SOFTWARE PARTITIONING AND SCHEDULING FOR IMPROVING PERFORMANCE AND ENERGY CONSUMPTION

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Zhong Wang, B.S., M.S.

______________________________
Xiaobo Sharon Hu, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2007
© Copyright by
Zhong Wang
2007
All Rights Reserved
SOFTWARE PARTITIONING AND SCHEDULING FOR IMPROVING PERFORMANCE AND ENERGY CONSUMPTION

Abstract

by

Zhong Wang

With the advances of the contemporary computer technology, the complexity grows significantly in both hardware architecture and software application. In order to meet the performance requirement of target applications, more and more emphasis is put on the compiler techniques to exploit both hardware and software parallelism. Scheduler, an important compiler component to allocate operations to hardware resources, is crucial to the success of a computing system. In this thesis, several novel scheduling optimization techniques are presented to address the challenge faced by existing computing architectures and applications.

The first targeted architecture is a system with memory hierarchy and processor comprising multiple processing and memory units. Loop partition scheduling technique is proposed to take advantage of the memory hierarchy and effectively hide the memory access latency for the loop-intensive applications. The concept of balanced partition schedule is presented to achieve the best memory access latency toleration and hardware resource utilization. Various extensions of the base problem are studied in depth. The solution are presented for the system model with multiple-level memory hierarchy, memory size constraint and loop model with initial data and multiple nested loops.
Multiple cluster architecture becomes more and more popular due to its superiority over centralized architecture. Inter-cluster communication, achieved by explicit register-to-register move, is compiler-controlled and invisible to the programmer. The thesis proposes an efficient scheduling algorithm which take into account ILP, register file size and inter-cluster communication constraints. Furthermore, the solution is completed by deliberate the effect of distributed caches. The consideration of data spilling, cache conflicts and cache communications are integrated into the algorithm.

Another target architecture is multi-bank memory architecture, which brings the scheduling complexity and difficulty of variable partitioning. The approach in the thesis not only improves the existing techniques when exploiting the parallelism, but also considers the serialism to take advantage of multiple operating modes of the memory banks. By identifying the best tradeoff between parallelism and serialism, both goals of performance and energy saving can be achieved. A novel memory access graph model, which captures both information of parallelism and serialism, forms the basis for this scheduling approach.
CONTENTS

FIGURES ................................................................. v

TABLES ................................................................. viii

CHAPTER 1: INTRODUCTION ............................................. 1
  1.1 Loop optimization with memory hierarchy ....................... 2
  1.2 Clustered architecture ........................................... 7
  1.3 Multi-bank memory system ....................................... 8
  1.4 Contribution .................................................... 10
  1.5 Organization of the thesis ...................................... 12

CHAPTER 2: BACKGROUND ................................................ 13
  2.1 Uniformization ................................................... 13
  2.2 Loop nest representation ....................................... 14
  2.3 Multi-dimensional retiming ...................................... 17
  2.4 Resource constraint scheduling ................................ 18
    2.4.1 Multi-dimensional rotation scheduling algorithm .......... 19
    2.4.2 Modulo scheduling ......................................... 21
    2.4.3 The comparison between two scheduling algorithms ....... 23

CHAPTER 3: PARTITIONING WITH THE MEMORY CONSTRAINT . 24
  3.1 Basic Framework ................................................ 26
    3.1.1 Partitioning the iteration space .......................... 26
    3.1.2 Memory unit operation ..................................... 28
    3.1.3 Architecture model ........................................ 31
    3.1.4 Framework of the algorithm ............................... 33
  3.2 Theoretical foundation .......................................... 34
  3.3 Algorithms ....................................................... 42
    3.3.1 Scheduling the ALU ........................................ 42
    3.3.2 Scheduling the memory .................................... 42
    3.3.3 Balanced schedule ......................................... 48
<table>
<thead>
<tr>
<th>FIGURES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The speed gap between memory and CPU</td>
<td>2</td>
</tr>
<tr>
<td>2.1 The MDFG representation of IIR filter</td>
<td>15</td>
</tr>
<tr>
<td>2.2 (a) MDFG extracted from a Wave Digital Filter (b) equivalent Fortran code</td>
<td>18</td>
</tr>
<tr>
<td>2.3 (a) MDFG after retimed by r(D)=(0,1) (b) equivalent Fortran code</td>
<td>19</td>
</tr>
<tr>
<td>2.4 (a) Initial MDFG (b) After retiming r(D) = (1,0) (c) After retiming r(A) = (1,0)</td>
<td>20</td>
</tr>
<tr>
<td>2.5 (a) Initial schedule (b) The schedule after rotated node D (c) The schedule after rotated node A</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Different kinds of memory operations and its corresponding regions</td>
<td>29</td>
</tr>
<tr>
<td>3.2 An example of memory arrangement</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Two adjacent partitions</td>
<td>35</td>
</tr>
<tr>
<td>3.4 The division of a partition under difference cases</td>
<td>37</td>
</tr>
<tr>
<td>3.5 the parallelogram decided by the delay vector</td>
<td>44</td>
</tr>
<tr>
<td>4.1 The architecture model</td>
<td>62</td>
</tr>
<tr>
<td>4.2 The different kinds of delay dependence for the second level memory</td>
<td>66</td>
</tr>
<tr>
<td>4.3 The area of the prefetch and writing back operation</td>
<td>68</td>
</tr>
<tr>
<td>4.4 (a) The first level partition schedule. (b) Two level partition schedule</td>
<td>71</td>
</tr>
<tr>
<td>4.5 (a) Prefetch operations in the second level partition. (b) Keep operation in the second level partition</td>
<td>72</td>
</tr>
<tr>
<td>4.6 The different sub-regions in the boundary partitions</td>
<td>73</td>
</tr>
<tr>
<td>4.7 The relationship between the average schedule length and the second memory size for IIR and DPCM</td>
<td>88</td>
</tr>
<tr>
<td>5.1 The footprint</td>
<td>93</td>
</tr>
<tr>
<td>5.2 One dimensional line segments</td>
<td>94</td>
</tr>
</tbody>
</table>
8.2 (a) An example DFG, where $L$ (resp., $S$) followed by an integer $i$ represents a \textit{LOAD} (resp., \textit{STORE}) operation on variable $i$. Other nodes are non-memory operations. Edges denote the precedence constraint between operation. (b) Schedule with only performance consideration. (c) Schedule when mode transition time is 2 cycles. (d) Schedule when mode transition time is 4 cycles.

8.3 (a) Memory operation mobility graph for DFG in 8.2(a) (b) Memory Access Graph with the first dimensional weight, each node number corresponds to the associated variable number. (c) Memory Access Graph with the second dimensional weight.

8.4 An example to illustrate the result of applying Algorithm 1. (a), The original assembly code given by SPAM compiler. (b). The assembly code after applying Algorithm 1.

8.5 Assembly code size results. The results are normalized with respect to the SPAM result.

8.6 Execution time of assembly code. The results are normalized with respect to the SPAM result.

8.7 Percent of low-current cycles over the total execution clock cycle.

8.8 Algorithm execution time.
TABLES

3.1 WITHOUT MEMORY CONSTRAINTS ASSUMING $T_{\text{prefetch}} = 10$ 55
3.2 REDUCING THE AVAILABLE MEMORY TO 2/3 OF THE OPTIMAL .......................... 56
3.3 MEMORY REQUIREMENT IS 1/2 OF THE ORIGINAL .................. 57

4.1 THE AVERAGE ACCESS TIME UNDER DIFFERENT ITERATION SPACE SIZES .......................... 64
4.2 WITHOUT MEMORY CONSTRAINTS ASSUMING $T_{\text{prefetch}} = 12$ 84
4.3 EXPERIMENTAL RESULTS WITH ABOUT 1/2 OF ORIGINAL SIZE .......................... 85
4.4 EXPERIMENTAL RESULTS WITH ABOUT 1/4 OF ORIGINAL SIZE .......................... 86

5.1 EXPERIMENTAL RESULTS WITH ONLY ONE INITIAL DATA 107
5.2 EXPERIMENTAL RESULTS WITH THREE INITIAL DATA .......................... 108

6.1 EXPERIMENT RESULT OF AVERAGE EXECUTION TIME .......................... 134

7.1 BENCHMARKS AND THE CLUSTERED CONFIGURATION .......................... 153
CHAPTER 1

INTRODUCTION

With the advances of the contemporary computer technology, the complexity grows significantly in both hardware architecture and software application. In order to meet the performance requirement of target application and take full advantage of the computing system architecture features, more and more emphasis is put on the compiler techniques to exploit both hardware and software parallelism. Scheduler, an important compiler component to allocate operations to hardware resources, is crucial to the success of a computing system.

When considering scheduling, two foremost factors are target architecture features and characteristics of target applications. Some architectures features are specially designed for their target applications. For instance, multiple memory banks are adopted to increase memory access parallelism for data intensive applications such as DSP and multimedia applications. Special instructions and hardware units are added in order to speed up some program pattern such as loop and data address calculation. The efficient schedule can only be generated with the full awareness of these architecture features. On the other hand, the restrictions posed by the architecture, e.g., register file, memory size, etc, should be respected to find the applicable schedule on the real system. With the traditional scheduling technique for general purpose computer architecture, it is hard and inefficient for them to take full advantage of these architecture features and sat-
isfy these constraints. Therefore, new compiler and scheduling approaches, which already integrated the considerations for these special features and constraints, are highly demanded.

In this chapter, we discuss the motivation, the problem we address and the contributions. The discussion is grouped based on the different target architecture features. The chapter concludes with an outline of the rest of the dissertation.

1.1 Loop optimization with memory hierarchy

![Figure 1.1. The speed gap between memory and CPU](image)

With the development of VLSI technology, the speed gap between memory and CPU become larger and larger. As illustrated in Figure 1.1. memory latency is becoming an increasingly important performance bottleneck as this gap continues
to grow. The CPU has to wait for the operands loaded from the memory, thereby
degrade the system performance. While cache hierarchies are an important step
toward addressing the latency problem, they are not a complete solution. Tech-
niques that can cope with the large latency of memory accesses are essential for
achieving high processor utilization [54]. In this thesis, several scheduling algo-
rithms are presented to tackle the problem of long memory access latency.

The traditional idea to hide the memory latency is through prefetching. The
previous research on prefetching can be classified into three categories: prefetching
based on hardware [19, 30, 77], software [49, 56], or both [16, 55, 97]. Hardware-
based prefetching techniques, requiring some support units connected to the cache,
rely on the dynamic information available during program execution. Software
prefetching schemes depend on compiler technology to analyze a program stati-
cally and insert explicit prefetch instructions into the program code. One advan-
tage of software prefetching is that much compile-time information can be explored
in order to effectively issue the prefetching instruction. Bianchini et al developed
a runtime data prefetching strategy for software-based distributed shared-memory
systems [9]. Wallace and Bagherzadel proposed a mathematical model and a new
prefetching mechanism. A simulation on the SPEC95 benchmarks showed an im-
provement in the instruction fetching rate [79]. Mowry proposes a new compiler
algorithm [53] for inserting prefetches into multiprocessor code. This algorithm
attempts to minimize overheads by only issuing prefetches for references that
are predicted to suffer cache misses. In their work, all major efforts are put on
scheduling prefetching, i.e., memory operations. The ALU part of the schedule
is not considered. Nevertheless, solely considering the prefetching is not enough
for optimizing the overall system performance. As we point out in this thesis, too
many prefetching operations may lead to an unbalanced schedule with a very long memory part schedule length. In contrast, our new algorithms give the detailed analysis of both the ALU and memory parts of the schedule. In order to compensate the memory latency, the concept of balanced schedule is proposed. It denotes a balanced relationship between the computations and memory loading.

One of the most common program patterns in computation intensive application is loop. Loop performance plays an important role to determine the overall system performance for many scientific applications. Several loop pipelining techniques have been proposed to improve the loop performance. Wang and Parhi presented an algorithm for resource-constrained scheduling of DSP applications when the number of processors is fixed and the objective is to obtain a schedule with the minimum iteration period [80]. Wolf et al studied combinations of various loop transformation techniques (such as fission, fusion, interchanging, etc) and presented a model for estimating total machine cycle time, taking into account software pipelining, register pressure and loop overhead [95]. Passos and Sha proved that in the multi-dimensional case (e.g., nested loops), full-parallelism can always be achieved by using multi-dimensional retiming [59]. Modulo scheduling by Ramanujam [64] is a technique for exploiting instruction level parallelism (ILP) in the loop. It can result in high performance code but increased register requirements [48]. Rau and Eichenberger have done research on optimum modulo schedules, taking into consideration the minimum register requirement. They consider not only the data flow, but also the control flow of the program [27, 68]. These techniques only focus on ALU operations and can yield an efficient ALU schedule by exploiting instruction level parallelism. However, their performance may vastly degrade when memory latency is counted in the schedule. Our ap-
approach adopts loop pipelining to optimize the ALU schedule and integrate the prefetching idea to compensate the data fetching latency.

To exploit the data locality existed in loops, one good method is to group the elemental computation points. Related work can be found in the loop tiling. Loop tiling is mainly applied to the distributed system to reduce communication time. Wolf and Lam proposed a loop transformation technique for maximizing parallelism or data locality [94]. Boulet et al introduced a criterion for defining optimal tiling in a scalable environment. In his method, an optimal tile shape can be determined and the tile size obtained from the resources constraints [61]. Another interesting result was produced by Chame and Moon. They proposed a new tile selection algorithm for eliminating self interference and simultaneously minimizing capacity and cross-interference misses. Their experimental results show that the algorithm consistently finds tiles that yield lower miss rates [12]. However, the traditional tiling techniques only concentrate on reducing communication cost. They do not consider how to best balance the computation and communication schedules. There is no detailed schedule consideration in their algorithms.

