Optimizing Memory Cost with Loop Transformations

Hailong Shi
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all
Part of the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/497

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
Optimizing Memory Cost With Loop Transformations

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

Hailong Shi
B.S., Jilin University, 2007

2011
Wright State University
I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Hailong Shi ENTITLED Optimizing Memory Cost With Loop Transformations BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Meilin Liu, Ph. D.
Thesis Director

Mateen Rizki, Ph.D., Chair
Department of Computer Science and Engineering

Committee on Final Examination

Meilin Liu, Ph. D.

Jack Jean, Ph. D.

Soon Chung, Ph. D.

Andrew Hsu, Ph.D.
Dean, School of Graduate Studies
ABSTRACT

Shi, Hailong. M.S. Department of Computer Science and Engineering, Wright State University, 2011. Optimizing Memory Cost With Loop Transformations

Embedded systems are usually constrained in terms of timing, power, and memory. Many embedded applications, especially in the multi-media and telecom domains, are inherently data dominant. These embedded DSP applications usually exhibit intensive computations in the form of multi-level loops. The performance of these embedded DSP applications mainly depends on the code quality of the loops and the memory hierarchy design. During the design phase of the embedded system, it is important to estimate the overall storage requirement to guide the memory system design and take advantage of the memory system by program transformations and loop transformations.

Loop transformations including loop permutation, loop fusion and loop tiling are important program transformation techniques utilized to improve the data reuse, thus reduce the life time of data elements and memory cost. In this thesis, we propose a technique to estimate the memory cost of a nested loop. We then propose loop optimization strategies to combine various loop transformation techniques to reduce the overall memory cost. In addition to significantly reduce the running time from original loops to fused loops, the results of our experiment presented that our proposed loop optimization schemes can also reduce memory cost and cache misses.
# TABLE OF CONTENTS

## CHAPTER 1. INTRODUCTION ................................................. 1
1.1 Motivation .............................................................. 1
1.2 Thesis Outline ......................................................... 3

## CHAPTER 2. BASIC CONCEPTS ............................................. 4
2.1 Data Dependences ....................................................... 4
2.2 Data Flow Graph ....................................................... 5
2.3 Retiming ................................................................. 6
2.4 Multi-dimensional Data Flow Graph ................................. 7
2.5 Loop Dependence Graph ............................................... 8
2.6 Multi-dimensional Retiming .......................................... 9
2.7 Data Locality ............................................................ 11
2.8 Cache Organization ..................................................... 12

## CHAPTER 3. OPTIMIZING MEMORY COST WITH LOOP TRANSFORMA-
          TIONS ................................................................. 14
3.1 Memory-Size Estimation ................................................ 14
   3.1.1 Memory Model ..................................................... 15
   3.1.2 Code-Size Estimation ............................................ 15
   3.1.3 Memory-Size Estimation of a Loop ............................. 16
3.2 Loop Transformations for Memory Cost Reduction ............... 18
3.3 Legalizing Loop Fusion with Improved Data Locality .......... 21
   3.3.1 Possible Dimensions for Legalizing Loop Fusion ............ 21
   3.3.2 The Select_LF Technique ...................................... 23
3.4 Legalizing Loop Fusion Technique with Minimal Memory Cost 26
   3.4.1 Motivation Example ............................................. 27
   3.4.2 Algorithm of Legalizing Loop Fusion with Minimum Memory Cost 30
3.5 Simulation ................................................................. 31
  3.5.1 SimpleScalar Installation ........................................ 32
  3.5.2 Performance ......................................................... 34

CHAPTER 4. CONCLUSION ..................................................... 37

REFERENCES ............................................................... 40
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data Dependences</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>DFG with retiming</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>(a) Program from a digital circuit. (b) Data flow graph for program (a).</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>(a) A two-level loop. (b) The corresponding LDG</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>(a) The retimed MDFG. (b) The code of the retimed MDFG.</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>An example loop</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>An intermediate format statement and its machine code</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>An example loop</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Loop permutation is used to legalize loop fusion</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>(a) An example loop’s Loop Dependence Graph. (b) Retiming the 2nd dimension’s constraint graph. (c) The retimed loop dependence graph by retiming the 2nd dimension.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>A loop dependence graph and the retimed loop dependence graph.</td>
<td>28</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

3.1 Memory size estimation ................................................................. 18
3.2 The computation of $Prev_{Dim}$ and $Min_{Prev}$ of the LDG shown in Figure 3.4(a). ................................................................. 23
3.3 Execution time comparison. ......................................................... 34
3.4 Cache misses for different cache settings. ................................. 35
ACKNOWLEDGEMENTS

During my one and a half years’ study at Wright State University, My advisor gave me a lot of help and constructive suggestions for both studying and life. So I would like to thank my advisor, Dr. Meilin Liu so much, for her brilliant thoughts, advice, and inspirations.

I also would like to thank my thesis committee members: Drs. Jack Jean and Soon Chung for taking their invaluable time to serve on my committee and help in improving this document.

I also wish to thank my fellows at Wright State University for their help and support throughout my master study.

September 2011
CHAPTER 1
INTRODUCTION

1.1 Motivation

For embedded systems, there are some limitations in terms of timing performance, power consumption, and memory size. Many embedded applications are inherently computation intensive. The data intensive applications are usually modeled in the form of multi-level loops \([1, 6, 8]\). Because the performance of applications for embedded systems mainly depends on the code quality of the loops and the memory hierarchy design, it is important to apply loop transformations to reduce memory cost. Thus we can fully take advantage of the available limited hardware resources \([1]\).

The growing gap between processor and memory speeds is another challenge for an embedded system to achieve high performance \([1, 5, 17]\). Usually, the processor speed is much faster than the memory speed. So we need to reduce number of memory accesses to improve the system performance. Memory performance has a major effect on overall system performance, especially for the data-dominated applications \([1, 17]\), and memory also have a major influence on power consumption. The “memory wall” problem, i.e., the growing gap of speed between CPU and memory, becomes even more serious when throughput in the processor part is propelled by multi-core architectures. In this thesis, I propose to address the “memory wall” problem by estimating memory size, minimize the memory cost by various loop transformation techniques, and thus make a better use of the on-chip memory.
In the application programs, variables and data arrays need to be brought into the main memory before they are accessed. Each scalar variable and array element has its lifetime. Scalar variables and array elements with non-overlapping lifetimes can be mapped to the same physical memory locations, which is called “in-place mapping”. The memory size estimation based on the data dependency analysis and the liveness analysis of the data elements will be used to guide the memory configuration and partition. A reduction of the amount of memory can decrease the number of levels in the memory hierarchy. A significant reduction of the amount of the memory would ideally allow the storage of all variables in the on-chip memory, thus enabling the removal of off-chip memory access. Using the initial memory size estimation, program transformations can be applied to further reduce the memory size for data intensive applications.

Loop transformations including loop permutation and fusion are important program transformation techniques, and they can be used to increase the data reuse, correspondingly lifetime of data elements will be reduced. This lifetime reduction of data elements, could reduce the overall storage space.

Loop fusion is one of the most effective techniques to increase the performance of programs by combining and transforming multiple loops into one loop [1, 11, 17]. Loop fusion increases data locality by bringing data accesses closer both temporally and spatially. Thus, loop fusion can reduce memory cost. There are several memory research investigations using loop fusion to optimize application programs [1, 8, 10, 11, 17]. However, loop fusion is not always legal because of fusion-preventing dependencies among loops. In this thesis, we will try to apply loop permutation to enable loop fusion. Also, Loop shifting or retiming will be used to enable loop fusion when loop permutation cannot legalize loop fusion while considering the memory cost of the
transformed loop.

