>
<td>5</td>
<td>11</td>
<td>3 2-D</td>
<td>78.1</td>
<td>16.8</td>
<td>5.1</td>
<td>99.9</td>
<td>0.76</td>
</tr>
<tr>
<td>loop+ remap</td>
<td>71</td>
<td>3</td>
<td>7</td>
<td>2 2-D 1 1-D</td>
<td>85.4</td>
<td>11.0</td>
<td>3.6</td>
<td>99.9</td>
<td>0.43</td>
</tr>
</tbody>
</table>

each of these cases. Remapping suffers from high latencies while gathering the cache lines, and does not perform well. The cost model successfully predicted the best optimization in all cases.

4.2.7 IR KERNEL

The optimization choices for ir_kernel, shown in Figure 2.4 are C, R, L, L+R and L+C. Table 4.9 shows the characteristics of each optimization candidate. As predicted in our discussion in Section 3.2, a combination of loop transformation and data restructuring results in the best performance for ir_kernel. L+C outperforms C, and L+R outperforms R in all cases. Between L+R and L+C, the former performed better, but the cost model predicted otherwise. The main reason for this misprediction is the high incidence of conflict misses in L+C compared to L+R.
Table 4.9. Performance Characteristics of the Optimizations for IR_KERNEL

<table>
<thead>
<tr>
<th>IR_KERNEL</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>111</td>
<td>2</td>
<td>4</td>
<td>2 1-D</td>
<td>96.4</td>
<td>3.1</td>
<td>0.5</td>
<td>99.9</td>
<td>1.00</td>
</tr>
<tr>
<td>copy</td>
<td>120</td>
<td>4</td>
<td>8</td>
<td>4 2-D</td>
<td>97.6</td>
<td>2.0</td>
<td>0.4</td>
<td>99.9</td>
<td>1.07</td>
</tr>
<tr>
<td>remap</td>
<td>154</td>
<td>2</td>
<td>4</td>
<td>2 2-D</td>
<td>98.9</td>
<td>0.5</td>
<td>0.6</td>
<td>99.9</td>
<td>1.26</td>
</tr>
<tr>
<td>pureloop</td>
<td>111</td>
<td>2</td>
<td>4</td>
<td>2 1-D</td>
<td>94.7</td>
<td>4.8</td>
<td>0.5</td>
<td>99.9</td>
<td>0.96</td>
</tr>
<tr>
<td>loop+copy</td>
<td>117</td>
<td>3</td>
<td>6</td>
<td>2 1-D</td>
<td>97.6</td>
<td>2.0</td>
<td>0.4</td>
<td>99.9</td>
<td>1.07</td>
</tr>
<tr>
<td>loop+remap</td>
<td>135</td>
<td>2</td>
<td>4</td>
<td>3 1-D</td>
<td>99.0</td>
<td>0.5</td>
<td>0.6</td>
<td>99.9</td>
<td>1.26</td>
</tr>
</tbody>
</table>

The penalty for this incorrect prediction was high and causes the cost model’s recommendations to be only 85% of BEST.

4.2.8 KERNEL6

Kernel6 is a general linear recurrence equation solver that comes from the Lawrence Livermore National Laboratories benchmark suite of loop kernels. The recurrence relation in this loop makes loop permutation illegal (see Figure 4.5).

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 4.5. Kernel6 code
Table 4.10. Performance Characteristics of the Optimizations for KERNEL6.

<table>
<thead>
<tr>
<th>KERNEL6</th>
<th># of Lines</th>
<th># of Nests</th>
<th># of Loops</th>
<th># of Arrays</th>
<th>L1 Hit</th>
<th>L2 Hit</th>
<th>Miss Rate</th>
<th>TLB Hit</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>43</td>
<td>2</td>
<td>5</td>
<td>1 2-D, 1 1-D</td>
<td>87.4</td>
<td>8.8</td>
<td>3.8</td>
<td>95.6</td>
<td>1.00</td>
</tr>
<tr>
<td>copy</td>
<td>49</td>
<td>3</td>
<td>7</td>
<td>2 2-D, 1 1-D</td>
<td>90.9</td>
<td>6.2</td>
<td>2.9</td>
<td>98.3</td>
<td>1.24</td>
</tr>
<tr>
<td>remap</td>
<td>60</td>
<td>2</td>
<td>5</td>
<td>2 2-D, 1 1-D</td>
<td>95.1</td>
<td>3.7</td>
<td>1.2</td>
<td>99.9</td>
<td>1.92</td>
</tr>
</tbody>
</table>