Loop partitioning employs the idea of grouping the elemental computation points. It improves the traditional loop tiling in that it carefully examine the relation between the ALU and memory operation schedules when generating the detailed schedules. Loop partitioning partitions the entire iteration space into small execution blocks called partition. Depending on the architecture configuration and loop patterns, the best partition is selected to hide the memory latency to the largest extent.

We first deal with the architecture in which the first level memory has limited size. A consequence of the memory size constraint is that we cannot prepare an
arbitrary amount of data in the first level memory. In such case, a perfect balance between computation and memory accessing latency is not always possible. Thus, how to find a partition which can derive a close balanced schedule is our major concern. An algorithm is proposed to find such partition and the corresponding detailed schedule.

A memory hierarchy with three or more levels or memory is widely adopted in the contemporary system architecture. The general role of the second level cache is to reduce the cache miss penalty for the first level cache. Instead of loading data from the rather slow main memory, the data can be fetched from the relatively faster second level cache when the first level cache miss occurred. However for the loop intensive application, we observed that the second level memory cannot be fully utilized due to the simple cache replace algorithm. Therefore, a multiple level partition scheme is proposed to solve this problem.

Two different types of data are generally involved in loop calculation, i.e., intermediate data\(^1\) and initial data\(^2\) Without considering the initial data, the balance of memory loading and computation will be destroyed, which means a not so efficient schedule as thought. We propose an algorithm to specially deal with the application with initial data. The relation between the initial data for the consecutive partitions is deliberated.

An important problem related to the loop performance is the treatment of multiple consecutive loop nests. With these adjacent loop nests, data locality may include not only those in each loop nest but also those between different loop nests. The previous mentioned methods can well handle the first kind of data

\(^1\)Intermediate data are consumed and modified during the computation, they can appear as both left and right operands

\(^2\)Initial data can only be consumed during the computation, they can only serve as the right operands
locality, but is not aware of the second. We propose an algorithm to partition multiple loop nests at the same time such that all the possible data locality is exploited to the largest extent.

To make our methods applicable to more architecture models, we extend them to take into consideration the memory hierarchy in which the memory is in low associativity. With low associativity, simple grouping the iterations could bring more conflict misses to the cache which may compromise the gain from the data locality exploitation. To prevent the loss due to the conflict misses, we integrated the data padding idea into our partition algorithm to overcome the problem of low associative cache. An algorithm is introduced to find the pad and apply the pad to data arrays such that the conflict misses can be eliminated.

1.2 Clustered architecture

Many high performance processors are designed with wide issue widths to exploit extensively the instruction level parallelism (ILP). With the increased capability of overlapping operations comes the increased need to supply register bandwidth. Eventually the access time to and from a central register file becomes the bottleneck of the cycle time of the processor. It is very difficult to solve many problems associated with a large centralized register file, such as long access time, excess silicon area for address decoders, complicated bypassing logic, etc. Hence, a natural solution is to deploy a clustered architecture.

By distributing the registers, function units and caches into different clusters, register files can be located near their data consumers and producers, such that both access time and power are saved. The performance can also be greatly improved due to the reduced clock cycle time. Clustered architectures are gaining
popularity, e.g., the architecture of Texas Instruments’s TMS320C6000, Equator’s MAP1000, Analog’s TigerShare and HP Lx all adopt multiple clusters.

However, a big concern in clustered architecture is scheduling. In the clustered architecture, the inter-cluster MOVE operation is required whenever a non-local (not in the current cluster) variable is accessed. These inter-cluster data transfers may lead to undesirable increase in schedule latency if the previous centralized architecture scheduling algorithm is directly applied. An inefficient schedule often ruins the benefit obtained from the reduced processor cycle time. Therefore, the efficient scheduling technique which can cope with multiple clusters is very crucial to the success of such architecture.

In the context of multimedia and digital signal processing (DSP) applications, there exist large number of loops. Loop schedule latency can significantly influence the whole system performance. Thus high quality software pipelining algorithms targeting VLIW clustered architecture are dispensable for such applications. In this thesis, we propose a rotation scheduling algorithm to optimize the loop performance in the clustered architecture, where both register files and data caches are distributed among clusters.

1.3 Multi-bank memory system

To meet the ever increasing demands for higher performance and lower power on embedded systems, domain specific processors with sophisticated architectures are being designed and deployed to better match target applications. One such architecture, often referred to as a non-orthogonal architecture [17], is characterized by irregular data paths comprising of a heterogeneous register set and multiple memory banks. A number of embedded DSP processors, e.g., Analog Device
ADSP2100, Motorola DSP56000 and NEC uPd77016, are based on this architecture. Compared to a large, centralized homogeneous register file, a heterogeneous (in terms of instruction usage) register set organized in a distributed fashion can reduce both access time and power, as well as simplify the control logic and chip layout design [23]. The use of multi-bank memory can potentially improve the exploitation of instruction level parallelism, which in turn may decrease memory access time and energy consumption compared to a single large memory [8].

Harvesting the benefits provided by the non-orthogonal architecture hinges on effective compiler support. Parallel operations afforded by multi-bank memory give rise to the problem of how to maximally utilize the instruction level parallelism. Similarly, heterogeneous register sets increase the difficulty in deciding which register set to use for a certain instruction. A good compiler should consider the heterogeneous register set assignment and instruction scheduling together, since the two are closely related [99]. It is not difficult to see that compilation techniques for general purpose architectures are not adequate to handle the irregularity in the architecture. In this thesis, we focus on two critical steps in the compilation process, i.e., partitioning variables (or data) among the memory banks, and scheduling memory access operations. The decisions made in the two steps can have a significant impact on the overall program code size, execution time and energy consumption.

It is well known that memory components in embedded systems, particularly those for data intensive applications, are a major power consumer [11]. To help ease the energy demands by memory, advanced memory modules are designed to operate in different modes, e.g., active, idle and sleep [52, 65], which have different operating current. The exploitation of different operating modes together with
multiple memory banks further complicate the problem of variable partitioning and memory operation scheduling. On top of this, performance requirement often conflicts with energy saving.

In this thesis, we present our approach to variable partitioning and memory access operation scheduling in the presence of multi-bank memory and multiple memory operating modes for maximizing energy saving without sacrificing performance.

1.4 Contribution

The contributions made by this research effort are listed as following.

1. We extend the traditional partition scheduling algorithm by considering the first level memory constraint [84, 87]. Given a memory size constraint, a closed balanced partition schedule is derived. Moreover, the calculation of the first level memory consumption is provided.

2. We present the algorithm [85, 86] to generate the partition schedule under the memory hierarchy with multiple levels of memory. In this algorithm, the execution of iteration space is partitioned in a multiple level way to take full advantage of the memory between the first level memory and main memory. The detailed schedule for each partition level is also presented.

3. An algorithm [89, 91], which studies the influence of both initial and intermediate data, is proposed. We introduce the condition for partition size in order to minimize the initial data transfer between different levels of memory. A balanced partition schedule can be obtained for both computation-intensive and data intensive applications.
4. We consider the partition for multiple consecutive loop nests [90]. An algorithm is proposed to partition all the loop nests at the same time to take advantage of the data locality between loop nests. Two different loop nest patterns are studied. The partition schedule can be in the format of either fine-grained or coarse-grained execution order. The traditional loop fusion can be regarded as a special case of this algorithm.

5. Data padding is integrated into our algorithm such that they can fit in the architecture with low-associativity cache [88].

6. We propose a rotation scheduling algorithm [92] to optimize the loop performance in the clustered architecture, where both register files and data caches are distributed among clusters. A global MOVE operation scheduling method is used to gather all the global scheduling and resource information. Furthermore, through judiciously scheduling the traffic between register and local cache, costly inter-cache move operation is eliminated to improve the schedule length.

7. We present an approach [81, 82] for variable partitioning and instruction scheduling to maximally exploit the benefits provided by multi-bank memory architecture. Our approach is built on a novel graph model which strives to capture both performance and power demands. The performance is improved through exploiting more memory access parallelism. The energy saving is achieved by increasing the serialism, thus enable the memory module stay longer in low power operating mode.
1.5 Organization of the thesis

In the next chapter, the background knowledge is introduced to help the reader understand the technique presented in this thesis. These background knowledge includes uniformization, scheduling, retiming, data flow graph, etc. They are the foundation of the entire thesis.

Chapter 3 will discuss how to apply loop partition on the architecture with limited first level memory size. Chapter 4 talks about the two level partitioning technique, which can be easily extended to multiple level partitioning for multiple levels of memory. Chapter 5 is related to loop partitioning, taking into consideration of both the initial data and intermediate data. Chapter 6 discusses the multiple loop nests scheduling technique which can deal with multiple consecutive loop nests in applications and presents the data padding technique and its application in partitioning technique. Chapter 7 shows the scheduling framework which can be applied to clustered architecture with distributed cache. Chapter 8 presents the novel graph model to capture both parallelism serialism in multi-bank memory system and the scheduling mechanism devised on the basis of this graph model. At the end, Chapter 9 concludes the thesis.
CHAPTER 2

BACKGROUND

In this chapter, we review some background knowledge which lays the foundation for the entire thesis.

2.1 Uniformization

It is mentioned in the last chapter that the loop partitioning technique is mainly applied on the multi-dimensional loop with uniform data dependence. Such data dependence can be found in a lot of applications. Moreover, more general affine and linear data index can be transformed into the uniform index.

For a linear parameterized recurrence equation in the form of

\[ U(z) = f(V(I(z - \theta_z), ...) \quad with \quad \theta_z = A \cdot z + B \cdot p + C \]

(where \( z \) is the loop index vector, \( p \) is the size parameter vector of the equation, \( A, B, C \) are all integer constant matrices), it is non-uniform if the number of different dependences depends on size parameter \( p \). A system of linear recurrence equations is a finite set of equations in the above form. It has been proved that any system of linear recurrence equations can be transformed into the normal form in which all the computation equations are fully indexed, and have the same index dimension. In the normal form, all the equations can be classified into input
equations, computation equations and output equations. The uniformization of computation equations is try to find a finite set of integral vectors (independent on \( p \)), \( A_1, A_2, \ldots, A_t \), such that any \( \vartheta_z \) can be expressed as a non-negative integral combination of these vectors, i.e., \( \vartheta_z = \sum_{j=1}^{t} \alpha_j \cdot A_j \) with \( \alpha_j \in \mathbb{N} \). With a set of such vectors, the linear recurrence can be transformed into a set of uniform recurrences. The detailed analysis of uniformization can be found in [24] and [63].

The techniques in this thesis are majorly used to optimize the schedule of the computation recurrences. Another work done by A. Agarwal et al [1] and F. Rastello et al [67] concentrates on the input and output equations. They use data footprint to capture the combined set of data accesses made by references within each uniformly intersecting class. A data partition size is selected to minimize the communication in multiprocessors with caches.

2.2 Loop nest representation

In a uniform nested loop, an iteration is the execution of the loop body once. It can be represented by a graph called multi-dimensional data flow graph (MDFG).

**Definition 1** A multi-dimensional data flow graph (MDFG) \( G = (V, E, d) \) is an edge-weighted directed graph, where \( V \) is the set of computation nodes, \( E \subseteq V \times V \) is the set of dependence edges, and \( d \) is a function from \( E \) to \( \mathbb{Z}^n \) representing the multi-dimensional data dependence (delay vector) between two nodes, where \( n \) is the depth of the nested loop.

From the above definition, each node in an MDFG denotes a computation. Represented by an MDFG, an iteration can also be thought as the execution of all
nodes in $V$ one time. Iterations are identified by a vector $i$, equivalent to a multi-dimensional loop index, starting from $(0, 0, \ldots, 0)$. An edge with delay $(0, 0, \ldots, 0)$ represents an intra-iteration dependency, and an edge with non-zero delay ($d(e)$) represents an inter-iteration dependency. This means that the execution of the current iteration will use data computed $d(e)$ iterations before.

\begin{align*}
& DO 10 n = 1, N \\
& DO 20 m = 1, M \\
& y(n, m) = x(n, m) + c(0, 1) y(n, m - 1) + \\
& c(0, 2) y(n, m - 2) + c(1, 0) y(n - 1, m) + \\
& c(1, 1) y(n - 1, m - 1) + c(1, 2) y(n - 1, m - 2) + \\
& c(2, 0) y(n - 2, m) + c(2, 1) y(n - 2, m - 1) + \\
& c(2, 2) y(n - 2, m - 2) \\
& 20 \text{ CONTINUE} \\
& 10 \text{ CONTINUE}
\end{align*}

(a) FORTRAN code of an IIR filter  
(b) MDFG representation

Figure 2.1. The MDFG representation of IIR filter

It is worthwhile to note that MDFG can be thought of as the computationally finer-grained description of data dependence than the Data Dependence Graph (DDG) or Statement Dependence Graph (SDG). The node in MDFG represents an ALU computation, thereby consumes one basic ALU computation time unit. On the contrary, a node in DDG or SDG represents a statement, which may consume uncertain number of ALU computation time units depending on the complexity of the statement. Therefore, it is much easier to use MDFG instead of DDG or SDG.
when scheduling the computation. Furthermore, Most DSP filters, such as IIR, two dimensional filter, etc, can directly modeled by MDFG. Consider the example in Figure 2.2. The FORTRAN code derived from the IIR filter equation is shown in Figure 2.1(a). An equivalent MDFG is presented in Figure 2.1(b). The graph nodes represent two kinds of operations: nodes denoted by an 'A' followed by an integer are additions, while nodes labeled 'M' followed by an integer represent multiplications. Notice that dependence vectors are represented by pairs $(d_x, d_y)$, where $d_x$ corresponds to the dependence distance in the Cartesian axis representing the outermost loop and $d_y$ corresponds to the innermost loop.