1.2 Thesis Outline

In this thesis, we first propose and investigate a scheme to compute the memory cost of a basic block and a loop nest based on the data dependency distance. Then, we propose to combine various loop transformation techniques to achieve the minimal memory cost. For a single loop nest, we try to apply loop permutation to reduce memory cost whenever it is legal. For a group of loops, we will try to apply loop permutation to enable loop fusion and then fuse the loops together, so that the code size and the memory cost will be minimal. If loop fusion is still not legal after the first two steps, we will apply retiming to legalize loop fusion, then fuse loops to reduce memory cost. In addition to significantly reduce the running time from original loops to fused loops, the simulation results of our experiment presented that our proposed loop optimization schemes can also reduce memory cost and cache misses.

The rest of this thesis is organized as follows: In Chapter 2, we introduce some basic concepts used in this thesis. In Chapter 3, we first propose a technique to estimate the memory cost of a nested loop. We then propose to combine various loop transformation techniques to reduce the memory cost of application programs, thus to improve timing performance of the application programs. In Chapter 4, we conclude the thesis.
CHAPTER 2
BASIC CONCEPTS

We introduce some basic concepts and principles in this chapter, such as data dependences, data flow graph (DFG), retiming, multi-dimensional retiming, data locality and cache architecture.

2.1 Data Dependences

Data dependence analysis is critical to source-to-source program transformations in compiler optimization [1, 2, 17]. A dependence exists between two statements in a program if there exists a control-flow path from the first statement to the second and both statements reference the same memory location. Dependences between two statements in a program commonly mentioned in the literature include input dependences, true dependences, anti-dependences and output dependences which are called name dependences [10].

There are four types of data dependences as shown in Figure 2.1. For two sequentially executed statements $S_1$ and $S_2$ which reference a common array element $B[i]$ in a nested loop $L$, a data dependence exists if $S_1$ references the same array element in some iteration $\vec{i}$ as $S_2$ in some iteration $\vec{i}'$, where $\vec{i}$ is executed before iteration $\vec{i}'$. The dependence is classified as a true dependence when $S_1$ writes $B[i]$ and $S_2$ reads $B[i]$; or as an anti-dependence when $S_1$ reads $B[i]$ and $S_2$ writes $B[i]$; or as an output dependence when both $S_1$ and $S_2$ write into $B[i]$. Note that true dependences,
ant-dependences and output dependences have constraints on the order in which statements execute correctly. A safe program transformation should keep all the true dependences, anti dependences, and output dependences the same as in the original program. But for input dependences, they do not need to be preserved for correctness.

Loops in a nest are numbered from the outermost to innermost. The outermost loop is defined as loop level one. The difference of the sink loop iteration and source loop iteration of a data dependence in a loop is called the distance of a data dependence. A dependence is defined to be loop carried if the distance of a data dependence is larger than vector $\vec{0}$. If the distance of a data dependence in a loop is $\vec{0}$, then the data dependence is called loop independent.

2.2 Data Flow Graph

The definition of data flow graph is defined in [21], and we repeat it here for clarity. A data flow graph $(DFG) G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where $V$ is the set of computation nodes, $E \subseteq V \times V$ is the set of edges
representing dependences, $\vec{d}$ is a function from $E$ to $Z^+$, representing the number of delays between two nodes, and $t : V \to Z^+$ is a mapping from node $V$ to a set of positive integers, representing the time of each node [21].”

![DFG with retiming](image)

**Figure 2.2. DFG with retiming**

Programs with one-level loops could be modeled by cyclic data flow graphs as shown in Figure 2.2. The execution of all nodes in $V$ exactly represents an iteration, i.e., the execution of one instance of the loop body. Iterations are numbered by an index $i$. The edges whose edge weight are non-negative are represented by edges with bar lines [21]. An edge whose edge weight is 2 is represented by an edge with two bar lines as shown in Figure 2.2.

### 2.3 Retiming

The retiming technique was proposed by Saxe and Leiserson [7], and was extended in [21]. Here we repeat it for clarity. “The retiming technique can be applied on a data flow graph to redistribute the edge weights in the graph. The edge weight are redistributed around in the graph in the following way: a delay is taken from each of
the incoming edges of node \( v \), and then added to each of the outgoing edges of \( v \), or vice versa. Note that the retiming technique preserves data dependences of the original DFG. The retiming function \( r : V \rightarrow Z \) is the number of delays moved through node \( v \in V \) [21].

After apply retiming on a given graph, we get a retimed graph. The retimed graph has the same nodes and edges as the original graph. But the weight of the edges in the retimed graph is different from the edge weight in the original graph.

### 2.4 Multi-dimensional Data Flow Graph

In this thesis, we utilize the generalization of the basic DFG model. The definition of Multi-dimensional data flow graph is defined in [9], we repeat here for clarity.

The multi-dimensional data flow graph (\( MDFG \)) is used to model a multi-level loop. “An MDFG \( G = (V, E, \vec{d}, t) \) is a node-weighted and edge-weighted directed graph, where each node \( V \) represents a statement, \( E \subseteq V \times V \) is the set of edges representing dependences, and each edge is labeled by an n-dimensional delays between two nodes, where \( t \) is a function from \( V \) to positive integers, representing the computation time of each node [9].”

A two-dimensional data flow graph \( G = (V, E, \vec{d}, t) \) is an \( MDFG \) with edge weight function \( d : w \rightarrow Z^2 \). We use \( \vec{d}(e) = (d_x, d_y) \) to denote a two-dimensional edge weight, and \( d_x \) represents the edge weights on \( x \)-dimension, and \( d_y \) represents the edge weights on \( y \)-dimension.

The program presented in Figure 2.3(a) is taken from a electronic circuit and its DFG is presented in Figure 2.3(b). There are five edges in this data flow graph, \( A \rightarrow B, A \rightarrow C, B \rightarrow D, C \rightarrow D \) and \( D \rightarrow A \).
For \( i = 2 \) to \( n \) do
  For \( j = 2 \) to \( n \) do
    S1: \( D(i,j) = B(i-2,j+2) \times C(i-2,j-2) \);
    S2: \( A(i,j) = D(i,j) \times 0.5 \);
    S3: \( B(i,j) = A(i,j) + 1 \);
    S4: \( C(i,j) = A(i,j) + 2 \);
  Endfor
Endfor

Figure 2.3. (a) Program from a digital circuit. (b) Data flow graph for program (a).

Iterations of a nested loop are indexed by a vector \( \vec{i} = (i_1, i_2, \ldots, i_n) \), where \( i_k \) is the value of the \( k^{th} \) loop index in the loop nest, ordering from the outermost to the innermost loop. In a sequential loop, the iterations are executed in lexicographic order of the index vectors. A \( N \)-dimensional vector \( \vec{v} \) is smaller than a \( N \)-dimensional vector \( \vec{u} \) according to the lexicographic order if and only if \( \vec{v}_1 = \vec{u}_1, \ldots, \vec{v}_{l-1} = \vec{u}_{l-1} \) and \( \vec{v}_l < \vec{u}_l \) for some integer \( l \leq N \) [9].

2.5 Loop Dependence Graph

For two sequentially executed two-level nested loops \( L1 \) and \( L2 \), with iteration \( (i_1, i_2) \in L2 \) and \( (j_1, j_2) \in L1 \), we say that there is a loop dependence vector \( \vec{d} = (i_1 - j_1, i_2 - j_2) \) that represents the data dependence between these two loops if a data value computed in \( (j_1, j_2) \in L1 \) is consumed in \( (i_1, i_2) \in L2 \) [16].

In order to present the data dependences between loops clearly, we use a loop dependence graph (LDG) to model a nested loop. The definition of a multi-dimensional loop dependence graph is defined in [9], we repeat it here for clarity.