We used remapping-based and copying-based array restructuring to optimize this kernel. Table 4.10 shows the characteristics of the different optimization candidates. The choice of whether R or C is better depends on the amount of reuse in the loop nest. We ran 22 experiments with various loop and data set sizes. Our cost model accurately chose the best optimization in 21 cases, and even in the single mispredicted case the performance was very close. When we used the cost model to select the optimizations to perform, we achieved an overall geometric mean speedup of 1.99, whereas the best overall mean speedup was 2.00. In contrast, applying remapping (R) or copying (C) exclusively results in an average speedup of only 1.92 and 1.24, respectively.

### 4.3 Summary

No single optimization is best for all the benchmarks. If we try to determine which optimization was the most beneficial for the maximum number of benchmarks, then there is a four-way tie between C, R, L and L+R (syr2k and ext vote for C, cfft2d and kernel6 for R, matmult and btrix for L, and vpenta and ir_kernel for L+R). Thus, it makes sense to choose optimizations based on the benchmark rather than apply a fixed set of optimizations to all benchmarks.

Combining loop and array restructuring can result in better performance than either individual optimization. For example, in vpenta, loop permutation optimizes the two most expensive loop nests, but the five remaining loop nests still have
strided accesses that loop transformations alone could not 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.

However, combining restructuring optimizations is not always a win even when their goals are not mutually conflicting. 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. But this additional optimization degrades performance because the overhead of array restructuring is higher than its benefits. The cost model correctly recognized this and chose pure loop restructuring as its best choice.

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

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 the given choices. The prediction success rate for the cost model across all benchmarks on average is 72%. This is quite good considering the fact that if the choices had been made completely at random, the prediction success rate would have been 22%. Though this metric is fairly indicative of the accuracy of the cost model, it is somewhat conservative. It does not distinguish between a close second and dead last. For example, in vpenta, though the cost model’s choice of L was incorrect 89% of
the time, \( L \) was slower than the best choice of \( L+R \) by only 5\%. In contrast, the choice of \( C \) would have been slower by 58\%.

- **quality of ranking.** This metric quantifies the accuracy of the relative ranking of the optimization choices. It is more general than the best prediction hit rate because it considers all the choices. We now define this metric mathematically. If we have \( N \) optimization choices \( opt_1, \ldots, opt_N \) then 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 \( N \), both inclusive. We define *quality of ranking* of the cost model as the arithmetic mean of all these probabilities.  

Notice that the cost model’s quality of ranking across all benchmarks is 77\%. In particular, for *vpenta*, the cost model’s quality of ranking is 89\% reflecting the fact that the cost model accurately ranked all the optimizations correctly except the first and second (\( L+R \) and \( L \) respectively, which were very close).

- **worst possible performance:** This metric tells us what the performance would be if the worst choice of optimization was made for each experiment. It gives us a lower bound for the worst possible performance for the cost model. For example, in *syr2k* the slowdown would have been by a factor of 10 if the worst choice of optimization was made for each experiment. For the benchmarks studied, the worst possible performance that can be achieved is 32.9\% of best. That the cost model is much closer to 100\% than 32.9\% shows the effectiveness of the cost model.