Iterations are identified by a vector $\vec{i}$, equivalent to the nested loop index and starting from $(0, 0, \ldots, 0)$. An edge with delay $(0, 0, \ldots, 0)$ represents an intra-iteration dependence, and an edge with non-zero delay $(d(e))$ represents an inter-iteration dependence. We will use delay dependence to represent the dependence among different iterations, i.e., the non-zero edge’s weight. The execution of the entire loop will scan over all loop indices. All iterations constitute the iteration space.

The execution of the entire loop will scan over all loop indices. It can be regarded as the execution of all iterations with different index vectors. The iteration space can be described by the cell dependence graph.

**Definition 2** The **cell dependence graph (CDG)** of an MDFG is a directed acyclic graph, showing the dependencies between different iterations. A **computational cell** is the CDG node that represents a copy of the MDFG and is equivalent to one iteration. The **dependence delay set** $D$ is the set containing all non-zero dependence vectors in CDG.
Therefore, an iteration can be seen as one node in the iteration space. A schedule vector of a CDG can be regarded as the normal vector for a set of parallel hyperplanes, of which the iterations in the same hyperplane will be executed in sequence. For example, a schedule vector of (1,0) means the row-wise execution sequence. An MDFG is said to be realizable if we can find an execution sequence for each node. For example, if there exists a delay vector (1,1) from node 1 to node 2, and (2,1) from node 2 to node 1, the computation of node 1 and node 2 depend on each other, and no execution sequence which can satisfy the delay dependence exists. To be realizable, an MDFG must satisfy two criteria: there must exist a schedule vector $s$ for the CDG with respect to $G$ such that the inner product $s \cdot d(e) \geq 0$ for any $e \in E$; and the CDG must be acyclic.

2.3 Multi-dimensional retiming

A multi-dimensional retiming on an MDFG $G$ is a function that redistributes the MDFG nodes in the iteration space. A new MDFG $G_r$ is created, such that each iteration still has one execution of each node in $G$. The retiming vector $r(u)$ of a node $u \in G$ represents the offset between the original iteration containing $u$, and the one after retiming. The delay vectors change accordingly to preserve dependencies, i.e., $r(u)$ represents delay components pushed into the edges $u \rightarrow v$, and subtracted from the edges $w \rightarrow u$, where $u, v, w \in G$. After retiming, the execution of node $u$ in iteration $i$ is moved to the iteration $i - r(u)$.

After retiming, a prologue and epilogue are generated. A prologue is the set of instructions that must be executed to provide the initial data for the iterative process. An epilogue is a complementary set of instructions which needs to be executed to complete the process. Considering that the entire problem consists of
a large number of iterations, the time required to run the prologure and epilogue are negligible.

Figure 2.2 shows the MDFG extracted from a Wave Digital Filter (WDF) and the equivalent code. If we apply a retiming vector \((0, 1)\) on node \(D\), the result is shown in Figure 2.3.

![MDFG and equivalent Fortran code](image)

Figure 2.2. (a) MDFG extracted from a Wave Digital Filter (b) equivalent Fortran code

2.4 Resource constraint scheduling

In this section, we will introduce two scheduling algorithms which take resource constraint into consideration.
2.4.1 Multi-dimensional rotation scheduling algorithm

Multi-dimensional rotation scheduling algorithm [58] is used to get a static compact schedule for one iteration. The inputs to the rotation scheduling algorithm are an MDFG and its corresponding initial schedule, which can be obtained by running the list scheduling algorithm. Rotation scheduling reduces the schedule length (the number of control steps needed to execute one iteration of the schedule) of the initial schedule by exploiting the concurrency across iterations. It accomplishes this by shifting the scope of the iteration in the initial schedule down so that nodes from different iterations appear in the same iteration scope. Intuitively speaking, this procedure is analogous to rotating tasks from the top of each iteration down to the bottom. Furthermore, this procedure is equivalent to retiming those tasks (nodes in the MDFG) in which one delay will be deleted from all incoming edges and added to all outgoing edges, resulting in an intermediate retimed graph. Once the parallelism is revealed, the algorithm reassigns the
rotated nodes to positions so that the schedule is shorter.

Figure 2.4. (a) Initial MDFG (b) After retiming r(D) = (1,0) (c) After retiming r(A) = (1,0)

In this technique, we first get the initial schedule by list scheduling, and find the down-rotatable node set, i.e., a set in which no node has a zero delay vector coming from any node not in this set, so that the node in this set can be rotated down through retiming. At each rotation step, we then rotate some nodes in this set down and try to push them to their earliest control step according to the precedence constraints and resource availability. This can be implemented by selecting a particular retiming vector and using it to do retiming on the previous MDFG. Thus we can get a shorter schedule after each control step. This step is repeated until the shortest schedule under resource constraints is achieved.

Consider the example in Figure 2.4 (a). Let nodes A and D represent multiplication operations, while nodes B and C represent addition operations. The initial schedule obtained from using list scheduling has length 4, as seen in Figure 2.5
Figure 2.5. (a) Initial schedule (b) The schedule after rotated node D (c) The schedule after rotated node A

(a). The set \{D\} is a down-rotatable set. Retiming node D using the retiming function \(r(D) = (1,0)\) (see Figure 2.4 (b)) results in the node being down-rotated and tentatively pushed to its earliest control step, which is control step 3. The result is seen in Figure 2.5 (b). At this time, the node set \{A\} is a down-rotatable set. Applying the retiming function \(r(A) = (1,0)\), as seen in Figure 2.4 (c), we see that node A can be rotated down and pushed into its earliest control step 4. This result is seen in Figure 2.5 (c). Thus we can get a schedule with a length of only two control steps.

2.4.2 Modulo scheduling

Like rotation scheduling, modulo scheduling [25, 26, 33–35, 93] is a resource constrained scheduling framework which strives to pipeline a loop by overlapping nodes from different iterations. Unlike rotation scheduling, the modulo scheduling technique requires the expected schedule length, called the *initiation interval*, to be selected before scheduling is attempted. The candidate initiation interval is normally set to the *minimum initiation interval* (MII) which is a lower bound on the smallest possible schedule length. Such a lower bound is categorized by
two types, resource-constrained and recurrence-constrained minimum initiation interval. The former is obtained from the resource usage requirements of the computation, while the later is actually a lower bound of the iteration period which is defined as following:

**Definition 3** The iteration bound of a data-flow graph $G$ is the maximum time to delay ratio of all cycles in $G$. This can be represented by the equation $B(G) = \max_{l} \left\{ \frac{T(l)}{D(l)} \right\}$, where $T(l)$ and $D(l)$ is the summation of all computation times and the summation of delays, respectively, in a cycle $l$.

The calculation of recurrence-constrained MII can be found in the literature. The MII is chosen as the maximum of the resource-constrained and recurrence-constrained MII. Once this value is computed, modulo scheduling use the MII as the template for a node to be repeated every MII interval. For example, assuming that the MII is 3, if node $A$ is first scheduled at time 0, one should expect that this node will be repeated at time steps 3, 6, 9 and so on. Since the schedule is periodic in nature, we can use a table, called the *modulo reservation table*, to store only one iteration of the schedule [38]. Note that modulo scheduling uses the modulo operation ($\%$) to check if a certain time step has been occupied by any node, e.g. to verify that a node was scheduled at time 5, we compute 5 modulo its initiation interval and use this to check the modulo reservation table. For example, node $A$ is not scheduled at time step 5 since $5 \% 3 \neq 0$.

Most modulo scheduling variants rely on the list scheduling framework to select nodes to be scheduled. In order to deal with the underlying inter-iteration dependencies, modulo scheduling checks if the inequality $(S(v) + d(e)) \times II - S(U) \geq t(u)$ holds, where $S(u)$ and $S(v)$ are the starting time of each node and $t(u)$ is the computation time of node $u$. 

22
2.4.3 The comparison between two scheduling algorithms

Rotation scheduling makes use of non-resource constrained scheduling, namely the retiming algorithm to parallelize nodes. It iteratively improves a resulting schedule while obtaining an improved intermediate retimed graph. Although the general concept and goal of rotation and modulo scheduling are very similar, we believe that the iterative approach in rotation scheduling should be better than the one pass algorithm (based on list scheduling) of the modulo scheduling framework.

Without a feasible initiation interval, modulo scheduling will not be able to produce a valid schedule table. However, we can find a feasible initiation interval by choosing an upper bound and using a binary search to find the resulting schedule table. Furthermore, since modulo scheduling relies on the greedy list scheduling framework, even when the initiation interval is feasible, modulo scheduling may not be able to find the scheduling result to fit into it. In other words, because of the greediness of list scheduling, modulo scheduling may pick a wrong node to be scheduling first. This scheduled node may prevent other nodes from being scheduled within the given initiation interval. Some experimental result of two algorithms can be find in the literature [78].
Prefetching data into the memory nearest to the CPU can effectively hide the long memory latency. In our model, a two level memory hierarchy exists. A remote memory access will take much more time than a local memory access. We also assume a processor consists of multiple ALU and memory units. The ALUs are for doing the computations. The memory units are for performing operations to prefetch data from the remote memory to the local memory. Given a uniform nested loop, our goal is to find a schedule with the minimal execution time (e.g., the minimal average iteration schedule length, since the number of iterations in the nested loop is constant). This goal can be accomplished by overlaying the prefetching operations as much as possible with the ALU operations, so that the ALU can always keep busy without waiting for the operands. In order to implement prefetching, we need to find the best balance between the ALU operations and the prefetching operations under the local memory size constraint. A consequence of the local memory constraint is that we cannot prepare an arbitrary amount of data in the local memory before the program. This chapter proposes techniques that prefetch information into the local memory at run-time based on compile time information and the partition iteration space, thereby circumventing the local memory size constraint problem.
We will restrict our study to nested loops with uniform data dependencies. Even if most loop nests have affine dependencies, the study of uniform loop nests is justified by the fact that an affine loop nest can always be transformed into an uniform loop nest (as discussed in uniformization in Chapter 2).

This chapter will combine two aspects, instruction level parallelism and reducing the memory access latency, to balance and minimize the overall schedule for uniform dependency loops. This new approach will make extensive use of compile time information about the usage pattern of the information produced in the loop. This information, together with an aggressive memory partitioning scheme, produces a reduction in memory latency unrealizable by existing means. New “instructions” are introduced in this approach to ensure that memory replacement will follow the pattern determined during compilation.

In a standard system with 4 ALU units and 4 memory units, and making the assumption that memory accesses take 10 cycles, our algorithm presented in this chapter can obtain an average schedule length of 4.01 CPU clock cycles. Using the traditional list scheduling algorithm, the average schedule length will become 23 CPU clock cycles when memory access time is taken into consideration. Using the rotation technique to explore more instruction level parallelism, the schedule length is 21 CPU clock cycles because of the long memory prefetch time which dominates the execution time. Finally, using the PBS algorithm presented in [14] which takes into account the balance between ALU computation and memory access time, a much better performance can be obtained, but the average schedule length is still 7 CPU clock cycles. Therefore, our algorithm can make a large improvement.

Our new algorithm exceeds the performance of existing algorithms by optimiz-
ing both the ALU and memory schedules. In our algorithm, the ALU part can be scheduled by any loop scheduling algorithm. We optimize the ALU schedule by using the multidimensional rotational scheduling algorithm because it has been shown to achieve an optimal ALU schedule [58]. This chapter presents a method of deciding the best partition which achieves a balanced schedule, as well as deriving the theory to calculate the total memory requirement for a certain partition. Experiments show the improvement.

3.1 Basic Framework

In this section, we review the idea of partitioning iteration space. The architecture model and the framework of the algorithm are also illustrated in this section.

3.1.1 Partitioning the iteration space

Regular execution of nested loops proceeds in either row-wise or column-wise manner until the end of the row or column is reached. However, this mode of execution does not take full advantage of either the locality of reference present or the available parallelism, since dependencies have both horizontal and vertical components. The execution of such structures would be made more efficient by dividing the iteration space into regions that better exploit spatial locality called partitions. Once the iteration space is divided into partitions, the execution proceeds in partition order. That is to say, each partition is executed in turn from left to right. Within each partition, iterations are executed in row-wise order. At the end of a row of partitions, we move up to the next row and continue from the far left in the same manner.
The key to our memory management technique is the way in which the data produced in each partition are handled. A preliminary step in making that decision is determining the amount of data that needs to be handled. For this purpose, two important pieces of information are the shape and size of the partition. To decide a partition shape, we use partition vectors $P_X$ and $P_y$ to represent two boundaries of a partition. Without loss of generality, the angle between $P_x$ and $P_y$ is less than $180^\circ$, and $P_x$ is clockwise to $P_y$. Due to the dependencies in the CDG, these two vectors can not be chosen arbitrarily. The following property give the conditions of a legal partition shape.

**Lemma 1** For vectors $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, define the cross product $p_1 \times p_2 = x_1y_2 - x_2y_1$. Partition vectors $P_x$ and $P_y$ are legal iff $d_e \times P_x \leq 0$ and $d_e \times P_y \geq 0$ for each delay vector $d_e$ in the CDG.

**Proof:** Since execution proceeds in partition order, dependency cycles between partitions would lead to an unrealizable partition execution sequence. The constraints stated above guarantee that dependency vectors can not cross the lower and left boundaries of a partition, thus guaranteeing the absence of cyclic delay dependencies. \hfill \Box

The *counterclockwise region* of a vector $P$ is the region found by sweeping the plane in a counterclockwise direction, starting at $P$ and ending when the sweeping line becomes vertical. The definition of the clockwise region is similar. Given a set of dependence edges $d_1, d_2, \ldots, d_n$, we can find two extreme vectors. One is the left-most vector, in relation to which all vectors are in the counterclockwise region. The other is the right-most vector, in relation to which all vectors are in the clockwise region. It is obvious that the left-most and right-most vectors satisfy Lemma 1, and thus they are a pair of legal partition vectors.
Because nested loops should follow lexicographic order, the vector $s = (0, 1)$ is always a legal scheduling vector. Thus the positive x-axis is always a legal partition vector if we choose $(0, 1)$ as the base retiming vector. We choose the left-most vector from the given dependence vectors, and use the normalized left-most vector as our other partition vector. The partition shape is then decided by these two vectors.