“A multi-dimensional loop dependence graph (MLDG) \( G = (V, E, \delta, o) \) is a
Figure 2.4. (a) A two-level loop. (b) The corresponding LDG

In a loop dependence graph, an edge $e$ with edge weight $\delta(e) < (0, 0, ..., 0)$ represents a fusion-preventing dependence. The fusion-preventing dependence edges are $e_1$ and $e_2$ in the loop dependence graph as shown in Figure 2.4(b) for the loop shown in Figure 2.4(a).

2.6 Multi-dimensional Retiming

The multi-dimensional retiming technique was developed by Passos et. al. to optimize the multi-dimensional problems based on the retiming technique proposed by Saxe and Leiserson [7, 12, 13]. In order to get the minimum cycle time, we can use retiming to
redistribute edge weights. In this thesis, based on the technique of multi-dimensional retiming to redistribute the delays in the data flow graph (DFG) or the loop dependence graph (LDG), we develop the graph transformation techniques.

![Graph Transformation Technique Diagram](image)

**Figure 2.5.** (a) The retimed MDFG. (b) The code of the retimed MDFG.

The definition of a multi-dimensional retiming technique is defined in [13]. For clarity, we repeat the definition here. “For a multi-dimensional data flow graph (MDFG) or the multi-dimensional loop dependence graph (MLDG), a multi-dimensional retiming \( \vec{r} \) is a function from \( V \) to \( Z^n \). The retiming value \( \vec{r}(u) \) represents how many delays are added into the edges \( u \rightarrow v \) and subtracted from the edges \( w \rightarrow u \), for \( u, v, w \in V \). Therefore, in the retimed MLDG \( G^r \), we get \( \delta^r(e) = \delta(e) + \vec{r}(u) - \vec{r}(v) \) for each edge \( e : u \rightarrow v \). The summation of the edge weights in a cycle remains a constant after retiming. The meaning of retiming value \( \vec{r}(u) \) is that a node \( u \) originally executed in the iteration \( \vec{i} \) is shifted to the iteration \( \vec{i} - \vec{r}(u) \) [8]. For a data flow graph, a new iteration consists of nodes from different iterations. For a loop dependence graph, all the computations of loop \( u \) are executed \( \vec{r}(u) \) iterations earlier [8].”

A **two-dimensional retiming** \( (\vec{r}) \) is a function from \( V \) to \( Z^2 \). Retiming recon-
struct the iterations in a loop, Figure 2.5(a) shows the retimed DFG with retiming function \( r(D) = (1, 0) \) for the original DFG shown in Figure 2.3(b). Some copies of some statements of the original loop are shifted before the loop body and the prologue is generated; Some copies of some other statements of the original loop are shifted after the loop body and become the epilogue. The prologue and the epilogue is to used to complete the execution of the whole loop after retiming. The prologue section of a retimed loop is composed of the set of statements that have to be executed before the loop body to provide the necessary data for the iterative process. Correspondingly, the statements that will be executed after loop body to complete the process are called epilogue. We can use the computed retiming value to decide the number of copies of a statements \( u \) needed to be shifted into prologue or epilogue [20].

2.7 Data Locality

In computer science, locality of reference, is the phenomenon of the same variable being frequently accessed. There are two basic types of locality, Temporal locality, and Spatial locality as explained as follows.

- Temporal locality (locality in time): if a data item is referenced, it will tend to be referenced again soon.

- Spatial locality (locality in space): if a data item is referenced, the data items whose addresses are close by will tend to be referenced soon.

Consider the following example from Figure 2.6:

In the Figure 2.6, it includes these two types of data locality. For the access of \( A[i] \), it is temporal locality, since memory location of \( A[i] \) is referred twice in each
For i = 0 to n
  S1: A[i] = B[i] + 4
  S2: C[i] = A[i] − B[i+1]
Endfor

Figure 2.6. An example loop.

iteration. For the access of array B, it not only owns temporal locality but also has spatial locality. Because this array is allocated to a contiguous memory space, after accessing an element of B in the first statement, the item that next to this element will be accessed immediately in the second statement of each iteration.

There are two reasons for the occurring of data locality. Firstly, highly related data is always stored in neighboring memory locations such as array and vector. Secondly, programs involving loops usually access arrays or vectors by index. Due to these properties, it is feasible to use cache to take advantage of data locality to improve system performance.

2.8 Cache Organization

Because of the speed gap between CPU and main memory, computer designers have introduced cache memories. Cache serves as a component that temporary stores data so that later accesses for that data can be provided faster [14]. So cache plays a very important role in computer architecture. The working mechanism of cache is illustrated as follows: After we get the physical address of a variable from the memory management unit, we partition the physical address into three parts, tag, index, and offset. According to the index part, we can get the corresponding entry in the cache directory. Then we compare the tag in the cache directory with the tag in the physical
address. If they are the same, it is cache hit. Otherwise it is cache miss. If it is a cache miss, the cache will manage to get the data from next level of memory hierarchy, and it will take hundred of clock cycles.
CHAPTER 3

OPTIMIZING MEMORY COST WITH LOOP TRANSFORMATIONS

In this chapter, we will first introduce the code-size estimation. Then we will propose and investigate a scheme to compute the memory cost of a basic block and a loop nest based on the data dependency distance. Next, we propose to combine various loop transformation techniques to achieve the minimal memory cost. For a single loop nest, we try to apply loop permutation to reduce memory cost whenever it is legal. For a group of loops, we will try to apply loop permutation to enable loop fusion and then fuse the loops together, so that the code size and the memory cost will be minimal. If loop fusion is still not legal after the first two steps, we will apply retiming to legalize loop fusion, then fuse loops to reduce memory cost. The simulations showed that compared with the running time of the original loops, the running time of the transformed loops by using our loop optimization strategy is significantly reduced.

3.1 Memory-Size Estimation

Many applications, especially in the multi-media and telecom domains, are inherently data dominant [19]. For these kind of applications, memory performance dominates system performance. Also, memory usually occupies half of the surface of the integrated chip of multi-core system; A main factor to decide the area of the multi-core CPU chip. Memory size estimation and data dependency analysis will play a key role in exploring the memory hierarchy design [1, 17, 18, 20]. Memory Size Estimation is
based on static program analysis and data dependency analysis. It can be divided into two parts: code size estimation and data size estimation.

### 3.1.1 Memory Model

The definition of Memory model is defined in [4], “During the compiling phase, the compiler must determine, for each variable computed in the program, where that variable will reside. For the program to execute, the compiler must assign a specific location, such as a register or a memory location. In general, compilers choose from one of two memory models, Register-to-Register Model or Memory-to-Memory model. Under the Register-to-Register Model, the compiler keeps values in registers aggressively, ignoring any limitations imposed by the size of the machine’s physical register set. Any value that can legally be kept in a register for most of its lifetime is kept in a register. Values are stored to memory only when the semantics of the program require it. Under the Memory-to-Memory model, the compiler assumes that all values are kept in memory locations. Values move from memory to a register just before they are used. In this thesis, we choose Memory-to-Memory model [4].”