---

2For example, consider a case where there are four possible optimizations choices \( opt_1, opt_2, opt_3 \) and \( opt_4 \) for an application, and that three experiments have been run for this application. Suppose the ranking of the estimated- and observed-cost’s for the three experiments are as follows: first, estimated: \( opt_1, opt_3, opt_4, opt_2 \), and observed: \( opt_4, opt_3, opt_1, opt_2 \); second, estimated: \( opt_2, opt_3, opt_4, opt_1 \), and observed: \( opt_2, opt_1, opt_3, opt_4 \); third, estimated: \( opt_1, opt_3, opt_4, opt_2 \), and observed: \( opt_1, opt_3, opt_4, opt_2 \). Then, \( P_{12} = 1, P_{13} = 1/3, P_{14} = 1/3, P_{33} = 1, P_{24} = 1, \) 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.
• *actual performance*: This metric measures how close the cost model’s choice of optimizations is to that of **BEST**. This is the primary metric we use to describe 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 calculated to be 94.8%. This indicates that the cost model’s predictions, if not the **BEST**, come close to it for all the benchmarks.

Table 4.11 shows the accuracy of the cost model using the quantitative metrics discussed above for each benchmark. We find that quantifying the overheads and accesses 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. 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

<table>
<thead>
<tr>
<th>Application</th>
<th>Best Prediction Hit Rate</th>
<th>Quality of Ranking</th>
<th>Number of Experiments</th>
<th>Worst Possible Performance</th>
<th>Actual Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>matmul</em></td>
<td>0.64</td>
<td>0.63</td>
<td>11</td>
<td>25.0%</td>
<td>91.8%</td>
</tr>
<tr>
<td><em>syr2k</em></td>
<td>0.92</td>
<td>0.72</td>
<td>12</td>
<td>10.4%</td>
<td>94.0%</td>
</tr>
<tr>
<td><em>vpenta</em></td>
<td>0.11</td>
<td>0.89</td>
<td>9</td>
<td>41.4%</td>
<td>93.3%</td>
</tr>
<tr>
<td><em>btrix</em></td>
<td>0.67</td>
<td>0.81</td>
<td>9</td>
<td>51.7%</td>
<td>95.6%</td>
</tr>
<tr>
<td><em>cfft2d</em></td>
<td>0.80</td>
<td>0.64</td>
<td>10</td>
<td>34.4%</td>
<td>100%</td>
</tr>
<tr>
<td><em>ex1</em></td>
<td>1.00</td>
<td>0.79</td>
<td>8</td>
<td>27.7%</td>
<td>100%</td>
</tr>
<tr>
<td><em>ir_kernel</em></td>
<td>0.67</td>
<td>0.71</td>
<td>9</td>
<td>60.9%</td>
<td>84.9%</td>
</tr>
<tr>
<td><em>kernel6</em></td>
<td>0.95</td>
<td>0.94</td>
<td>22</td>
<td>42.5%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>0.72</td>
<td>0.77</td>
<td><strong>22</strong></td>
<td><strong>32.9%</strong></td>
<td><strong>94.8%</strong></td>
</tr>
</tbody>
</table>
driving restructuring optimizations.

4.4 Caveats

Our cost model has a number of known inaccuracies; improving its accuracy is an area for future work. First, we do not consider the impact of any optimization on TLB performance. For some input sizes, TLB effects can dwarf cache effects, so adding a TLB model 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 system have less impact on program performance than spread out cache misses, but our model gives them equal weight. Third, we do not consider cache reuse (i.e., we estimated $f$ to be zero) or cache interference. We assume that the costs of different loop nests are independent, and thus additive. If there is significant inter-nest reuse, our model will overestimate the memory costs and recommend an unnecessary optimization. We do not have a framework for calculating $f$, and would benefit from a framework such as cache miss equations proposed by Ghosh et al. [13]. Similarly, not modeling cache interference could result in the model underestimating the memory costs if there are significant conflict misses in the optimized applications.
CHAPTER 5

RELATED WORK

This chapter explores program restructuring techniques for improving locality of arrays in loop nests and on analytical models for program and memory performance.

5.1 Loop Transformations

McKinley, Carr and Tseng present a loop transformation framework for data locality optimization that guides loop permutation, fusion, distribution, and reversal [23]. 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 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 [33]. 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 this model can 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 chapters, loop transformations
alone are not sufficient for achieving good data locality in many cases.