3.1.2 Memory unit operation

As discussed above, the entire iteration space will be divided into partitions, and the execution sequence is determined by these partitions. Assume that the partition in which the loop is executing is the current partition. Relative to this current partition, there are different kinds of partitions and each kind of partition corresponds to a different kind of memory unit operation.

**Definition 4** *The next partition is the partition which is adjacent to the current partition and lies to the right of the current partition along the x-axis. The second next partition is adjacent to and lies on the right of the next partition. The definitions of third next partition, fourth next partition, etc., are similar. The other partitions are those partitions which are in a different row of partitions from the current one.*

As discussed in Section 2, a delay dependency going from one partition to another means that the execution of the destination partition will use some data computed during the execution of the starting partition. Depending on which kind of partition the endpoint of the delay dependency is located in, either a *keep* or *prefetch* operation will be used.
**Definition 5** Delay dependencies that go into the next $m^{th}$ partition use `keep.m` memory operations to keep the corresponding data in the first level memory for $m$ partitions. Delay dependencies that go into the other partitions use `prefetch` memory operation to fetch data in advance.

![Diagram](image)

(a) Different kind of delay dependencies

(b) Three different regions

Figure 3.1. Different kinds of memory operations and its corresponding regions

For instance, for the delay dependencies in Figure 3.1(a), $d_1$ needs a `keep.1` operation; $d_2$ needs a `keep.2` operation; $d_3$ needs a `keep.3` operation; and $d_4$ and $d_5$ need prefetch operations.

Given a delay vector, the current partition can be divided into three different regions. Let $(x, y)$ be an iteration within the current partition and translate the delay vector so that it begins at this point. If the vector terminates in a partition in the same row as the current one, $(x, y)$ lies in the `keep area`. If it terminates in
a partition in a different row, \((x, y)\) lies in the *prefetch area*. Otherwise the delay vector terminates within the current partition and \((x, y)\) lies in the *inter area*. For example, in the Figure 2(b), the delay vector \(d\) determines these three regions. The region ABFE can be treated as the prefetch area, region GFCH as the keep area, while EGHD can be treated as the inter area.

The reasons we have the above different kinds of memory unit operations are based on two observations. First, in the real loop, the delay dependency is not long enough to make the value of \(m\) in *keep\_m* too large. This implies that the data kept in the first level memory must be used during the execution of a partition in the near future. Second, fetching data from the second level memory takes much more time than just keeping data in the first level memory according to our memory arrangement.

It is possible that several different delay dependencies start from the same MDFG node in the same iteration, so that we can spare some memory operations depending on the end point of the delay dependency.

**Property 1** *Those delay dependencies with the same starting MDFG node in the same iteration and different ending nodes can be placed into one of three classes:*

* a) If they end at nodes in the same partition, we can merge their memory unit operations.

* b) If they end at nodes in different next partitions in the same row as the current partition, keep operations are needed. We use the longest keep operation to represent all of them.*

* c) In any other situation, we cannot merge memory unit operations corresponding to these delays.*
3.1.3 Architecture model

Our technique is designed for use in a system containing a processor with multiple ALUs and memory units. Associated with the processor is a local memory of limited size. Accessing this local memory is fast. A much larger memory, remote memory, also exists in the system. However, accessing it is significantly slower. Our technique is to load data into local memory before its explicit use so that the overall cost of accessing the remote memory can be minimized. Therefore, overlapping the ALU computation and the memory accesses will lead to a shorter overall execution time. The goal of our algorithm is to overlap the memory access and program execution as much as possible, while satisfying the first level memory size constraint at the same time.

Our scheme is a software-based method, in which some special memory instructions are added to the code at compile time. When the processor encounters these instructions during program execution, it will pass them to the special hardware memory unit which processes them. The memory unit is in charge of putting data in the first level memory before an executing partition needs to reference them. Two types of memory instructions, prefetch and keep, are supported by memory units. The keep instruction keeps the data in the first level memory for the use during a later partition’s execution. Depending on the partition size and delay dependencies, the data will need to be kept in the first level memory for different amounts of time. The advantage of using keep is that it can spare the time wasted for unnecessary data swapping, so as to get a better performance schedule.

When arranging data in memory, we can allocate memory into several regions for different operations to store data. Pointers to each of these regions will be kept in different circular lists, one each for the keep and prefetch data. Thus,
when the execution reaches the next partition, we need only move the list element one step forward rather than performing a large number of data swaps. Assume, for example, that the results produced in a partition belong to one of three classes: used in this partition, used one partition in the future or used two partitions in the future. During the execution of the current partition, we can arrange data in memory as seen in Figure 3.2. We have two circular lists: \{p1,p3\} and \{p2,p4,p5\}. When we move to the next partition, we only move the list element forward one step, thus getting the lists \{p3, p1\} and \{p4, p5, p2\}. We still obey the same rule to store different kinds of data.

This architecture model can be found in real systems such as embedded systems and DSP processors in which multiple functional units and small local memories share a common communication bus and a large set of data stored in the off-chip memory. It is important to note that our local memory cannot be regarded as a pure set-associative cache, because important issues such as cache consistency and
cache conflict are not considered here. In other words, the local memory in our technique can be thought as a fully associative cache with some simple intelligence such as the ability to group the different kinds of data.

3.1.4 Framework of the algorithm

**Algorithm 1** Find the partition size which can lead to the schedule with minimal average schedule length

**Input:** The MDFG after rotation; Number of ALU units and Memory Units; The first level memory size constraint

**Output:** the partition size

1. Do the rotation to get the ALU schedule.
2. Get the optimal partition size $V_x \times V_y$ under no memory constraint. //see section 4.3, theorem 12
3. Calculate the memory requirement. // see section 4.5, theorem 13
   If it satisfies the first level memory constraint
   then output the partition size
   return
4. else, calculate the memory requirement when $f_x = 1, f_y = F_y$. //see section 4.2 and section 2.3
5. If this size is larger than the first level memory constraint
   then print(“no suitable partition exists”)
   stop
6. For each delay vector $d = (d_x, d_y)$, calculate the projection on the x-axis along $P_y$ direction, $ld = d_x - d_y \times \frac{P_y.x}{P_y.y}$.
7. Let $f_x = ld$, and for each $ld$, calculate the memory requirement. //see section 4.4, theorem 13
8. Find the interval whose left endpoint satisfy the memory constraint, but whose right endpoint doesn’t.
9. Repeatedly increase $f_x$ within this interval until it reaches the memory constraint.
10. Output the partition size.
In Algorithm 1, we divide the partition schedule into two parts: the ALU part and the memory part. In the ALU part of the schedule, we use the \textit{multi-dimensional rotation scheduling algorithm} to create the schedule for one iteration, then duplicate this one iteration according to the partition size to obtain the final ALU schedule.

The memory part will be executed by the memory unit at the same time as the ALU part. It gives the global schedule for all memory operations which are executed in the current partition. These operations will have all data needed by the next partition’s execution ready in the first level partition.

3.2 Theoretical foundation

The main theme of this chapter is the division of an iteration space into distinct partitions that can be effectively used in the execution of loop structures. Due to the nature of loop structures, in a two dimensional representation of the iteration space, all inter-iteration dependencies can be reduced to vectors that consist of non-negative \( y \) components. In this context, each partition considered will be represented by a parallelogram-shaped region \( ABCD \), with \( AB \parallel CD \) and \( AD \parallel BC \), in which all corners fall on a point in the iteration space. Here assume \( AB \) is the lower boundary and \( AD \) the left boundary, as in Figure 3.3. Partitions are then defined by four parameters which describe its lower and left boundaries:

1. A two dimensional vector \( P_x = (P_x.x, P_x.y) \) that determines the orientation of the lower boundary of the parallelogram. In our approach its value is always \((1,0)\).

2. A constant \( f_x = \frac{|AB|}{|P_x|} \), which is the length of the partition’s lower boundary.
3. A two dimensional vector $P_y = (P_y.x, P_y.y)$ that determines the orientation of the left boundary of the parallelogram. The components of $P_y$ are relatively prime. $P_x$ and $P_y$ are the partition vectors of the partition.

4. A constant $f_y = \frac{|AD|}{|P_y|}$, which is the length of the partition’s left boundary.

![Figure 3.3. Two adjacent partitions](image)

Given the four parameters of a partition, a partition size and shape can be determined.

**Definition 6** A basic partition is a parallelogram with the following properties:

1. Two its sides are parallel to the x-axis, each with these length $f_x \times |P_x|$.

2. The other pair of its sides is parallel to $P_y$, each with length $|P_y|$.

For example, the two partitions in Figure 3.3 have been divided into 4 basic partitions.

35
Algorithm 2 Calculate the keep operations needed for a given delay vector \(d = (d_x, d_y)\) under the certain partition

**Input:** The partition vector \(P_x\) and \(P_y\), the partition size \(f_x \times f_y\)

**Output:** The number of keep operations

1. Let \(m = \lceil \frac{d_x - d_y P_{y,x}}{f_x \times P_{x,x}} \rceil\). Let partition \(p'\) lie \(m\) partitions in the future. Then \(m\) is the number of partitions that the delay vector can span along \(p_x\) direction

2. Let \(np = \lceil \frac{d_y}{P_{y,y}} \rceil\) be the number of basic partitions the delay vector can span along the \(P_y\) direction.

3. If \(m = 1\), the only partitions involved are the current partition and next partition. The number of keep-1 operations can be calculate according to the results next in this section.

4. If \(m > 1\), find the coordinates of the upper left corner of \(p'\).

5. Find the node \(n\) in the current partition that maps to the upper left corner of \(p'\) under the delay vector considered. Once \(n\) is known, the region of two different kinds of keep operation can be determined as show in Figure 5.5(b).

6. Calculate the total number of results that need to be kept in memory for use in the \((m-1)^{th}\) and \(m^{th}\) next partitions, according to the results derived next in this section.

Once the iteration space has been divided into partitions, the next step in the optimization process is to determine a repetitive pattern that the inter-iteration dependency vectors follow within these partitions. For each dependency vector \(d = (d_x, d_y)\), we can calculate the number of iterations in its keep area with Algorithm 2.

In Figures 5.5(a) and 5.5(b), the partition size is \(f_x P_x \times f_y P_y\) and the dotted lines give the boundary of each basic partition. The nodes in region JBHG will be treated with keep operations as shown in Figure 5.5(a) when \(m = 1\). In Figure 5.5(b), when \(m > 1\), a partition can be divided into two regions ABFE and EFCD. For a delay vector \(d = (d_x, d_y)\), the nodes in region ABFE will be treated with keep operations. Based on the point \(n\) this region consists of two sub-parallelograms, AJ\(nE\) and JBF\(n\). Each will map to different future partitions according to this delay vector. In Theorems 4 and 5, we determine how many
nodes there are in these keep areas.

![Diagram](image)

Figure 3.4. The division of a partition under difference cases

For ease of further calculations, we introduce the following definition.

**Definition 7** An **integral-area keep parallelogram** is a parallelogram that satisfies the following conditions:

1. A pair of its sides is parallel to the x-axis.

2. Its non-horizontal sides are either vertical (i.e., parallel to the y axis) or their slope is a rational number \( \frac{m}{n} \), with \( m, n \in \mathbb{N} \), and \( m, n \) relatively prime.

3. Its width is a multiple of the inverse of the slope's numerator, i.e., \( w = \frac{t}{m} \) for some \( t \in \mathbb{N} \).

4. One of the endpoints of its lower boundary has integer coordinates.
5. Its height is represented by a positive integer and is a multiple of the slope, i.e., \( h = l \times m \) for some \( l \in \mathbb{N} \).

As a prerequisite for calculating the number of keep operations needed for a given partition, we have the following lemma.

**Lemma 2** Let \( R \) be an integral-area keep parallelogram. The number of points with integer coordinates \( I \) in \( R \) with the exception of its right and upper boundaries is given by the formula \( I = w \times h \), where \( w \) is the width of \( R \) and \( h \) is the height of \( R \).

**Proof:** An integral-area keep parallelogram with width 1 and height \( h \), which has a point with integral coordinates at the left endpoint of the lower boundary, contains \( h \) points. This parallelogram can be divided into \( h \) sub-parallelograms each with width \( \frac{1}{h} \). It can be proven that the left boundary of each sub-parallelogram passes through exactly one integer point. Thus, the number of integer points in each sub-parallelogram is 1. Therefore, for any integral-area keep parallelogram with width \( w = n/h, n < h, n \in \mathbb{N} \) and height \( h \), the number of integer points is \( I = n = w \times h \).

For any integral-area keep parallelogram with width \( w = t/m \) and height \( h = l \times m \), assume that the point with integer coordinates is the left endpoint of the lower boundary. (The proof is similar for the case where the right point has integer coordinates). Let \( p = \lfloor t/m \rfloor \) and \( q = t \mod m \). We can first divide this parallelogram into \( l \) parts, each with the same number of integer points. Then, divide each sub-parallelogram with width \( w = t/m \) and height \( h = m \) into two parts, one parallelogram with width \( w = p \times m \) and the other with width \( w = q/m \).

As above, the number of integer points in the second part will be \( q \). The number
of integer points in the first part is $p \times m$. Thus, the total number of points is $l \times (p \times m + q) = w \times h$.