### 3.1.2 Code-Size Estimation

Code-Size Estimation has been defined in [9], here we just repeat for clarity. “After we choose the specific Memory Model, in the compiling phase of an application program, we apply static program analysis to estimate the code size of the target machine code. Given the input source program, we convert the source program into the intermediate representation to estimate the target machine code size. We estimate the code size for the basic blocks and then for the program region, and then for the whole program. A region is a group of closely related basic blocks such as superblocks and the body of a
A three address code statement will be generally translated into four machine instructions as shown in Figure 3.1. A loop control instruction or a branch control instruction is generally translated two machine instructions. Based on these basic facts, we can estimate the code size by the number of instructions and their intermediate format code. [9]”

![Figure 3.1. An intermediate format statement and its machine code.](image)

### 3.1.3 Memory-Size Estimation of a Loop

The total amount of memory needed for a loop is computed based on the given loop and its corresponding data flow graph. The memory cost \(MC\) of a data dependency represented by an edge weighted by \((d_1, d_2, \cdots, d_n)\) in the data flow graph is computed using the bounding box approximation.

\[
MC = 1 + \sum_{i=1}^{n} (d_i \times \prod_{j=i}^{n} (UB_{DD_j} - LB_{DD_j})),
\]

where \(n\) is the number of array dimensions, \(UB\) and \(LB\) are the bounding box boundaries for each dimension. The maximum memory cost \((MMC)\) of a computation node \(v\) in the data flow graph is defined as the maximum of the memory units consumed of all the outgoing edges, i.e.,
$$MMC(v) = \max_{e \in \mathcal{E}} (MC(e)) \) .

Each node in the data flow graph is a computation node, and its computation result may be used by multiple computation nodes. Thus, the computation node may have multiple outgoing edges representing the data dependencies whose source data is the result of this computation node. For example, in the loop shown in Figure 3.2, the result of the computation node A will be used by both the computation nodes B and C, so computation node A has two outgoing edges \( e_1 : A \rightarrow B \) and \( e_2 : A \rightarrow C \) representing the data dependency between the computation nodes A and B, and the data dependency between the computation nodes A and C, respectively.

Intuitively speaking, the computation of \( C[i, j] \) needs to use the value of \( A[i, j-3] \), which is computed 3 iterations before. The computation of \( B[i, j] \) needs to use the value of \( A[i, j] \), which is computed in the same iteration. If four memory units are allocated to hold at least 4 elements of array A, then in iteration \( (i,j) \), \( A[i, j-3] \) computed 3 iterations before is still in the local memory.

For a given loop, the total memory cost of this loop is defined as the summation of the memory units of each computation node in the data flow graph of the given loop, i.e., \( TMC(L) = \sum_{v \in V} (MMC(v)) \).

\[
\begin{align*}
\text{For } i &= 1, N \\
\text{For } j &= 3, M \\
S1: &\quad A[i,j] = 3 + D[i-1,j]*3; \\
S2: &\quad B[i,j] = 5 - A[i,j-3]; \\
S3: &\quad C[i,j] = A[i,j]*4; \\
S4: &\quad D[i,j] = B[i,j-2] + C[i-2,j-1];
\end{align*}
\]

Figure 3.2. An example loop.
For example, for the loop shown in Figure 3.2, the memory cost of computation node A is 4, the memory cost of computation node B is $M - 2$, the memory cost of computation node C is $2M - 4$, and the memory cost of computation node D is $M - 2$, so the total memory cost of the loop is $TMC = MMC(A) + MMC(B) + MMC(C) + MMC(D) = 4M - 4$.

<table>
<thead>
<tr>
<th>Loop ordering</th>
<th>Node A</th>
<th>Node B</th>
<th>Node C</th>
<th>Node D</th>
<th>Total Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i,j)</td>
<td>4</td>
<td>M-2</td>
<td>2M-4</td>
<td>M-2</td>
<td>4M-4</td>
</tr>
<tr>
<td>(j,i)</td>
<td>3M-8</td>
<td>2M-5</td>
<td>M</td>
<td>2</td>
<td>6M-11</td>
</tr>
</tbody>
</table>

Table 3.1. Memory size estimation

3.2 Loop Transformations for Memory Cost Reduction

Data locality is improved by loop transformation including loop permutation [1, 10, 11, 17] and loop fusion. Thus, the lifetime of data elements is reduced. This lifetime reduction of data elements could increase the possibility of reuse of memory locations. Then data with non-overlapping lifetimes can be assigned to the same physical address. The overall runtime storage requirement of the application is reduced accordingly.

In this section, we will try to combine various loop transformation techniques based on data dependency analysis and liveness analysis to further minimize the data memory size. Our proposed loop optimization strategy is as follows.

1. For a single loop nest, test if loop permutation is legal; If it is legal, apply loop permutation to reduce memory cost. The algorithm to test the legality of loop permutation is shown in Alg. 3.2.1.

2. For a group of loops, test if loop permutation is legal; If it is legal, then test if loop permutation can eliminate fusion-preventing dependencies to enable loop fusion.
If so, apply loop permutation to enable loop fusion, then fuse loops to reduce memory cost. The algorithm 3.2.2 is to test if loop permutation can be used to eliminate loop-fusion preventing dependencies. If the count returned from the algorithm 3.2.2 is larger than 0, then loop permutation eliminated some fusion-preventing dependencies. If all the fusion-preventing dependencies are eliminated, then loop permutation can be used to enable loop fusion. If there are several possible loop permutation to enable loop fusion, we will choose a particular permutation so that the fused loop achieves minimal memory cost.

3. If loop permutation cannot eliminate all the fusion-preventing dependencies, then apply retiming to legalize loop fusion, then fuse loops to reduce memory cost. The legalizing loop fusion algorithm will be presented in the next section.

**Algorithm 3.2.1 Legality Testing of Loop Permutation**

**Input:** A data flow graph $G = (V, E, w, t)$ and a given permutation

**Output:** flag, to show if the loop permutation is legal or not

1. $flag \leftarrow 0$
2. for all edges $e \in E$ do
   
   Apply loop permutation, $w'(e)$ is the data dependency after the given loop permutation for the original data dependency $w(e)$
3. end for
4. for all edges $e \in E$ do
   
   if $w(e) > (0, \cdots, 0)$ and $w'(e) < (0, \cdots, 0)$ then
   
   $flag \leftarrow 1$
   
   end if
   
   if $w(e) < (0, \cdots, 0)$ and $w'(e) > (0, \cdots, 0)$ then
   
   $flag \leftarrow 1$
   
   end if
5. end for
6. return $flag$

For the example loop shown in Figure 3.2, after applying the algorithm 3.2.1,
the flag we get is 0, so it is legal to apply loop interchange. The storage requirements for the original loop and the transformed loop are summarized in Table 3.1. It is clear that different loop ordering can influence the memory cost significantly. The loop order \((i, j)\) is much better than the loop order \((j, i)\) when the total memory cost of the loop is concerned. So in this case, we will not apply loop permutation.

**Algorithm 3.2.2 Loop Permutation to Enable Loop Fusion**

**Input:** A loop and its corresponding loop dependency graph \(G = (V, E, \delta, o)\) and a given permutation

**Output:** \(\text{count}\)

\(\text{count}_{\text{disable}}\) is the number of data dependencies that was changed into fusion-preventing dependencies from non fusion-preventing dependencies by the given loop permutation.

\(\text{count}_{\text{disable}} \leftarrow 0\)

\(\text{count}_{\text{enable}}\) is the number of the fusion-preventing dependencies eliminated by the given loop permutation.