5.2 Data Transformations

Leung and Zahorjan introduced array restructuring, a technique to improve the spatial locality of arrays in loop nests [19]. 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 nonzero 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. introduce scalar replacement [4], which works as follows. 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. Throughout this thesis, we assume that the optimization is done wherever possible. This assumption also simplifies the calculation of the cost model.
5.3 Combined Transformations

Cierniak and Li propose a linear algebraic framework for locality optimization by unifying data and control transformations [10]. 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. [14] 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 does 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.

5.4 Cost Models

Saavedra et al. 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 [30]. This 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. Not modeling cache and TLB misses caused their model to underestimate the running time of applications with high miss rates. In a later work, Saavedra and Smith [29] 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 exercise every level of the memory hierarchy. However, they do not model or estimate the cache hit rates of applications. Rather, they use 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. introduce Cache Miss Equations (CMEs), a precise analytical representation of cache misses in a loop nest [13]. 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. This would allow us to improve the accuracy of our cost model's predictions, and also to deal with small data sizes.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

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

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

We draw the following conclusions in this thesis.

- Copying-based array restructuring, though useful in improving locality of multi-dimensional array accesses, is not widely applicable because the overhead of copying can overwhelm the benefit derived from improved locality.
• Hardware support for remapping from a smart memory controller enables new data restructuring optimizations. In particular, we present remapping-based array restructuring, a new data restructuring optimization that does not incur the high overhead of copying. However, hardware remapping introduces other overheads, including the cost of translation and gathering on every access.

• 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, combined restructuring is useful, but must be done carefully.

• Restructuring can be harmful, so we need analytical guidelines to help predict when it will be useful. We present cost models to estimate the memory costs of the restructuring optimizations. We model the costs and benefits of hardware support for remapping to help decide where hardware support will be beneficial.

• We model the memory costs of applications as a whole in an integrated analytical framework, which allows us to compare different restructuring optimizations and their combinations, with or without hardware support, in the same framework. By being able to pick from a mix of individual and combined optimizations, our cost model selects the combination of optimizations closest to optimal performance of the options we considered.

6.1 Future Directions

The rest of this chapter contains recommendations for how our integrated analytical framework can be improved.

The most significant improvement to our analytical cost model would be the addition of a mechanism to estimate cache miss rates for each reference in a
loop nest. Cache Miss Equations (CMEs) represent a significant step in this direction [13]. Adding CMEs to our framework would allow us to use the cost model for small data set sizes and handle cases where there is cache interference. Small data sets cause our cost model to overestimate the potential benefits of restructuring, while the latter results in an underestimation of potential benefits. These problems were responsible for occasional mispredictions in our experiments. Adding CMEs would also allow us to extend the cost model to handle blocking optimizations, including loop tiling.

Integration of restructuring optimizations could have been taken further. The lack of automation of the cost model set an upper limit on the number of combinations that we could consider. The set of optimization candidates that were considered in this thesis were loop transformations (L), remapping-based array restructuring (R), copying-based restructuring (C), loop plus copying-based restructuring (L+C) and loop plus remapping-based restructuring (L+R). We did not consider combining remapping and copying in the same program because doing so would have considerably increased the number of optimization candidates. For example, even for a loop nest with just four arrays, the number of C+R combinations is 81 (i.e., $3^4$) since there are three choices – none, copying-based and remapping-based restructuring – for each array. Also, it would be interesting to study how to integrate static and dynamic array restructuring.

Finally, the cost model can also 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 would also involve extending the cost model to handle more types of remapping, e.g., the indirection vector remapping. Our analytical model framework is designed for regular memory accesses and cannot be linearly extended
to handle irregular memory access patterns that are inherent in indirection vector remapping. Developing memory cost models for irregular memory accesses is also an open problem.
APPENDIX

EXAMPLE COST MODEL

CALCULATION

This appendix contains the complete source code of the BTRIX benchmark and the \texttt{L+R} optimized version of BTRIX. The cost models for both these versions are developed completely, and they are compared both analytically and through actual simulation measurements.