For ease of notation, we introduce the following definitions.

**Definition 8** Let $\frac{a}{n}$ be the fractional part of a real number $a$. Given a horizontal interval $[a,b)$ of width $\frac{m}{n}$, with $m$ and $n$ integers, $m \neq n$, and a horizontal displacement $\frac{l}{n}$ with $l$ an integer, define

$$\delta_i(m,n) = \begin{cases} \left\lceil \frac{m}{n} \right\rceil - 1 & \text{if } \frac{a}{n} \notin \{0\} \cup (1 - \frac{m}{n}), 1 \\ \left\lceil \frac{m}{n} \right\rceil & \text{otherwise} \end{cases}$$

which is the number of integer points in this interval $[a,b)$.

**Lemma 3** Let $R$ be an integral-area keep parallelogram. The number of points with integer coordinates in the region below and to the left of any point $(p,q)$ of integer coordinates can be calculated as

$$\sum_{i=0}^{n-1} \delta_i(a,b)$$

where $a,b$ are integers such that $a/b$ is the distance from $(p,q)$ to the left boundary of $R$.

**Proof:** The length $L$ of any interval $[x,y)$ can be expressed as the sum of an integer and fractional part, i.e., $L = \lfloor L \rfloor + \text{frac}(L)$.

It is clear that for any interval $K$ with $\text{frac}(\text{length}(K)) \neq 0$, there will be at least $\lfloor \text{length}(K) \rfloor - 1$ points with integer coordinates in the interval. At the same time, there are at most $\lfloor \text{length}(K) \rfloor$ points with integer coordinates in the interval. If the left endpoint of $K$ is within $\text{frac}(\text{length}(K))$ of the next highest integer point,
the length of interval $K'$ from just beyond this integer point to the end of the interval is still greater than $\lceil \text{length}(K) \rceil - 1$, and therefore it must contain at least that many points with integer coordinates. It cannot contain more, since in that case the original interval would contain $\lceil \text{length}(K) \rceil + 1$ points. Thus $K'$ contains exactly $\lceil \text{length}(K) \rceil - 1$ points with integer coordinates and $K$ will contain $\lceil \text{length}(K) \rceil$ such points. From the definition of $\delta_i$, it becomes clear that the formula is correct.

\[ \square \]

**Theorem 4** Given a memory partition defined by $P_x, P_y, f_x, f_y$, and a dependency vector $d = (d_x, d_y)$ with $d_x, d_y$ positive integers such that $m = \left\lceil \frac{d_x - d_y P_y}{f_x P_x} \right\rceil$, the region of the partition from which results need to be kept in memory for use in the $(m - 1)^{th}$ next partition can be divided into two disjoint regions, $R_1$ and $R_2$, with $R_1$ an integral-area keep parallelogram and $R_2$ a region in which the iterations whose results need to be kept in memory satisfy the requirements of Lemma 3.

**Proof:** As shown in Figure 5.5(b), the first step in the proof is to determine the first point in the current partition which will map to the upper left corner of a basic partition in a future partition under this vector. For notational purposes, let this point be $fp$. Thus, the target partition will be the $m^{th}$ next partition in the future. The second piece of information needed is the first basic partition within the target partition that will be mapped onto by an element from the current partition under dependency vector $d$. To determine this, let $mp = \left\lfloor \frac{d_y}{P_y} \right\rfloor + 1$. Then, the basic partition to consider within the target memory partition is the $mp^{th}$ basic partition from the base of the partition. Now, the right, top corner point $p$ of this basic partition is $p = (m \cdot P_x, mp \cdot P_y)$.

40
Once p is known, the first step of the proof is completed by letting \( fp = (p.x - d_x, p.y - d_y) \). It is obvious that all points that will map into the target partition will be found to the left of a line through \( fp \) parallel to \( P_y \). The region NR, delimited by this line together with the left, lower and top boundaries of the current memory partition is an integral-area keep parallelogram.

Considering the entire current partition made up of \( f_y \) basic partitions, all but the last \( mp - 1 \) basic partitions will have iterations that map into the future \((m - 1)^{th}\) or \(m^{th}\) next partition. Of these basic partitions, all but the last one will have the entire NR region map into the \((m - 1)^{th}\) next partition. These basic partitions form region R1. The number of iterations from R1 that need to be kept in memory can be calculated using Lemma 2. The last partition will only have those iteration to the left and below the \( fp \) point and thus constitutes region R2. The number of iterations from this region that need to be kept in memory can be calculated with the aid of Lemma 3.

For example, in Figure 5.5(b), all results within region AJnE are treated with \textit{keep}\_\textit{m} – 1 operations. This region consists of R1 (i.e., AJIG in this Figure, I is the intersection of lines GH and Bn) and R2 (i.e., GInE). Using the same rule, we can derive the theorem below.

**Theorem 5** Given a memory partition defined by \( P_x, P_y, f_x, f_y \), and a dependency vector \( d = (d_x, d_y) \) with \( d_x, d_y \) positive integers such that \( m = \lceil \frac{d_x - d_y f_y}{f_x P_x} \rceil \), the region of the partition from which results need to be kept in memory for use in the \(m^{th}\) next partition can be divided into two disjoint regions, R1 and R2, with R1 an integral-area keep parallelogram and R2 a region in which the iterations whose results need to be kept in memory satisfy the requirements of Lemma 3.
3.3 Algorithms

In this section, we present the algorithm used in obtaining balanced ALU and memory schedules.

3.3.1 Scheduling the ALU

In the ALU schedule, the multi-dimensional rotation scheduling algorithm \[58\] is used to get a static compact schedule for one iteration. The inputs to the rotation scheduling algorithm are an MDFG and its corresponding initial schedule, which can be obtained by running the list scheduling algorithm.

3.3.2 Scheduling the memory

The memory unit prefetches data into the first level memory before it is needed and operates in parallel with the ALU execution. While the ALU is doing some computation, the memory unit will fetch the data needed by the next partition from other memory levels and keep some data generated by this or previous partitions in the first level memory for later use.

Different from ALU scheduling, which is based on the scheduling per iteration, the memory scheduling arranges the memory unit operations in one partition as a whole. Since prefetch operations do not depend on the current partition’s execution, we can arrange them as early as possible in the memory schedule. Theoretically, any order of prefetching operations for different data will give us the same schedule length, since no data dependence exists for these operations. In our experiment, for convenience, we arrange the prefetching operations in a partition in order of row-wise iterations. Note that each kind of keep operation depends on the ALU computation result of the current partition. Thus, we can
only arrange it in the schedule after the corresponding ALU computation has been performed. In our algorithm, we schedule the keep operation as soon as both the computation result and the memory unit are available.

According to the definition in Section 3, the two basic vectors $P_x$ and $P_y$ decide the partition boundary, while $f_x$ and $f_y$ are the lengths of the two boundaries expressed as multiples of $P_x$ and $P_y$, respectively. To satisfy the memory constraint and get an optimal average schedule length at the same time, we must first understand how the memory requirement and average schedule length change with partition size.

The memory requirement consists of three parts, the memory locations to store the intermediate data in one partition, the memory locations to store the data for prefetch operations and the memory locations to store the data for keep operations. The delay dependencies inside the partition require some memory locations to keep the intermediate computation results for later use. To calculate this part of the memory requirement, we can get parallelograms ABCD and EFCD, as seen in both Figures 3.5(a) and 3.5(b). AB, CD and EF are parallel to the x-axis and have length $f_x - d_x + dy \frac{P_{x-x}}{P_{y-y}}$. AD, BC and EF are parallel to the vector $P_y$. If $2d_y \leq P_{y*y}*f_y$, the length of BC is $d_y \sqrt{P_{y*y}^2+P_{x*x}^2} / P_{y-y}$, as seen in Figure 3.5(a). On the other hand, if $2d_y > P_{y*y}*f_y$, the length of FC is $(P_{y*y}*f_y-d_y) \sqrt{P_{y*y}^2+P_{x*x}^2} / P_{y-y}$, as seen in Figure 3.5(b). The memory requirement can be decided by the parallelogram and the corresponding delay vector.

Lemma 6 Given a partition with size $f_x P_x \times f_y P_y$, let $d = (d_x, d_y)$ be a delay vector with $d_x - d_y \frac{P_{y-x}}{P_{y*y}} \leq f_x$.  

1. If $2d_y \leq P_{y*y}*f_y$, then the memory requirement for storing intermediate partition data is equal to the number of integer points in the parallelogram
Figure 3.5. the parallelogram decided by the delay vector $ABCD$ plus the number of integer points on the line $AH$ (see Figure 3.5(a)).

2. If $2d_y > P_y \cdot y \cdot f_y$, then the memory requirement for storing intermediate partition data is equal to the number of integer points in the parallelogram $EFCD$ plus the number of integer points on the line $EH$ (see Figure 3.5(b)).

Proof: From Figure 3.5 (a), we can easily see that all nodes except those in the parallelogram $EFCD$ will need prefetch and keep operations to satisfy this delay dependence. Their memory requirement will be considered in the memory requirement of the prefetch and keep operations. Only those nodes in parallelogram $EFCD$ have delay vectors which end at nodes in the same partition, and therefore their memory requirement should be considered here. Moreover, we can reuse the memory locations for those nodes on the line $HB$ and above the line $AB$ in parallelogram $EFBA$, because their data lifetime will not overlap with those nodes on and below the line $AH$. In conclusion, the memory requirement is the sum of the number of nodes in the parallelogram $ABCD$ and the number of nodes on the line $AH$.

Similarly, when $2d_y > P_y \cdot y \cdot f_y$, as seen in Figure 3.5(b), we reach the conclusion that the memory requirement is the sum of the number of nodes in the
parallelogram EFCD and the number of nodes on the line EH (H is the intersection of lines DB and EF).

If all delays start from different nodes, then the overall internal-partition memory requirement will be the sum of all memory requirements for each delay dependence. On the other hand, if multiple delay vectors start from the same MDFG node in the same iteration, more consideration is needed. In this situation, the memory requirement will be the union of all the parallelogram decided by the delay dependencies.

In order to determine the amount of memory needed for the memory operations, we need the following lemma.

**Lemma 7** The amount of memory needed by the memory operations is 2 locations for a prefetch operation and $m + 1$ locations for a keep$m$ operation.

*Proof:* The data that need prefetch and keep operation will last for two partitions in the first level memory, so two memory locations are needed. One is allocated for preloaded data for the current partition, the other is allocated for newly generated data for the next partition. The data that need keep operation will last for three partitions. As a result, it will need three memory locations: one for the data kept by the second previous partition, one for the data kept by the previous partition, and one for the new generated data. Following this pattern, we see that the memory requirement of keep$m$ is $m + 1$ locations. □

Knowing the memory consumption for each kind of memory operation, we also need to know the number of each kind of memory operation for a given partition size. The number of keep operations has been discussed in Section 3. The number of prefetch operations satisfies the relation below.

**Lemma 8** The number of prefetch operations for a partition of size $f_x P_x \times f_y P_y$
is $f_x$ times that of the number of prefetch operations required by a partition of size $P_x \times f_y P_y$.

**Proof:** In a partition of size $f_x P_x \times f_y P_y$, $f_x \times d_y$ elements in the top $d_y$ rows have results that will be used in future partitions, and so will be treated with prefetch operations. Therefore, the number of prefetch operations increases proportionally with $f_x$. □

The number of keep operations and the memory requirement of different kinds of keep operations have been given in Section 3 and by Lemma 8 above. The next lemma investigates the change in memory requirement when $f_x$ is decreased.

**Lemma 9** Given a partition size $f_x P_x \times f_y P_y$, let $(d_x, d_y)$ be a delay vector. When $f_x > d_x - d_y \frac{P_x}{P_y}$, the memory requirement of the keep operation for the delay dependence will not change when $f_x$ decreases. When $f_x \leq d_x - d_y \frac{P_x}{P_y}$, the memory requirement of the keep operation will decrease.

**Proof:**

If $f_x > d_x - d_y \frac{P_x}{P_y}$, this situation falls under the case 1 in Section 3. It is obvious that the memory requirement for keep operations, as well as the number of keep operations, will not change.

If $f_x \leq d_x - d_y \frac{P_x}{P_y}$, let $m = \left\lceil \frac{d_x - d_y \frac{P_x}{P_y}}{f_x \times P_x} \right\rceil$. The numbers of $keep_{-}(m - 1)$ and $keep_m$ operations can be decided by Theorems 4 and 5, respectively. Each of these two parts can be divided into two areas, $R1$ and $R2$. Assume the two areas of the $keep_{-}(m - 1)$ part are $R1$ and $R2$, while the two areas of the $keep_m$ part are $R1'$ and $R2'$.

We can obtain the memory requirement for all keep operations through the following steps.
• The number of \textit{keep-} \((m - 1)\) operations in \(R1\) is \(n_1 = \#BP \times P_y.y \times (m * f_x - d_x + \frac{d_y}{P_y.y} P_y.x)\), where \(#BP = f_y - \lceil \frac{d_y}{P_y.y} \rceil\) denotes the number of basic partitions in \(R1\).

• The number of \textit{keep-} \((m - 1)\) operations in \(R2\) is \(n_2 = \sum_{i=0}^{\lceil \frac{d_y}{P_y.y} \rceil - d_y - 1} \delta_i(a, b)\), where \(a\) and \(b\) satisfy the relation \(a/b = m f_x - d_x + \frac{d_y}{P_y.y} P_y.x\).

• The number of keep-\(m\) operations in \(R1'\) is \(n_3 = \#BP \times P_y.y \times P_y.x - n_1\).

• The number of keep-\(m\) operations in \(R2'\) is \(n_4 = (\lceil \frac{d_y}{P_y.y} \rceil - d_y - 1) \times f_x - n_2\).