\(\text{count}_{\text{enable}} \leftarrow 0\)

\(\text{for all } e \in E \text{ do}\)

Apply loop permutation, \(\delta'(e)\) is the data dependency after the given loop permutation for the original data dependency \(\delta(e)\)

\(\text{end for}\)

\(\text{for all } e \in E \text{ do}\)

\(\text{if } \delta(e) \geq (0, \cdots, 0) \text{ and } \delta'(e) < (0, \cdots, 0) \text{ then}\)

\(\text{count}_{\text{disable}} \leftarrow \text{count}_{\text{disable}} + 1\)

\(\text{end if}\)

\(\text{if } \text{delta}(e) < (0, \cdots, 0) \text{ and } \delta'(e) \geq (0, \cdots, 0) \text{ then}\)

\(\text{count}_{\text{enable}} \leftarrow \text{count}_{\text{enable}} + 1\)

\(\text{end if}\)

\(\text{end for}\)

\(\text{count} \leftarrow \text{count}_{\text{enable}} - \text{count}_{\text{disable}}\)

\(\text{return count}\)

The example loop shown in 3.3 showed that loop permutation can be used to enable loop fusion. In the original loop, the elements of A is alive after the computation
of the first loop, so the storage requirement for array A is \( N \times M \). After we legalize loop fusion by loop permutation, the storage requirement for array A in the fused loop is \( M + 1 \).

\[
\begin{align*}
\text{(a) Original Loops} & \quad \text{for } i=1, N \\
& \quad \text{for } j=1, M \\
& \quad A[i,j] = (A[i-1,j] + A[i,j-1]) \times 0.5; \\
& \quad \text{endfor} \\
& \quad \text{endfor} \\
\text{(b) The loops after loop permutation} & \quad \text{for } j=1, M \\
& \quad \text{for } i=1, N \\
& \quad A[i,j] = (A[i-1,j] + A[i,j-1]) \times 0.5; \\
& \quad \text{endfor} \\
& \quad \text{endfor} \\
\text{(c) Fused Loop} & \quad \text{for } j=1, M \\
& \quad \text{for } i=1, N \\
& \quad A[i,j] = (A[i-1,j] + A[i,j-1]) \times 0.5; \\
& \quad \text{endfor} \\
& \quad \text{endfor} \\
\end{align*}
\]

Figure 3.3. Loop permutation is used to legalize loop fusion.

### 3.3 Legalizing Loop Fusion with Improved Data Locality

In this section, we propose the Select\_LF technique to select one of the possible dimensions to legalize loop fusion such that the memory cost of the fused loop is significantly reduced compared with the original loops.

#### 3.3.1 Possible Dimensions for Legalizing Loop Fusion

When loop permutation cannot eliminate all the fusion-preventing dependencies to enable loop fusion, we will apply retiming to legalize loop fusion, then fuse loops to reduce memory cost. In this subsection, we provide a theorem to show which dimensions are possible to be retimed to legalize loop fusion. Before we present the theorem for the legalizing loop fusion, we need to define the minimum preventing dimension (\( \text{Min\_Prev} \)) of an LDG. First, we define the preventing-dimension (\( \text{Prev\_Dim} \)) of an edge weight \( \delta(e) \). If the edge weight \( \delta(e) \) of an edge \( e \) in the LDG of a nested N-level loop with J-level shared outer loop satisfies that \( (\delta_1(e), \delta_2(e), \cdots, \delta_N(e)) \geq \)


Then, we define the preventing-dimension $\text{Prev}_\text{Dim}$ of the edge weight $\delta(e)$ to be $\text{Prev}_\text{Dim}(\delta(e)) = \infty$. Otherwise, if an edge $e$ has a negative edge weight, i.e., $(\delta_1(e), \delta_2(e), \cdots, \delta_N(e)) < (0, \cdots, 0)$, then we define the preventing-dimension $\text{Prev}_\text{Dim}$ of $\delta(e)$ as $\text{Prev}_\text{Dim}(\delta(e)) = i$, such that $(\delta_1(e), \ldots, \delta_{i-1}(e)) = (0, \cdots, 0)$, and $\delta_i(e) < 0$, i.e., $\text{Prev}_\text{Dim}(\delta(e))$ is the minimum dimension of the edge weight vector $\delta(e)$ that gives a negative value. For example, for an edge $e$ with negative edge weight $\delta(e) = (0, -1, 0)$, the first negative element is in the second dimension, so $\text{Prev}_\text{Dim}(\delta(e)) = 2$, and for an edge with positive edge weight $\delta(e) = (0, 1, 0)$, $\text{Prev}_\text{Dim}(\delta(e)) = \infty$.

![Figure 3.4](image-url)

Figure 3.4. (a) An example loop’s Loop Dependence Graph. (b) Retiming the 2nd dimension’s constraint graph. (c) The retimed loop dependence graph by retiming the 2nd dimension.

Then, we define the minimum preventing dimension ($\text{Min}_\text{Prev}$) of a LDG to be the minimum dimension to give the negative value by all the edge weights of the LDG, i.e., $\text{Min}_\text{Prev} = \min_e \{ \text{Prev}_\text{Dim}(\delta(e)) \}, \forall e$. From the definition, we can see that all the edges $e$ with edge weight $\delta(e) \geq (0, \cdots, 0)$ must have $\text{Prev}_\text{Dim}(\delta(e)) = \infty$. Only the preventing dependence edges $e$ with $\delta(e) < (0, \cdots, 0)$
have the property that \( J + 1 \leq \text{Prev}_\text{Dim}(\delta(e)) \leq N \) for any \( N \)-level nested loop. The computed minimum preventing dimension (Min\_Prev) is the minimum dimension that we can retime to eliminate the fusion-preventing dependences.

In Table 3.2, we show how to compute the preventing dimension (Prev\_Dim) of the edge weights and the minimum preventing dimension (Min\_Prev) of the LDG shown in Figure 3.4(a). For example, for edge \( \{e1 : L1 \rightarrow L2\} \), \( \delta(e1) = (0, 0, -1) \), the first negative element of the negative edge weight \((0, 0, -1)\) is in the third dimension, so \( \text{Prev}_\text{Dim}(\delta(e1)) = 3 \). For edge \( \{e2 : L2 \rightarrow L3\} \), \( \delta(e2) = (0, 0, 3), (0, 0, 3) \) is a positive edge weight, so \( \text{Prev}_\text{Dim}(\delta(e2)) = \infty \). We take the minimum value of all the \( \text{Prev}_\text{Dim} \) as the minimum preventing dimension, so \( \text{Min}\_\text{Prev} = 3 \).

<table>
<thead>
<tr>
<th>Edges</th>
<th>( {e1 : L1 \rightarrow L2} )</th>
<th>( {e1 : L2 \rightarrow L3} )</th>
<th>( {e1 : L3 \rightarrow L4} )</th>
<th>( {e1 : L4 \rightarrow L5} )</th>
<th>( {e1 : L5 \rightarrow L6} )</th>
<th>( {e1 : L6 \rightarrow L1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge weight</td>
<td>((0, 0, -1))</td>
<td>((0, 0, 3))</td>
<td>((0, 0, -1))</td>
<td>((0, 0, 3))</td>
<td>((0, 0, -1))</td>
<td>((1, 0, 0))</td>
</tr>
<tr>
<td>\text{Prev}_\text{Dim}</td>
<td>3</td>
<td>\infty</td>
<td>3</td>
<td>\infty</td>
<td>3</td>
<td>\infty</td>
</tr>
<tr>
<td>\text{Min}_\text{Dim}</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2. The computation of \( \text{Prev}_\text{Dim} \) and \( \text{Min}_\text{Prev} \) of the LDG shown in Figure 3.4(a).

Based on the loop properties, Theorem 3.3.1 state that we can retime one of the possible dimensions from \((J + 1)\)-th dimension to \( \text{Min}_\text{Prev} \)-th dimension to legalize loop fusion for a \( N \)-level loop with \( J \)-level shared outer loop.