BTRIX Original Version: Source Code

(converted to C from Fortran)

```c
#include <stdio.h>

#define N 100
#define md N
#define jd N
#define kd N
#define ld N

long t0,t1,t2,t3,t4;
long loop_t, loop1_t, loop2_t, loop3_t, loop4_t, loop5_t, loop6_t, loop7_t;
double r_1,t;
int s[6][ld][kd][jd], a[md][md][6][6], b[md][md][6][6], c[md][md][6][6];
int u12[md], u13[md], u14[md], u15[md], u23[md], u24[md], u25[md], u34[md];
int u35[md], u45[md];
double l11[md], l21[md], l31[md], l41[md], l51[md], l61[md], l71[md], l81[md], l91[md];
double l12[md], l13[md], l14[md], l15[md], l16[md], l17[md], l18[md], l19[md], l20[md];

void btrix(int js, int je, int ls, int le, int k)
{
    for(j = js; j<je; j++)
    {
        if(j!=js)
```

{
    loop_t = sysclocks(); /* clock cycles timer in the RSIM simulator*/
    for(m = 1; m<=5; m++)
        for(n = 1; n<=5; n++)
            for(l=ls; l<le; l++)
                b[1][j][n][m] = b[1][j][n][m]
                    - a[1][j][1][m] * b[1][j-1][n][1]
                    - a[1][j][2][m] * b[1][j-1][n][2]
                    - a[1][j][3][m] * b[1][j-1][n][3]
                    - a[1][j][4][m] * b[1][j-1][n][4]
                    - a[1][j][5][m] * b[1][j-1][n][5];
        loop1_t += sysclocks() - loop_t;
}

/******STEP 2. COMPUTE L INVERSE************************/  

loop_t = sysclocks();
for(l=ls; l<le; l++)
{
    111[1] = 1.0 / b[1][j][1][1];
    u12[1] = b[1][j][2][1] * 111[1];
    u12[3] = b[1][j][3][1] * 111[1];
    u12[4] = b[1][j][4][1] * 111[1];
    u12[5] = b[1][j][5][1] * 111[1];
    121[1] = b[1][j][1][2];
    122[1] = 1.0 / (b[1][j][2][2] - 121[1] * u12[1]);
    u23[1] = (b[1][j][3][2] - 121[1] * u13[1]) * 122[1];
    u24[1] = (b[1][j][4][2] - 121[1] * u14[1]) * 122[1];
    u25[1] = (b[1][j][5][2] - 121[1] * u15[1]) * 122[1];
    131[1] = b[1][j][1][3];
}

loop2_t += sysclocks() - loop_t;

loop_t = sysclocks();
for(l=ls; l<le; l++)
{
    141[1] = b[1][j][1][4];
                        143[1] * u34[1]);
                        u35[1]) * 144[1];
    151[1] = b[1][j][1][5];
                        u34[1];
}
}
loop3_t += sysclocks() - loop_t;

loop_t = sysclocks();
if(j!=js)
{
    for(m = 1; m<=5; m++)
    for(l=ls; l<le; l++)
    {
        s[m][l][k][j] = s[m][l][k][j]
        - a[1][l][k][j] * s[1][l][k][j] - 1
        - a[1][l][k][j] * s[1][l][k][j] - 1
        - a[1][l][k][j] * s[2][l][k][j] - 1
        - a[1][l][k][j] * s[3][l][k][j] - 1
        - a[1][l][k][j] * s[4][l][k][j] - 1
        - a[1][l][k][j] * s[5][l][k][j] - 1;
    }
}
loop4_t += sysclocks() - loop_t;

loop_t = sysclocks();
for(l=ls; l<le; l++)
{
    FWD SUBSTITUTION */
    d1 = s[1][l][k][j] * 111[1];
    d2 = (s[2][l][k][j] - 121[1] * d1) * 122[1];
        d4) * 155[1];
    s[5][l][k][j] = d5;
    s[4][l][k][j] = d4 - u45[1] * d5;
        u25[1] * d5;
} 
loop5_t += sysclocks() - loop_t;