• According to Lemma 7, the overall memory requirement is \(mem_{f_x} = (n_1 + n_2) \times m + (n_3 + n_4) \times (m + 1)\).

If we decrease the length of the partition’s lower boundary by one, the difference in memory requirement is \(mem_{f_x} - mem_{f_x-1} = \#BP \times f_y.y + P_y.y - dy\) \(\Box\)

From the two lemmas above and Lemma 6, we can know the change in the memory requirement when \(f_x\) is decreasing.

\textbf{Theorem 10} Given a partition of size \(f_x P_x \times f_y P_y\), let \(\text{Size}_{\text{inter}}, \text{Size}_{\text{prefetch}}\) and \(\text{Size}_{\text{keep}}\) represent the memory requirement for intermediate data, prefetch operations and keep operations, respectively. Then when \(f_x\) is reduced,

1. if \(f_x > d_x - d_y \frac{P_y.x}{P_y.y}, \forall d = (d_x, d_y)\), \(\text{Size}_{\text{inter}}\) and \(\text{Size}_{\text{prefetch}}\) will decrease while \(\text{Size}_{\text{keep}}\) will not change.

2. if \(f_x \leq d_x - d_y \frac{P_y.x}{P_y.y}, \forall d = (d_x, d_y)\), \(\text{Size}_{\text{inter}}, \text{Size}_{\text{prefetch}}\) and \(\text{Size}_{\text{keep}}\) will all decrease.

\textit{Proof:} It is obvious from the above lemmas. \(\Box\)
Once the relation between the memory requirement and change in $f_x$ is known, the next step is investigating the change in memory requirement and average schedule length when $f_y$ changes. When $f_y \times P_y.y > \max\{d_y\}$, reducing $f_y$ will reduce the number of iterations in each partition without changing the number of prefetch operations. This will lead to a memory schedule that is much longer than the ALU schedule, which results in a large average schedule length when compared with the balanced schedule. When $f_y \times P_y.y \leq \max\{d_y\}$, reducing $d_y$ will reduce the number of prefetch operations as well as the number of iterations in the partition. However, in this situation, the number of prefetch operations is close to the number of iterations in a partition, which also means a very unbalanced schedule. Therefore, reducing $f_y$ will lead to a sharp decrease in performance. To satisfy the memory constraint, we prefer to reduce $f_x$ instead of $f_y$ since reducing $f_x$ only interferes with the balanced schedule by a small multiple of the number of keep operations.

3.3.3 Balanced schedule

The partition schedule consists of two parts, the ALU and memory schedules. In practice, the lower bound of the average partition schedule length is the average ALU schedule length. To get close to this lower bound, we should make the length of the memory schedule almost equal to the length of the ALU schedule. If we do not consider the first level memory constraint, we can always achieve this goal.

**Definition 9** A balanced schedule is a schedule for which the length of the memory schedule differs from the ALU schedule’s length by at most the execution time of one keep operation.
In the following theorem, we let \( \#\text{pre} \) be the number of prefetch operations, \( \#\text{keep} \) the number of keep operations, and \( \#\text{iter} \) the number of iterations in a partition. \( N_{\text{mem}} \) is the number of memory units and \( N_{\text{ALU}} \) the number of ALU units. \( T_{\text{keep}} \) and \( T_{\text{pre}} \) are the keep operation time and prefetch time, respectively, and \( L_{\text{ALU}} \) the length for one iteration in ALU part.

**Theorem 11** A partition schedule is a balanced schedule as long as it satisfies the following condition. Assume that \( N_{\text{ALU}} \leq N_{\text{mem}}, T_{\text{ALU}} \geq T_{\text{keep}} \). Then

\[
\left\lceil \frac{\#\text{pre}}{N_{\text{mem}}} \right\rceil \times T_{\text{pre}} + \left\lceil \frac{\#\text{keep}}{N_{\text{mem}}} \right\rceil \times T_{\text{keep}} \leq L_{\text{ALU}} \times \#\text{iter} + T_{\text{keep}} \quad (3.1)
\]

**Proof:** In the memory part of the schedule, the length of the prefetch part is \( \left\lceil \frac{\#\text{pre}}{N_{\text{mem}}} \right\rceil \times T_{\text{pre}} \), and the length of the keep part is \( \left\lceil \frac{\#\text{keep}}{N_{\text{mem}}} \right\rceil \times T_{\text{keep}} \). The length of the ALU part of the schedule is \( L_{\text{ALU}} \times \#\text{iter} \). If the above inequality is satisfied, we will have enough space in the memory part to schedule all of the memory operations. Furthermore, at the bottom of the memory part of the schedule, we leave out \( T_{\text{keep}} \) control steps to schedule those potential keep operations which correspond to the computational nodes in the last control step in the ALU part. Since \( N_{\text{ALU}} \leq N_{\text{mem}} \) and \( T_{\text{ALU}} \geq T_{\text{keep}} \), a legal memory schedule is guaranteed. Therefore, the length of the memory part of the schedule is at most \( T_{\text{keep}} \) control steps longer than that of the ALU part.

Once a balanced schedule is known, the following theorem proves that we can always reach this schedule by tentatively selecting the partition size. This is also the method of how to deciding the partition size which allow us to obtain a balanced schedule.
**Theorem 12** Given a partition, if there exist some $f_x$ and $f_y$ such that

1. $f_y \times P_y \geq d_y \forall d = (d_x, d_y) \in D$

2. $f_x > \max\{d_x - d_y \frac{P_{x,y}}{P_{x,x}}\}$ then the partition schedule is balanced.

**Proof:** These two conditions guarantee that there is no delay vector spanning more than two partitions. If the difference in length between the memory and ALU schedules is less than $T_{keep}$, we have a balanced schedule. On the other hand, if the memory part is more than $T_{keep}$ time units longer than the ALU part, we can always enlarge $f_y$, since (1) guarantees that the number of prefetch operations will not change with the increasing of $f_y$ and (2) guarantees that the number of keep operations is increasing at a slower rate than that of the number of iterations in a partition. Combined with the assumption of Theorem 11, we can reach the point when the memory and ALU parts are balanced. $\Box$

### 3.3.4 Partition schedule under memory constraint

The above subsection illustrates how to find a balanced schedule. The memory requirement of this kind of balanced schedule may exceed the memory constraint. In this case, we can satisfy the memory constraint by reducing the partition size according to Theorem 10.

We have mentioned that reducing $f_x$ can reduce the memory requirement and can achieve much better performance than reducing $f_y$ because it only unbalances the partition schedule by some number of keep operations, which will add little overhead to the average schedule length. To satisfy the memory constraint, we will reduce the partition size mainly by reducing $f_x$.

For a partition with size $P_x \times f_y P_y$, we can easily calculate its memory requirement by using the result from Sections 2.3 and 4.2. Let $\text{mem}_{\text{keepbase}}$ and
\( \text{mem}_{\text{prebase}} \) be the memory requirements for keep and prefetch operations for such a partition, respectively.

After all delay vectors have finished rotating for the ALU part, the projection of any delay vector on the x-axis along the direction of \( P_y \) can be calculated. Sorting all these projections in increasing order, the x-axis can be divided into intervals whose two endpoints are two adjacent projections in the sorted list. When \( f_x \) is within an interval and \( f_y \) is the same as for the balanced schedule, the memory requirement can be obtained by the following theorem.

**Theorem 13** Let the coordinates of the left and right endpoints for the \( m^{th} \) interval be \( PL_m \) and \( PR_m \), respectively, so that \( PR_m = PL_{(m+1)} \), and \( PL_1 = 0 \). When \( PL_m < f_x \leq PR_m \) and \( f_y \) satisfies \( f_y \times P_y \cdot y \geq \max\{y\} \), the memory requirement is:

1. \( \text{Size}_{\text{mem-require}} = \text{Size}_{\text{inter}} + f_x \times \text{mem}_{\text{prebase}} + \text{mem}_{\text{keep}} \)

2. \( \text{mem}_{\text{keep}} = \text{mem}_{\text{keepbase}} + \sum_{n=1}^{m} PL_n \times (f_y \cdot P_y \cdot y - dy) + (\# \text{interval} - m - 1) \times f_x \times (f_y \cdot P_y \cdot y - dy) \), where \#interval represent the overall number of intervals.

**Proof:** It can be obtained directly from the results in Subsection 4.2.

Therefore, we can use Theorem 13 to calculate the memory requirement for the dividing point from left to right, until we find the first dividing point that cannot satisfy the memory constraint. Then, at each step, we increase \( f_x \) by one and calculate its memory requirement using Theorem 13 until it can not be increased because of the memory constraint. Thus, this partition size will give us the optimal average schedule length under the memory constraint using the scheduling method introduced in this paper.
3.4 Experimental Result

In this section, the effectiveness of our algorithm is evaluated by running a set of DSP benchmarks. We assume a prefetch time of 10 CPU clock cycles, which is reasonable when considering the big performance gap between the CPU and main memory in contemporary computer systems. We apply five different algorithms on these benchmarks: list scheduling, a hardware prefetching scheme, a base partition algorithm, the pen-tiling algorithm and our algorithm. The list scheduling algorithm is the most traditional algorithm. It is a greedy algorithm which seeks to arrange the MDFG node as early as possible while satisfying the data dependence. In list scheduling, we use the same architecture model as that in our algorithm, but the ALU part uses the traditional list scheduling algorithm and the memory is not partitioned. In hardware prefetching scheduling, we use the model presented in [15]. In this model, to take advantage of the data locality, the next block in the remote memory is also loaded whenever a block is loaded from the remote memory to local memory. The same architecture model is also used. We use the multi-dimensional rotation scheduling algorithm to arrange the computations in the ALU schedule. Furthermore, the prefetching operations are added in the memory part. However, no partition is considered here. In the base partition algorithm, partitions are also used and the partition shape is the same as that in our algorithm, but the partition size is decided intuitively: each time $f_x$ and $f_y$ are increased by one in turn until the memory constraints are reached. This size is then the partition size used in the base partition algorithm. The pen-tiling algorithm presents a scalable criterion to define optimal tiling. This criterion, related to the communication to computation ratio of a tile, only depends upon its shape, not its size. The pen-tiling algorithm solves a combinatorial problem to
find a basic tile, then determines the final tile size depending on the local memory size constraint. We use the same architecture model and memory size constraints in the experiments using the pen-tiling algorithm. Because there is no discussion of the ALU schedule in [61], we use list scheduling in the experiment.

The first table presents the results without memory constraints, while the other two tables describe the results with memory constraints. Because the local memory requirements for different benchmarks differ greatly, it is more reasonable to adopt relative memory constraints instead of a fixed constraint for all benchmarks.

In the first table, the first column lists the benchmarks’ names “WDF”, “IIR”, “DPCM”, “2D”, “MDFG” and “Floyd” stand for Wave Digital filter, Infinite Impulse Response filter, Differential Pulse-Code Modulation device, Two Dimensional filter, Multi-Dimensional Flow Graph, and Floyd-Steinberg algorithm, respectively. The partition column lists the two boundary partition vectors which decide the optimal partition size, $V_x = f_x P_x$ and $V_y = f_y P_y$. In the two algorithms that use partitioning, $m_{req}$ represents the memory requirement under this partition size. For all algorithms $len$ represent the schedule length for one iteration. The $ratio$ column denotes the improvement our algorithm can obtain when compared with other algorithms.

In the Tables 2 and 3, we compared the effectiveness of the algorithms under memory constraints. List scheduling and hardware prefetching scheduling both have minimal memory requirements since they only consider the schedule for one iteration, as opposed to other algorithms which consider the schedule for the entire partition. Thus their schedule lengths stay constant in benchmarks which reduce the memory requirement. However this constraint has a large influence on our algorithm, the base partition algorithm and the pen-tiling algorithm. So we
compared these three algorithms’ performance with the reduction of memory size. All items have the same meanings as in Table 1.

As we can see from these tables, list scheduling and hardware prefetching scheduling have much worse performance than the other two algorithms. The reason is that, in list scheduling, the schedule is dominated by a long memory schedule, which is far from the balanced schedule. In hardware prefetching scheduling, little compiler-assisted information is available. Although the performance differs with data locality, it has on average the same performance as list scheduling.

In the first table, the pen-tiling algorithm performs worse than either the base partition algorithm or our algorithm. Because the list scheduling algorithm is used in the ALU schedule for the pen-tiling algorithm, its performance is restricted by the ALU part, which has a larger lower bound than would exist if we used the multi-dimensional rotation algorithm for the ALU schedule instead. This comparison demonstrates the benefit we can get from the soft pipelining technique. The base partition algorithm can sometimes compete with our algorithm in the case without memory constraints. This is mainly due to the large partition size. When we add the memory constraints, the performance difference is obvious from the last two tables.