**Theorem 3.3.1.** \( \text{Given a} \ N \)-level nested loop with \( J \)-level shared outer loop and its corresponding loop dependence graph \( G = (V, E, \delta, o) \), we can retime one of the possible dimensions from the \((J + 1)\)-th dimension to \( \text{Min}_\text{Prev} \)-th dimension to legalize loop fusion, where \( J + 1 \leq \text{Min}_\text{Prev} \leq n \).**

### 3.3.2 The Select\_LF Technique

According to Theorem 3.3.1, we can retime one of the possible dimensions from \( J + 1 \) to \( \text{Min}_\text{Prev} \) to legalize loop fusion for an \( n \)-level loop with \( J \)-level shared outer loop.
We propose the Select_LF technique as shown in Algorithm 3.3.1 to search and select one of the possible dimensions from $J + 1$ to $\text{Min}_\text{Prev}$ to legalize loop fusion such that the resultant code size of the fused loop is minimal. When $\text{Min}_\text{Prev} = (J + 1)$, then we can only retime the $(J + 1)$-th dimension.

**Algorithm 3.3.1 Selecting Dimension to Legalize Loop Fusion (Select_LF)**

**Input:** A loop dependence graph $G = (V, E, \delta, o)$

**Output:** Dimension number $\text{dim}$

/* Compute the minimum preventing dimension */

$\text{Min}_\text{Prev} = \min_e \{ \text{Prev}_\text{Dim}(\delta(e)) \}, \forall e$

$\text{dim} = J + 1, \text{size} = \infty$

for $i \leftarrow J + 1$ to $\text{Min}_\text{Prev}$ do

    Compute the set of retiming values $R_i$ for dimension $i$ to legalize loop fusion by the Bellman-Ford shortest path algorithm

    Compute the maximum retiming value $R_i$ for $i$-th dimension

    Compute the memory cost $C_i$ of the fused loop using the retiming values for $i$-th dimension

    if $C_i < \text{size}$ then

        $\text{dim} \leftarrow i, \text{size} \leftarrow C_i$

    end if

end for

return $\text{dim}$

In Algorithm 3.3.1, we computed the retiming values using the Bellman-Ford shortest path algorithm for all the possible dimensions from $(J + 1)$-th dimension to $\text{Min}_\text{Prev}$-th dimension. Also, we computed the maximum retiming value for each dimension from $(J + 1)$-th dimension to $\text{Min}_\text{Prev}$-th dimension. Based on the retiming values computed for each dimension, we compute the memory cost of the fused loop using the memory-cost calculation formula proposed in chapter 3. Then, we select the dimension to legalize loop fusion such that the memory cost of the fused loop is minimal.
For example, for the LDG shown in Figure 3.4(a), since $J = 1$, $Min_{Prev} = 3$, we can retime either the second or the third dimension to legalize loop fusion according to theorem 3.3.1. The constraint graph for computing the retiming values for the second dimension is shown in Figure 3.4(b), and the normalized retiming values computed by the Bellman-Ford algorithm are $r(L_1) = (0, 3, 0), r(L_2) = (0, 2, 0), r(L_3) = (0, 2, 0), r(L_4) = (0, 1, 0), r(L_5) = (0, 1, 0), r(L_6) = (0, 0, 0)$. The maximum retiming values for the second dimension is $R_2 = 3$. Similarly, we can construct the constraint graph for computing the retiming values for the third dimension. An the normalized retiming values by retiming the third dimension are $r(L_1) = (0, 0, 2), r(L_2) = (0, 0, 0), r(L_3) = (0, 0, 2), r(L_4) = (0, 0, 0), r(L_5) = (0, 0, 2), r(L_6) = (0, 0, 0)$. The maximum retiming value for the third dimension is $R_3 = 2$. The maximum retiming value for the third dimension is less than the maximum retiming value for the second dimension, i.e., $R_3 < R_2$. Also the memory cost of the fused loop by retiming the third dimension is much less than the memory of the fused loop by retiming the second dimension.

The Select_LF technique as shown in Algorithm 3.3.1 does the exhaustive search to select one dimension to legalize loop fusion such that the resultant memory cost of the fused loop is minimal. For the example LDG shown in Figure 3.4(a), the dimension selected by Algorithm 3.3.1 will be 3, since the memory cost of the fused loop by retiming the third dimension is less than the memory cost of the fused loop by retiming the second dimension. Compared to the number of the loop units, the dimension of the edge weight is a small number. So our algorithm is polynomial time with respect to the number of the loop units (the number of the nodes in the LDG).
3.4 Legalizing Loop Fusion Technique with Minimal Memory Cost

In the previous section, we proposed a legalizing fusion technique to improve the data locality and reduce the memory cost. But the memory cost of the fused loop does not necessarily be minimal. So we want to propose a legalizing technique to minimize the total memory cost of the fused loop while legalizing loop fusion. That is to compute the retiming values to legalize loop fusion and also the computed retiming values will make the total memory cost of the fused loop to be minimal. We use ILP (Integer Linear Programming) to solve the problem such that the computed retiming values are optimal. The constraint of the ILP is the inequalities to compute the retiming values such that there is no fusion-preventing dependences after retiming, and the objective function is to minimize the total dependence distance of the fused loop.

The input of the problem is a loop dependence graph, and the output is the computed retiming values. Assuming for an edge $e$ in the multi-edge loop dependence graph, its weight after the retiming is $\delta^r(e) = \delta(e) + r(u) - r(v)$, the inequality $r(v) - r(u) \leq \delta(e)$ must be true to satisfy that $\delta^r(e) \geq (0, \cdots, 0)$ to eliminate all the data dependence distances. A solution of the integer linear programming problem can be used as the retiming values, where $r_n$ represent the retiming value on the $n^{th}$ node.

We classify the outgoing edges according to the array name associated with it in the computation of the objective function, since the total memory cost is the summation of the memory cost of each computation node in the loop. When we compute the objective function, for each computation of one array in a loop node, there is one corresponding item in the objective function. That is one loop node might correspond to several items in the corresponding objective function. For each array computed in one loop, we select the maximum data dependence of this array to reflect the memory space needed, i.e., for all the outgoing edges that associated with the same array,
we select the edge with the maximum data dependence vector in the computation of the objective function. In summary, the ILP form for the given problem is shown as follows:

**ILP forms:**

\[ r_1 - r_2 \leq \delta(1,2) \]
\[ r_2 - r_3 \leq \delta(2,3) \]
\[ \ldots \]
\[ r_{n-1} - r_n \leq \delta(n-1,n) \]

The objective function: \( \min(\sum(\max(d^A_{\max}(v))), \forall v) \).

### 3.4.1 Motivation Example

The LDG shown in Figure 3.5 is the LDG for a loop with several sequential loops enclosed in one outer loop. For simplification of the presentation, suppose each loop node contains only one statement to compute the elements of one array. So we only need to select the outgoing edge with the maximum data dependence distance. When we do this, we are assuming that the outgoing edge with the maximum data dependence distance will be the edge with the maximum data dependence distance after retiming. There are fusion-preventing dependences in this example, and we want to remove the fusion-preventing dependences by transforming the loop and at the same time to minimize the dependence distance of the fused loop.

In order to minimize the dependence distance of the fused loop, we propose to retime both the second dimension and the third dimension, to legalize loop fusion based on the ILP form.