if(j!=je)
{
    loop_t = sysclocks();
    for(n = 1; n<=5; n++)
    for(l=ls; l<le; l++)
    {
        int c1 = c[1][l][n][1] * 111[1];
        int c2 = (c[1][l][n][2] - 121[1] * c1) * 122[1];
        int c3 = (c[1][l][n][3] - 131[1] * c1 - 132[1] * c2) * 133[1];
            144[1];
    }
b[1][j][n][5] = c5;
b[1][j][n][4] = c4 - u45[1] * c5;
}

loop6_t += sysclocks() - loop_t;
}

loop_t = sysclocks();
count=0;
je = je - 2;
for(j=je; j>js; j--)
  for(m = 1; m<=5; m++)
    for(l=1s; l<1e; l++)
      {
        s[m][l][k][j] = s[m][l][k][j]
        - b[1][j][k][m] * s[1][l][k][j + 1]
        - b[1][j][k][m] * s[2][l][k][j + 1]
        - b[1][j][k][m] * s[3][l][k][j + 1]
        - b[1][j][k][m] * s[4][l][k][j + 1]
        - b[1][j][k][m] * s[5][l][k][j + 1];
        count = (count+s[m][l][k][j])%1024;
      }

loop7_t += sysclocks() - loop_t;
printf("count = %ld\n",count);
return;
}

int main()
{
  t0 = sysclocks();
  /* BTRIX TEST PROGRAM */
  /* INITIALIZATION */
  nb = 25*md*md;
  ns = jd*kd*1d*5;
  t = 7.2759576141834259e=5;
  for (l = 1s; l < md; ++l)
    for (k = 1; k < md; ++k)
      for (j = 1; j <= 5; ++j)
        for (i_ = 1; i_ <= 5; ++i_)
          {
            t = ((int)(t * (float)78125.))%65536;
            a[1][k][j][i_] = t;
            t = ((int)(t * (float)78125.))%65536;
            b[1][k][j][i_] = t;
            t = ((int)(t * (float)78125.))%65536;
c[l][j][i_] = t;
/* L100: */
}
for (l = 1; l <= 5; ++l)
    for (k = 0; k < ld; ++k)
        for (j = 0; j < kd; ++j)
            for (i_ = 0; i_ < jd; ++i_)
            {
                t = ((int)(t * (float)78125.))%65536;
                s[l][k][j][i_] = t;
            /* L110: */
            }
        }
    }
    }
}

t1 = t2 = sysclocks();
i_ = it;
for (ii = 1; ii <= i_; ++ii)
    for (k = 0; k < kd; ++k)
        btrix(js, je, ls, le, k);
t3 = t4 = sysclocks();
printf("original: N= %d loop1_t= %ld loop2_t= %ld loop3_t= %ld loop4_t=\n %ld loop5_t= %ld loop6_t= %ld loop7_t= %ld\n", N, loop1_t,
        loop2_t, loop3_t, loop4_t, loop5_t, loop6_t, loop7_t);
printf("original: N= %d initial= %ld setup= %ld actual_code= %ld\n cleanup=%ld total= %ld\n", N, t1-t0, t2-t1, t3-t2, t4-t3,t4-t0);
    return 0;
}

**Memory Cost Calculation for the BTRIX Original Code**

*Loop1*: The memory cost of reference $b[l][j][n][m]$, i.e. $\text{MemoryCost}(b[l][j][n][m])$

\[
\frac{(ge-js+1) \times 5 \times (le-is+1)}{\text{max}(25, 1)} \times (1 - f) \times G_c,
\]