Our algorithm presented in this chapter can get the best result among these algorithms with or without the memory constraints. As the ratios in our tables indicate, the performance gain can be significant when our results are compared with those of the other algorithms. Deciding the partition shape and size is not complex. In our experiment, all of the partition sizes can be decided in less than three seconds on a UltraSparc-30 platform.
**TABLE 3.1**

**WITHOUT MEMORY CONSTRAINTS ASSUMING** $T_{prefetch} = 10$

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition</th>
<th>our</th>
<th>list</th>
<th>base</th>
<th>hardware</th>
<th>Pen-tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Vx</td>
<td>Vy</td>
<td>m_req</td>
<td>len</td>
<td>ratio</td>
</tr>
<tr>
<td>WDF</td>
<td>(4,0)</td>
<td>-12</td>
<td>4</td>
<td>116</td>
<td>4.06</td>
<td>16</td>
</tr>
<tr>
<td>IIR</td>
<td>(6,0)</td>
<td>-14</td>
<td>7</td>
<td>245</td>
<td>6.02</td>
<td>34</td>
</tr>
<tr>
<td>DPCM</td>
<td>(12,0)</td>
<td>-16</td>
<td>8</td>
<td>564</td>
<td>4.01</td>
<td>23</td>
</tr>
<tr>
<td>2D</td>
<td>(3,0)</td>
<td>0</td>
<td>4</td>
<td>227</td>
<td>12</td>
<td>53</td>
</tr>
<tr>
<td>MDFG</td>
<td>(3,0)</td>
<td>(0,23)</td>
<td>463</td>
<td>4.01</td>
<td>40</td>
<td>89.98%</td>
</tr>
<tr>
<td>Floyd</td>
<td>(4,0)</td>
<td>-12</td>
<td>4</td>
<td>149</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>
TABLE 3.2

REDUCING THE AVAILABLE MEMORY TO 2/3 OF THE OPTIMAL

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition size</th>
<th>our</th>
<th>partition size</th>
<th>base</th>
<th>partition size</th>
<th>Pen-tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vx</td>
<td>Vy</td>
<td>m_req</td>
<td>len</td>
<td>Vx</td>
<td>Vy</td>
</tr>
<tr>
<td>WDF</td>
<td>(2,0)</td>
<td>(-12, 4)</td>
<td>82</td>
<td>4.5</td>
<td>(3,0)</td>
<td>(-9,3)</td>
</tr>
<tr>
<td>IIR</td>
<td>(3,0)</td>
<td>(-18, 9)</td>
<td>181</td>
<td>6.04</td>
<td>(5,0)</td>
<td>(-10,5)</td>
</tr>
<tr>
<td>DPCM</td>
<td>(10,0)</td>
<td>(-16, 8)</td>
<td>423</td>
<td>4.01</td>
<td>(9,0)</td>
<td>(-18,9)</td>
</tr>
<tr>
<td>2D</td>
<td>(2,0)</td>
<td>(0,4)</td>
<td>179</td>
<td>12</td>
<td>(3,0)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>MDFG</td>
<td>(2,0)</td>
<td>(0,19)</td>
<td>347</td>
<td>5.26</td>
<td>(8,0)</td>
<td>(0,7)</td>
</tr>
<tr>
<td>Floyd</td>
<td>(2,0)</td>
<td>(-12,4)</td>
<td>111</td>
<td>6</td>
<td>(3,0)</td>
<td>(-9,3)</td>
</tr>
</tbody>
</table>
**TABLE 3.3**

MEMORY REQUIREMENT IS 1/2 OF THE ORIGINAL

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition size</th>
<th>our</th>
<th>partition size</th>
<th>base</th>
<th>partition size</th>
<th>Pen-tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vx</td>
<td>Vy</td>
<td>m_req</td>
<td>len</td>
<td>Vx</td>
<td>Vy</td>
</tr>
<tr>
<td>WDF</td>
<td>(1,0)</td>
<td>(-12, 4)</td>
<td>50</td>
<td>5.25</td>
<td>(2,2)</td>
<td>(-6,2)</td>
</tr>
<tr>
<td>IIR</td>
<td>(2,0)</td>
<td>(-12, 6)</td>
<td>117</td>
<td>6.83</td>
<td>(3,0)</td>
<td>(-8,4)</td>
</tr>
<tr>
<td>DPCM</td>
<td>(4,0)</td>
<td>(-16, 8)</td>
<td>267</td>
<td>4.5</td>
<td>(6,0)</td>
<td>(-12,6)</td>
</tr>
<tr>
<td>2D</td>
<td>(1,0)</td>
<td>(0,4)</td>
<td>113</td>
<td>12</td>
<td>(2,0)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>MDFG</td>
<td>(1,0)</td>
<td>(0,17)</td>
<td>232</td>
<td>6.24</td>
<td>(5,0)</td>
<td>(0,5)</td>
</tr>
<tr>
<td>Floyd</td>
<td>(1,0)</td>
<td>(-9,3)</td>
<td>64</td>
<td>7.67</td>
<td>(2,0)</td>
<td>(-6,2)</td>
</tr>
</tbody>
</table>
We have two part schedules in the architecture model. The ALU part executes the computation, while the memory part prepares all data for the next ALU partition computation, which means that the memory accesses and processor computations have been overlapped well, adding little overhead to the ALU part. Comparing data in these tables, we can see that the memory latency has been successfully hidden.

3.5 Conclusion

In this chapter, an algorithm which yields the minimum schedule length under memory constraints was proposed. This algorithm explores the ILP among instructions by using retiming techniques, while joining with data prefetching to produce high throughput schedules. Under our method, an ALU schedule and a memory schedule are produced for the partition. Then, through the study of the properties of different partition sizes under different memory constraints, the algorithm gives a partition size and shape so that the overall optimal average schedule length can be obtained. Experiments on DSP benchmarks show that this algorithm can always produce an optimal solution.
CHAPTER 4

MULTIPLE LEVEL PARTITIONING FOR HIDING MEMORY LATENCY

This chapter presents two-level partitioning algorithm for the system with multi-level memory hierarchy. We have briefly introduced the concept of one-level partition technique in Chapter 2 [84] and discussed in detail one level partitioning with memory constraint in last chapter [29]. However, no analysis is performed with three or more levels of memory, as is the case in most contemporary computer architectures. Our studies have shown that the one-level partition technique cannot take full advantage of the second level memory in such an architecture.

For instance, assume that 3 clock cycles are needed to fetch data from the second to the first level memory, 7 clock cycles to fetch data from the third to the second level memory, and 10 clock cycles to fetch data from the third to the first level memory. Suppose also that whenever a block is brought into the first level memory directly from the third level memory, a copy is kept in the second level memory. Our experiments have shown that the one-level partition technique produces poor results on such three-level memory hierarchy systems. With the DPCM filter as a benchmark, using the one-level partition technique for the first level memory and dynamic scheduling for the second level memory will result in almost the same average access time as the model in which no second level memory exists. The one-level partition technique resulted in an average fetching time of 9.968 cycles/fetch when the iteration space is large. In other words, the CPU always has
to load data from the third level memory, which has a cost of 10 cycles/fetch. On the contrary, our two-level partitioning technique, by appropriately choosing the partition shape and size, can prefetch all the data into the second level memory before the first level memory needs to access them. These prefetch operations execute in parallel with the ALU computations. Therefore, the average fetching time can be regarded as 3 cycles/fetch in this case. The lower average fetching time is the key to better performance under the first level memory size constraint, which is demonstrated by the experimental results.

In this chapter, we apply the idea of partitioning to the extra levels in the memory hierarchy. In both the first and second level memories, we adopt the partition technique and make extensive use of compile time information about the usage pattern of the information produced in the loop to generate an approximated “balanced schedule” in the first and second level schedules in order to minimize the overall average execution time and make better use of the second level memory. This chapter presents methods for deciding the best partition in both levels in order to achieve this goal. The new algorithm exceeds the performances of existing algorithms by optimizing both the ALU and memory schedules in the first level and taking into consideration a balance between the schedules of the two levels. In the situation of no memory constraint, the results produced by our algorithm will come within one instruction time unit of the theoretic lower bound. Experiments show the improvement can reach about 35% over the one-level partitioning algorithm with memory constraints and more than 80% over the traditional hardware prefetching scheme or list scheduling.
4.1 Architecture model

Our technique is designed for use in a system containing a processor with multiple ALUs and memory units. The first level memory is also located in the processor. The second and third level memories are off-chip memories. The first level memory has the tightest memory size constraint and the fastest access speed. The second level memory has medium memory size and access speed. The third level memory has the largest memory size and slowest access speed. This architecture is similar to the real system with L1 cache, L2 cache and main memory. Our technique is to load data into the first level memory before its explicit use so that the overall cost of accessing data can be minimized. Therefore, overlapping the ALU computations and the memory accesses will lead to a shorter overall execution time. The goal of our algorithm is to tolerate the memory access time by overlapping the ALU computations as much as possible.

Our scheme is a software-based method in which some special prefetching instructions are added to the code when it is compiled. When the processor encounters these instructions during program execution, it will pass them to the special hardware memory units for handling. The function of the memory unit is the same in both two levels: get the data ready before the lower level cache needs to reference them. Please refer to Section 1 of Chapter 3 for the details of memory unit and its operations. In the second level, only keep_1 operations exist.

Note that the lower level memories in the architecture model cannot be seen as pure caches, because issues such as cache consistency and cache conflict are not considered here. In other words, the lower level memory can be thought as a fully associative cache with some memory units connected to it.

Corresponding to ALU and memory units, there are ALU and memory sched-
ules in one-level partition algorithm [84]. To form the ALU schedule, Multi-
dimensional rotation scheduling algorithm [58] is used to obtain the schedule of a 
single iteration. Then this schedule is simply duplicated for each iteration in the 
partition. The memory schedule is formed by considering all the memory oper-
tions in the entire partition. The partition size is select to balance the ALU and 
memory schedules such that the long memory latency is effectively tolerated.

4.2 Two-level partition

In reality, contemporary computer systems always have more than two levels 
of memory. It is necessary to study scheduling with more levels of memory. In 
our model, we use three levels of memory, not only because it is the most common 
case, but because we can apply the same idea to a memory hierarchy with more 
than three levels.

We first consider the situation with dynamic scheduling (FIFO) in the second
level memory. Assuming that the access time from the third to first level memories
is 10 time units, that from the second to first level memories is 3 time units and
that from the third to second level memories is 7 time units, Table 4.1 gives the
average access time of the first level memory for two different benchmarks under
different iteration space sizes. This access time includes the time from the third to
first level memories and from the second to first level memories. In the experiment,
the first level memory size is the memory requirement, while the size of the second
level memory is ten times larger than that of the first. Using the relative second
level memory size can make us concentrate on the effect of more memory levels
and ten times larger is a reasonable assumption.

In the table, we use two different benchmarks (DPCM and WDF) and two dif-
ferent algorithms (one-level partition algorithm and pen-tiling algorithm, which is
introduced in the experiment section) to decide the first level partition. The table
lists the partition size \( p_{\text{size}} \), denoted by two vectors, and the first level memory
requirements \( m_{\text{req}} \) for each benchmark and algorithm with the benchmark’s name.
The average access time is listed for each iteration space. For example, \( 30 \times 30 \) in
the table indicates the two dimensional loop with each dimension equal to 30.
## TABLE 4.1

THE AVERAGE ACCESS TIME UNDER DIFFERENT ITERATION SPACE SIZES

<table>
<thead>
<tr>
<th>One-level partition algo</th>
<th>Pen-tiling algo</th>
<th>One-level partition algo</th>
<th>Pen-tiling algo</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPCM p_{size} = (4, 0)</td>
<td>50 x 50</td>
<td>6.211</td>
<td>DPCM p_{size} = (6, 0)</td>
</tr>
<tr>
<td>100 x 100</td>
<td>7.884</td>
<td>WDF p_{size} = (1, 0)</td>
<td>50 x 50</td>
</tr>
<tr>
<td>×(-16, 8)</td>
<td>9.872</td>
<td>×(-12, 4)</td>
<td>100 x 100</td>
</tr>
<tr>
<td>m_{req} = 267</td>
<td>9.968</td>
<td>m_{req} = 265</td>
<td>400 x 400</td>
</tr>
<tr>
<td>800 x 800</td>
<td>9.976</td>
<td>m_{req} = 50</td>
<td>9.991</td>
</tr>
</tbody>
</table>

m_{req} = 64
If the dynamic scheduling in the second level memory can take good advantage of the second level memory, we expect to see an average access time close to 3, which is the access time from the second to first level memories. However, from the data in the table, we can see that having more memory levels does not result in a substantial improvement when the iteration space is large. The reason is that the dynamic algorithm cannot predict the data access sequence. In addition, the data have already been swapped out to the third level memory by the time they are needed. This demonstrates that dynamic scheduling cannot make use of the information obtained during compilation to improve data locality and reduce access time. This phenomenon leads us to think about partitioning in the second level memory as well as in the first level memory.

The objective of the second level partition is to prefetch some data from the third to the second level memories and keep some data in the second level memory for the near future use. Therefore, whenever the first level memory wants to access these data, it can always find them in the second level memory, thus eliminating the time used to fetch data from the third level memory.

4.2.1 The different delay dependences

When considering the second level partition, we can treat the first level partition as the basic integral unit. Then, the second level partition will be the combination of a number of first level partitions. For instance, the second level partition size in Figure 4.2 is $3 \times 3$ first level partitions.

In the second level, the current partition, next partition and other partition have the same definitions as in the first level. In the case of two level partitions, there will be four kinds of delay dependences which need to be treated differently.
Figure 4.2. The different kinds of delay dependence for the second level memory

when scheduling the loop. They are indicated by $d_1$, $d_2$, $d_3$, and $d_4$ in Figure 4.2, the dotted lines delimiting first level partitions here. $d_1$ is a delay dependence that goes to one other partition from the current partition in both the first and second levels. For this delay dependence, the prefetch operation is needed in both levels. In the second level, $d_3$ is an intra-partition delay dependence, while it goes to the next partition in the first level. It only needs keep operations in the first level partition schedule and has no effect on the second level partition schedule.

Delay dependence $d_4$ is an intra-partition delay dependence in the second level but goes to one other partition in the first level. Therefore, it will be treated with a prefetch operation in the first level and a keep operation in the second level in order to retain the corresponding data for near future use. Moreover, some write-back operations are needed to put back the data corresponding to this delay dependence into the second level memory.

The delay dependence $d_2$ goes to the next partition in both first and second levels. Due to the execution sequence (the loop is executed by the first level partition sequence along the x-axis until the right boundary of the second level partition is reached, then continues from the left boundary of the second level partition)
partition along the next row of first level partitions), this delay dependence gives rise to some complications in the schedule. In the first level partition it cannot be kept in the first level memory as usual, since it will not be needed until the execution reaches a partition in the next second level partition. Instead, from the standpoint of the first level partition schedule, the data will be prefetched from the second level memory before it is needed. Thus, we will have the following definition.