The constraint inequalities for the second dimension is:
There are four loop nodes in the loop dependence graph, for loop node \( L_1 \), there are two outgoing edges, edge \( e_1 : L_1 \rightarrow L_2 \) and \( e_2 : L_1 \rightarrow L_3 \). Edge \( e_2 \) has a larger data dependence vector originally, and we assume that it will have a larger data dependence vector after retiming. So for node \( L_1 \), we select the outgoing edge \( e_2 \) which has the data dependence vector \((0, 0, -1)\) when computing the objective function. Similarly, there are two outgoing edges for node \( L_2 \), edge \( e_3 : L_2 \rightarrow L_3 \) and \( e_4 : L_2 \rightarrow L_4 \), and the outgoing edge \( e_3 \) has a larger data dependence vector, so we select edge \( e_3 \) with data dependence vector \((0, 0, -1)\) when we compute the objective function.

\[
\begin{align*}
    r_2(L_2) - r_2(L_1) &\leq -1 \\
    r_2(L_3) - r_2(L_2) &\leq 0 \\
    r_2(L_4) - r_2(L_3) &\leq -1 \\
    r_2(L_3) - r_2(L_1) &\leq 0 \\
    r_2(L_4) - r_2(L_2) &\leq -1
\end{align*}
\]
function. Loop node $L_3$ has only one outgoing edge $e_5 : L_3 \rightarrow L_4$, so we select the outgoing edge $e_5$ with the data dependence vector $(0, -1, 0)$ in the object function. Loop node $L_4$ has no outgoing edge, so there is no corresponding items for loop node $L_4$ in the objective function. In summary, the objective function for the second dimension is:

$$\min(0 + r_2(L_1) - r_2(L_3) + 0 + r_2(L_2) - r_2(L_3) + (-1) + r_2(L_3) - r_2(L_4)) = \min(r_2(L_1) + r_2(L_2) - r_2(L_3) - r_2(L_4) - 1).$$

The computed retiming values for the second dimension are $r_2(L_1) = 2, r_2(L_2) = 1, r_2(L_3) = 1, r_2(L_4) = 0$.

Similarly, we can get the constraint inequalities for the third dimension, which are shown as follows:

$$r_3(L_2) - r_3(L_1) \leq -1$$

$$r_3(L_3) - r_3(L_2) \leq -1$$

$$r_3(L_4) - r_3(L_3) \leq 0$$

$$r_3(L_4) - r_3(L_2) \leq 0$$

And the objective function is:

$$\min((-1)+r_3(L_1)-r_3(L_3)+(-1)+r_3(L_2)-r_3(L_3)+0+r_3(L_3)-r_3(L_4)) = \min(r_3(L_1) + r_3(L_2) - r_3(L_3) - r_3(L_4) - 2)$$

The computed retiming values for the third dimension are $r_2(L_1) = 2, r_2(L_2) = 1, r_2(L_3) = 0, r_2(L_4) = 0$. Similarly, we can compute the retiming values for the third dimension. So the computed retiming values by the ILP are $r(L_1) = (0, 2, 2), r(L_2) = (0, 1, 1), r(L_3) = (0, 1, 0), r(L_4) = (0, 0, 0)$, and the transformed loop dependence graph is shown in Figure 3.5(b). Suppose the iteration distance for all dimensions
of the loop are all N, the memory cost of node $L_1$ of the fused loop is $MC(L_1) = N + 2$; the memory cost of node $L_2$ in the fused loop is $MC(L_2) = 2$; the memory cost of node $L_3$ in the fused loop is $MC(L_3) = 1$; the memory cost of node $L_4$ in the fused loop is $M(L_4) = 1$. So the total memory cost of the fused loop is $M = M(L_1) + M(L_2) + M(L_3) + M(L_4) = N + 6$. The transformed loop dependence graph by the Select_LF technique is shown in Figure 3.5(c). And the total memory cost of the fused loop is $4N + 2$.

### 3.4.2 Algorithm of Legalizing Loop Fusion with Minimum Memory Cost

In the following, we will propose the improved legalizing loop fusion algorithm based on the ILP-form to legalize loop fusion and simultaneously achieve the minimal memory cost of the fused loop. The input of the algorithm is a loop dependence graph, and the output is the computed retiming values. In the algorithm, for each loop node that have multiple statements to compute the elements of multiple arrays, we select the outgoing edge with the maximum data dependence associated with the computation of one specific array. That is, for each loop node, we may select multiple outgoing edges that are associated with the computation of multiple arrays. This is used in the computation of the objective function. Then, based on the purpose of the retiming, that is we want to eliminate all the fusion-preventing dependences in the original loop dependence graph, we construct the ILP inequalities for each edge $e : u \rightarrow v$, which satisfies that $d_r(e) = d(e) + r(u) - r(v) \geq 0$. The objective function is to minimize the total data dependence distance of the fused loop, i.e., to minimize the memory cost of the fused loop. There are one corresponding item of the computation of one specific array in the objective function. The algorithm is shown in algorithm 3.4.1.
Algorithm 3.4.1 Legalizing Loop Fusion with Minimum Memory Cost

**Input:** A Loop Dependence Graph (LDG) $G = (V, E, D, \delta)$

**Output:** A retiming function $r$ of the loop dependence graph

/* Compute the maximum data dependence distance for the computation of each data array for each loop node */

for all $v \in V$ do
  for all data arrays $A$ computed in loop node $v$ do
    $d_{A_{max}}(v) \leftarrow \max(d : D(v, w))$, where $w$ is the set of the destination nodes of all the outgoing edges $e \in E$ of node $v$ associated with the computation of data array $A$, and $D(v, w)$ is the set of data dependence distances of all the outgoing edges $e$ of node $v$ associated with the data array $A$, $d_{A_{max}}(v)$ is the maximum data dependence distance corresponding the computation of array $A$ in loop node $v$
  end for
end for

Construct the ILP constraints for all the edges $e \in E$ to compute the retiming values with the defined objective function.

The ILP forms are:

$r_1 - r_2 \leq \delta(1, 2)$
$r_2 - r_3 \leq \delta(2, 3)$

......
$r_{n-1} - r_n \leq \delta(n-1, n)$

The objective function is: $\min(\sum_v(d_{max}(v)), v \in V), d_{r_{max}}(v) = d_{A_{max}}(v) + r(v) - r(w)$

Applying the computed retiming values on the $i$-th dimension

Compute the memory cost of the fused loop

return $r$

3.5 Simulation

In the simulation, we simulated a superscalar DSP processor with one-level cache on the chip under the Simplescalar environment. SimpleScalar was developed to serve as a computer architecture simulator [3]. It has a set of tools which can simulate a virtual computer system. By using the SimpleScalar, the user can set the configuration of the virtual computer system according to the features they need easily.
3.5.1 SimpleScalar Installation

In this subsection, we will introduce how to install SimpleScalar on ubuntu 9.04 system step by step. Before we begin to install SimpleScalar, we need to update the repository listing.

$ sudo apt-get update  Once the repository is updated, the dependency packages can be installed. Next, a temporary directory is created as the workspace for installing SimpleScalar and the downloaded files.

$ mkdir /tmp/simplescalar
$ cd /tmp/simplescalar
$ wget http://csrl.unt.edu/downloads/simplescalar.tgz
$ tar xvfz simplescalar.tgz

Because only certain versions of gcc are compatible with SimpleScalar, here we choose to install gcc version 3.4.

$ sudo apt-get install gcc-3.4
$ export CC="gcc-3.4"
$ export HOST=i686-unknown-linux
$ export TARGET=sslittle-na-sstrix
$ export IDIR=/opt/simplescalar

In order to make the task of installing and configuring easier, here we use export to export several environment variables. The next step is to install the SimpleScalar tools. Note that the included version of gcc is not needed.