where the cache line size, $c_{ls} = 32$ elements (L2 cache line is 128 bytes and $b$ is an array of ints) and the stride is $6 \times 6 \times md$. We make the simplifying assumption that cache reuse, $f$, is low if the working set of this application is larger than the L2 cache and therefore neglect the term $(1 - f)$. This cost equation simplifies to: $\text{MemoryCost}(b[l][j][n][m]) = 25 \times N^2 \times G_c$. The reference $b[l][j - 1][n][1]$ is accessed far apart from the reference $b[l][j][n][m]$ and thus is considered independent as it lies in a different cache line. The memory cost of this reference $b[l][j - 1][n][1]$ is $25 \times N^2 \times G_c$. References $b[l][j - 1][n][1]$, $b[l][j - 1][n][2]$, $b[l][j - 1][n][3]$, $b[l][j - 1][n][4]$ and $b[l][j - 1][n][5]$ lie on the same cache line. Thus, the memory cost of all these five references is just the memory
cost of gathering one reference $b[l][j-1][n][1]$. Similarly, the memory cost of the five dependent references of array $a$ is $25 \times N^2 \times G_c$. The memory cost of $Loop_1$ is the sum of the memory costs of the individual independent references, and is $75 \times N^2 \times G_c$.

$Loop_2$: In this loop, 15 independent references (l11, l21, l22, l31, l32, l33, u12, u13, u14, u15, u23, u24, u25, u34, u35) single-dimensional arrays of type double have stride-one accesses. Their memory cost is $(le-ls+1) \times (je-js+1) \times 15 \times G_c$. Since the elements have type `double` the cache line size used for these references is 16. The references to the $b$ array all lie in the same cache line (or possibly two), and their cumulative cost is just one cache line gather per iteration, i.e., memory cost is $(le-ls+1) \times (je-js+1) \times G_c$. Thus the total cost of this loop is $31 \times N^2 \times G_c$.

The calculation for the rest of the loops is similar, and we summarize them in Table A.1.

$Loop_3$:

Thus the memory cost of the `btrix` subroutine is the sum of all the memory costs of the individual loop nests, and is $157.625 \times N^2 \times G_c$. The `main` routine has three loop nests: the first loop nest initialize the arrays $a$, $b$ and $c$ and has a memory cost of $75 \times N^2 \times G_c$; the second loop nest initializes array $s$ and has a memory cost of $5 \times N^3 \times G_c$; and the third loop nest has a memory cost of $N \times MemoryCost(btrix)$.

<table>
<thead>
<tr>
<th>Loop Nest</th>
<th>Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Loop_1$</td>
<td>$75 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_2$</td>
<td>$31 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_3$</td>
<td>$47 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_4$</td>
<td>$30 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_5$</td>
<td>$405 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_6$</td>
<td>$185 \times N^2 \times G_c$</td>
</tr>
<tr>
<td>$Loop_7$</td>
<td>$30 \times N^2 \times G_c$</td>
</tr>
</tbody>
</table>
Table A.2. Total Memory Costs for the Various Optimizations of BRiX

<table>
<thead>
<tr>
<th>Optimization Candidate</th>
<th>Total Memory Cost of Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O$</td>
<td>$(157.78N^3 + 2.34N^2)G_c$</td>
</tr>
<tr>
<td>$L$</td>
<td>$(43.90N^3 + 84N^2)G_c$</td>
</tr>
<tr>
<td>$C$</td>
<td>$(43.40N^3 + 111.375N^2)G_c$</td>
</tr>
<tr>
<td>$L + C$</td>
<td>$(23.62N^3 + 85N^2)G_c$</td>
</tr>
<tr>
<td>$L + R$</td>
<td>$(40.03N^3 + 84N^2)G_c + 107N^2 + 0.156N^3G_r(N^2)$</td>
</tr>
<tr>
<td>$L_f$</td>
<td>$(14.84N^3 + 84N^2)G_c$</td>
</tr>
<tr>
<td>$L + R_f$</td>
<td>$(36.75N^3 + 84N^2)G_c + 107N^3 + 0.156N^3G_r(N^3)$</td>
</tr>
</tbody>
</table>

The total memory cost of this program thus is the sum of the memory costs of these three loop nests in the *main* subroutine. It is $(157.625 \times N + \frac{5}{32} \times N + \frac{75}{32} \times N^2 \times G_c$.

Table A.2 shows the final memory costs as estimated by the cost model for each optimized version of *brix*.
REFERENCES