$ tar xvf simpletools-2v0.tgz
$ rm -rf gcc-2.6.3
$ sudo mkdir -p /opt/simplescalar
After the SimpleScalar tools are installed, the SimpleScalar utilities are installed.

```
$ tar xvfz simpleutils-990811.tar.gz
$ cd /tmp/simplescalar/simpleutils-990811
$ ./configure --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR
$ make CC=gcc-3.4
$ sudo make install CC=gcc-3.4
```

Now it is time to install SimpleScalar and move it to `/opt` where everyone can access it.

```
$ tar xvfz simplesim-3v0d.tgz
$ cd simplesim-3.0
$ make config-pisa
$ make CC=gcc-3.4
$ cd /tmp/simplescalar sudo mv simplesim-3.0 /opt/simplescalar
```

Next, we are going to configure the SimpleScalar.

```
$ cd /tmp/simplescalar/ tar xvfz gcc-2.7.2.3.ss.tar.gz
$ cd /tmp/simplescalar/gcc-2.7.2.3
$ export PATH=$PATH:$IDIR/sslittle-na-sstrix/bin
$ ./configure --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR
```

Before we can use the makefile, one of the source file contains syntax errors that must be corrected.

```
$ gedit insn-output.c
```

A backslash must be added to the end of lines 675, 750 and 823. The makefile also contains an error and must be changed.
$ gedit Makefile

At the end of line 130, add `-I/usr/include`. There is another library file that must be changed, and also write access is needed.

$ chmod 755 cxxmain.c
$ gedit cxxmain.c

The functions malloc() and realloc() are redefined here in lines 2978 and 2979. These lines should be commented out. Now, the makefile can be used.

$ make LANGUAGES="c c++" CFLAGS=-O3 CC=gcc-3.4
$ sudo cp patched/sys/cdefs.h /opt/simplescalar/sslittle-nsstrix/include/sys/
$ make enquire CC=gcc-3.4
$ /opt/simplescalar/simplesim-3.0/sim-safe ./enquire -f float.h-cross
$ sudo make install LANGUAGES="c c++" CFLAGS=-O3 CC=gcc-3.4
$ PATH=$PATH:/opt/simplescalar/bin

Then, SimpleScalar should be installed and ready for use.

### 3.5.2 Performance

<table>
<thead>
<tr>
<th>Benchmark Programs</th>
<th>Number of Loops</th>
<th>Execution Time (millisecond)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>Fused</td>
</tr>
<tr>
<td>Livermore14</td>
<td>4</td>
<td>16.5</td>
<td>4.8</td>
</tr>
<tr>
<td>Livermore18</td>
<td>7</td>
<td>28.6</td>
<td>27.2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>4</td>
<td>8.2</td>
<td>4.5</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>13</td>
<td>158.5</td>
<td>92.8</td>
</tr>
</tbody>
</table>

Table 3.3. Execution time comparison.

In the simulation, we implemented the select loop fusion technique (Select LF), and the improved legalizing loop fusion technique with minimal cost (LF MMC). We
simulated a superscalar DSP processor with one-level cache on the chip under the SimpleScalar environment. There are two different cache settings. One of the cache settings is a 1-way cache with a 32Byte cache line and a capacity of 16KB. The other cache settings is a 2-way cache with a 32Byte cache line and a capacity of 32KB.

In the simulation, we use some benchmarks including SPEC95, Livermore, Jacobi, and Tomcatv. We model the programs by loop dependence graph and data dependence graph. The comparison of the timing performance and the cache misses of the original loops, and the fused loops was used to validate our proposed technique.

LL14 is the 14th kernel from the Livermore benchmark. LL18 is the 18th kernel program taken from the Livermore benchmark. Tomcatv is a program selected from the benchmark SPEC2000 suite that has five loop nests. Jacobi is a program selected from the Jacobi benchmark that has two loop nests. Among these test programs, LL18 and Jacobi are fused by applying the LF_MMC technique, and the other two test programs are fused by applying the Select_LF techniques.

In the simulation, we use the cache setting of 1-way cache with a 32Byte cache line and a capacity of 16KB under SimpleScalar environment. The running time of the original loops and the fused loops was presented in Table 3.3. The first column of the table shows the name of the benchmark program. The second column gives the

<table>
<thead>
<tr>
<th>Benchmark Programs</th>
<th>Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16KB, 1 way, Cache line of 32 Bytes</td>
</tr>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Livermore14</td>
<td>0.1673</td>
</tr>
<tr>
<td>Livermore18</td>
<td>0.0627</td>
</tr>
<tr>
<td>Jacobi</td>
<td>0.0229</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>0.0728</td>
</tr>
</tbody>
</table>

Table 3.4. Cache misses for different cache settings.
number of the loops in the original program. The third column shows the running time of the original program. The fourth column gives the execution time of the program after loop fusion is applied. The fifth column reports the performance improvements.

The cache miss rates of the original loops and the transformed loops was presented in Table 3.4. The first column of the table gives the name of the benchmark program. The second column and the third column gives the cache misses of the original program and the transformed program under the cache setting of 1-way cache with a 32Byte cache line and a capacity of 16KB. The fourth column and the fifth column gives the cache misses of the original program and the transformed program under the cache setting of a 2-way cache with a 32Byte cache line and a capacity of 32KB. Compared with the original loops, the timing performance of the fused loop was increased 40.9% on average by our loop fusion technique as shown in Table 3.3.
CHAPTER 4

CONCLUSION

The increasingly complicated embedded systems require extensive design automation and optimization tools. Some specific architectures of the embedded processors, for example, VLIW architecture, and multi-core architectures, have been proposed to satisfy the high-performance requirements [1, 15, 17]. It is extremely important to exploit multiple levels of parallelism to take advantage of the hardware architecture. On the other hand, over the past 20 years, processor speed has become significantly faster than memory speed. It is critical to increase the data locality and data reuse to take advantage of the memory hierarchy and hide the memory latency to improve the overall performance of the embedded systems [1, 17].

In this thesis, we first propose and investigate a scheme to compute the memory cost of a basic block and a loop nest based on the data dependency distance. Then, we propose to combine various loop transformation techniques to achieve the minimal memory cost. For a single loop nest, we try to apply loop permutation to reduce memory cost whenever it is legal. For a group of loops, we will try to apply loop permutation to enable loop fusion and then fuse the loops together, so that the code size and the memory cost will be minimal. If loop fusion is still not legal after the first two steps, we will apply retiming to legalize loop fusion, then fuse loops to reduce memory cost. Our studies on the loop transformation techniques yield promising results on loop fusion, loop permutation, and memory cost reduction.
In this thesis, we focus on using loop transformation schemes to optimize the code and reduce the memory cost. We define the memory cost model to guide loop transformations. Our work results in new algorithms, and new loop transformation techniques that directly improve timing performance, and memory performance for embedded systems. In future, we will continue on the research work for compiler optimization techniques with multiple objectives for embedded systems, multi-core systems. Our goal is to use loop transformation techniques and their combination to improve the code quality and performance of the programs for embedded systems, and multi-core systems. This thesis can be extended in the following directions.

1. An integrated compiler framework will be developed to decide when to apply a particular loop transformation, and when to apply the combination of loop transformation techniques by studying the relationships between the different techniques for a variety of machine architectures.

2. Recently multi-core architecture has become the processor design trend. A multi-core embedded system creates new challenges for both hardware and software designs. A critical performance issue with multi-core architectures is to exploit multiple levels of parallelism including thread level parallelism, loop level parallelism, and instruction level parallelism and data level parallelism.

At the software level, in order to minimize communication cost among cores, more advanced partitioning and thread scheduling techniques are needed. From the hardware level, we need to study how to design a better memory hierarchy so the memory performance will be better in a multi-core environment. Based on our data dependency analysis, and memory cost model, we explore the memory hierarchy design, i.e., we should configure the on-chip memory to be software
controlled Scratch Pad Memory or Cache, and we also can propose algorithm to compute the scratch-pad memory or cache configurations for each loop nest.

3. Cache is an important component to decrease the gap between the processor and memory speed. The use of caches in multi-core systems has been limited due to cache coherency problems. It will be one of my future research topics to develop new memory hierarchy, and new cache partitioning techniques for multi-core systems.
REFERENCES