**Definition 10** A boundary partition is a first level partition with at least one delay dependence going to the next partition in the second level.

In a boundary partition, some or all of the keep operations in an ordinary partition transform into prefetch operations. The number of keep operations which become prefetch operations depends on the relation between the first level partition size and the distance of the delay dependence. For instance, if there are only \(\text{keep}_1\) operations in the ordinary first level partition, the boundary partitions are found only in the rightmost column of first level partitions. If a \(\text{keep}_2\) operation also exists, then the boundary partition also includes the next right-most column of first level partitions. However in a first level partition of this column, only \(\text{keep}_2\) operations become prefetch operations. After deciding which keep operations will become prefetch operations, the boundary partition schedule lengths can be determined accordingly. In the second level partition, the data corresponding to \(d_2\) will be also prefetched from the third level memory to the second level for use in the next second level partition.
4.2.2 The write back operation

As about write back operations which write data back to the higher level memory, they can be considered as being hidden in the prefetch operations. We first prove that the number of write-back operations is the same as the number of prefetch operations in the first level partition.

**Lemma 14** In the first level partition, the number of the write-back operations is always the same as the number of prefetch operations.

*Proof:*

![Figure 4.3. The area of the prefetch and writing back operation](image)

In Figure 4.3, the shaded upper region denotes the area of write-back operations and the shaded lower region represents the area of prefetch operations. If there is a delay dependence to another partition in the shaded upper region, the corresponding data need to be written back to the higher level memory. At the same time, there must exist the same delay dependence coming into the shaded lower region from another partition, which means the corresponding data need to
be prefetched in the previous partition. Therefore, they have identical number of operations.

As discussed in [84], one memory operation (prefetch or keep) consumes two memory locations. One is used for the datum needed in the current partition, the other is used for next partition’s computation. Provided that the numbers of write back and prefetch operations are identical, it is possible to treat them as an integrated operation. Next we will prove that the memory consumption is still two for such an integrated operation. At the same time, how to integrate these two operations into one operation is illustrated.

**Theorem 15** Every write back operation can be combined with a prefetch operation into an integrated operation, with two memory locations required.

*Proof:* From the above proof, we know that there exists both a write back operation and a prefetch operation to each delay dependence going into another partition. We can treat them with one operation by the following method.

1. In the current partition, for each delay dependence going into one other partition, we use one memory location to store data prefetched for the use of the next partition, and one memory location to store data which need to be written back.

2. In the next partition, we first write one datum back to the higher level memory, then use this location to store the prefetched data. At the same time, whenever a datum to be written back is computed, a datum prefetched by the previous partition must have been consumed. This location can be used to store the computed data.

☐
From this theorem, we can simply double the prefetch time to consider the write back operation. The model and algorithm can still be useful. The following theorem defines the relationship between the number of write back operations and prefetch operations in the second level partition.

**Theorem 16** The number of prefetch operations and write back operations, which are used to write data back to the third level memory, are identical. These two operation can be regarded as one integrated operation when considering the second level partition schedule.

*Proof:* It is easy to verify this theorem based on the proof of Theorem 15. □

### 4.2.3 Two level partition scheduling

With all the pieces now in place, we perform two-level partition scheduling by considering both levels separately. The first level partition schedule consists of two parts as one-level partition technique [84], as shown in Figure 4.4(a). However the prefetch part is made up of integrated prefetch and write back operations as discussed in the last subsection.

Because the keep operation can only been issued after the corresponding computation, we always issue prefetch operations ahead of all keep operations in the memory part schedule. Also, due to this dependence between keep operation and the corresponding datum’s finishing time, the length of the memory part is always longer than the ALU part. and some idle time may exist in the memory part. Therefore, the schedule length for one partition in the first level $L_{first}$ satisfies the inequality.

$$L_{first} \geq \#prefetch_{first} \times T_{2-1} + \#keep_{first} \times T_{keep}$$
Figure 4.4. (a) The first level partition schedule. (b) Two level partition schedule

where \( \#\text{prefetch}_{\text{first}} \) is the number of prefetch operations per first level partition, \( T_{2-1} \) is the time cost to load data from the second to first levels, \( \#\text{keep}_{\text{first}} \) is the number of keep operations per first level partition, and \( T_{\text{keep}} \) is the time cost per keep operation. The boundary partition schedule length is different. Its value will depend on how many keep operations change into prefetch operations.

In the second level, there exist only two kinds of operation: \textit{prefetch} to fetch those data corresponding to data dependences like \( d_1 \) and \( d_2 \) in Figure 6 and \textit{keep} to keep the data corresponding to the data dependences like \( d_4 \) in Figure 6. The two-level partition schedule is shown in Figure 4.4(b). The length of the \textit{first level schedule} is the summation of all the first level partition schedules in a second level partition and the length of the \textit{second level schedule} is the sum of the lengths of the prefetch and the keep parts.

The length of the prefetch part can be calculated from the area shown in Figure 4.5(a). The length of the keep part can be calculated from the area shown
in Figure 4.5(b). The overall length is

\[ L_{\text{second}} = L_{\text{prefetch}} + L_{\text{keep}} \]
\[ = \#\text{prefetch}_{\text{second}} \times T_{3-2} + \#\text{keep}_{\text{second}} \]

where \#prefetch_{second} is the number of prefetch operations of the second level partition, \(T_{3-2}\) is the time cost to load data from the third to the second level, and \#keep_{second} is the number of keep operations in the second level partition.

4.3 Two Level Partition Scheduling algorithm

Knowing the constitution of the first and second level schedules, we need to find the relation between these two schedules in order to achieve the optimal average schedule length. From the analysis above, the first level schedule can be determined independently. Therefore, the lower bound of the average schedule length is the average schedule length of the ordinary first level partition, which equals the one-level partition schedule length where the prefetch time can be regarded as the fetching time from the second level to the first level. We now wish to determine the second level partition size which will make the overall schedule optimal.
4.3.1 The property of the operation amount

The first level schedule length depends on the ALU schedule, as well as the number of prefetch and keep operations in the first level. On the other hand, the second level schedule length only depends on the prefetch and keep operations in the second level. The following properties show the relationship between the numbers of operations in the first and the second level partitions.

The first level partitions in a second level partition can be classified into ordinary partitions and boundary partitions. Assume that the region size consisting of only ordinary partitions is \( x_1 \times y \) and that consisting of only boundary partitions has size \( x_2 \times y \). Let the number of prefetch operations in the first level be \( \text{pre}_{\text{first}} \) and the number of keep operations be \( \text{keep}_{\text{first}} \). The number of prefetch and keep operations in the second level partition can be calculated as follows.

**Property 2** In the region consisting of only ordinary partitions, the number of prefetches in the second level partition is \( \text{pre}_{2\text{ord}} = x_1 \times \# \text{pre}_{\text{first}} \) and the number of keep operations is \( \text{keep}_{2\text{ord}} = x_1 \times (y - 1) \times \# \text{pre}_{\text{first}} \).

![Figure 4.6. The different sub-regions in the boundary partitions](image-url)
In the region consisting of only boundary partitions, there exist three sub-regions for a given delay dependence $d$, as shown in Figure 4.6. In the figure, the dotted lines delimit the first level partitions. The three different sub-regions are designated in the figure.

Definition 11  In the region consisting of only boundary partitions, the top region for a delay dependence is the region from which this delay dependence starts and goes into one other second level partition. The boundary region for a delay dependence is the region from which this delay dependence starts and goes into the next second level partition. The keep region for a delay dependence is the region from which this delay dependence starts and goes into one other first level partition, while remaining in this second level partition.

The one-level partition scheduling algorithm guarantees that the $P_y$ projection of a delay dependence will not be longer than the length of the first level partition size along the $P_y$ direction. That is to say, the top region is only included in the top row of the first level partitions in a second level partition. The number of integer points in these regions is the number of corresponding operations for this delay dependence. These numbers can be calculated using the same parallelogram idea given in [84]

Property 3  For a given delay dependence, the total number of integer points in the top region and the boundary region is the number of prefetch operations in the boundary partitions. The number of integer points in the keep region is the associated number of keep operations in the boundary partitions.
4.3.2 The balanced two levels partition schedule

Since the lower bound of the length of the two-level partition schedule is determined by the first level schedule, we should make the first level schedule longer than the second level schedule to achieve good performance. Analyzing the relationship between the first and second level schedules, we first give the following definition.

**Definition 12** Given the first level partition schedule and a second level partition size, a **balanced two-level partition schedule** is a schedule in which the first level schedule is longer than the second level schedule and the schedule length difference between the first and second level is minimal.

The following theorem proves the existence of such a balanced two level partition schedule.

**Theorem 17** If the schedule of the first level partition is determined, a second level partition size that makes the two level partition schedule a balanced two level partition schedule can always be found.

**Proof:** Assume the size of the second level partition size is $x \times y$. We first prove that the schedule length difference between the first level and the second level will increase as the value of $x$ is enlarged.

Suppose that the second level partition size increased from $x \times y$ to $(x+1) \times y$, where the value of $x$ is large enough that the number of boundary partitions will not increase when $x$ is enlarged. The length of the first level schedule increases by more than $(\#pre_{first} \times T_{21} + \#keep_{first}) \times y$, while that of the second level schedule length increases by more than $\#pre_{first} \times T_{32} + \#pre_{first} \times (y - 1)$. The first item is always larger than the second item as long as $y \geq \frac{T_{32} - 1}{T_{21} - 1}$.
We now prove the existence of a second level partition size which can make the first level schedule longer than the second level schedule. Assume the size of a region consisting of only boundary partitions is $z \times y$. Then the first level schedule length satisfies:

$$L_{\text{first}} > (\#_{\text{pre} \text{first}} \times T_{21} + \#_{\text{keep} \text{first}})xy$$

The second level schedule length satisfies:

$$L_{\text{second}} < \#_{\text{pre} \text{first}}(x-z)T_{32} + \#_{\text{pre} \text{first}}xy + (\#_{\text{pre} \text{first}} + \#_{\text{keep} \text{first}})T_{32}zy$$

Therefore,

$$L_{\text{first}} - L_{\text{second}} > \#_{\text{pre} \text{first}}(T_{21}xy - T_{32}x - xy - T_{32}zy) + \#_{\text{keep} \text{first}}(xy - T_{32}zy)$$

When the following is satisfied, we can guarantee that the first level schedule length is longer than the second level schedule length.

$$\begin{cases} 
x & \geq T_{32}zy \\
y & \geq \frac{T_{32}+1}{T_{21}-1} \end{cases}$$

In fact, these are rather loose constraints for finding a feasible size.

\[ \square \]

4.3.3 The calculation of the memory requirement

In the two level partition schedule, we need to calculate both the first and second level memory requirements.

**Theorem 18** The first level memory requirement $\text{mem}_{1, \text{req}}$ is calculated by the following equation:

$$\text{mem}_{1, \text{req}} = \text{mem}_{\text{inter}} + \sum \#_{\text{keep} \cdot n} \times (n+1) + 2 \times \#_{\text{pre} \text{first}}$$

In the equation, $\text{mem}_{\text{inter}}$ is the memory requirement for the intermediate data in the first level partition, $\#_{\text{keep} \cdot n}$ is the number of keep-$n$ operations, and
\( \#\text{pre}_{\text{first}} \) is the number of prefetch operations.

**Proof:**

From Theorem 15, the number of write-back operations equals the number of prefetch operations, and their integrated operation only requires 2 memory locations. Thus in the first level memory, the memory requirement consists of three parts: those used to store prefetched and written back data, those used to store kept data, and those used to store intra-partition intermediate data. Based on the knowledge of the memory consumption for each kind of memory operation introduced before, the above equation is true.

We can use the above theorem to derive the memory requirement for the first level memory, in which the calculation of \( \text{mem}_{\text{inter}} \) is the same as in the one-level partition algorithm.

**Theorem 19** The memory requirement for the second level \( \text{mem}_{2\text{req}} \) is calculated by the equation

\[
\text{mem}_{2\text{req}} = 2 \times \#\text{pre}_{\text{second}} + \#\text{keep}_{\text{second}},
\]

where \( \#\text{pre}_{\text{second}} \) represents the number of prefetch operations in the second level, and \( \#\text{keep}_{\text{second}} \) is the number of keep operations.

**Proof:** From the above analysis we know that the second level partition schedule consists of only two parts, one consisting of prefetch operations, the other of keep operations. The memory requirement also depends on these two parts. The keep operations in the second level partition are different from those in the first level partition. Corresponding data are only kept in the second level memory for the current partition to use; hence the lifetime is only one second level partition. This
keep operation only needs one memory location to store the data. Thus we can
directly obtain the above equation using previously derived information. □

Combining the above theorem and two properties in Section 4.3.1, the memory
requirement for both the first and second level memories can be calculated.

In choosing the second level partition size, we select the size that can make
the two level partition schedule a balanced two level partition schedule. This is
the tradeoff between the performance improvement and the memory consumption.
After achieving a balanced two level partition schedule, increasing $x$ can still bring
some performance improvement. On the other hand, the memory requirement is
increasing at a much larger rate than the performance improvement. For instance
for an IIR filter, when there is some first level memory constraint, the balanced
two level partition schedule requires a second level partition size of $7 \times 10$. If the
second level partition size is increased by one along the $P_x$ direction, the memory
requirement increases by 12.94%, while the schedule improvement increases by
only 0.058%. Therefore it is more reasonable to adopt a second level partition size
which can generate a balanced two level partition schedule.

We have demonstrated that we can always find such an $x$ to generate a balanced
two level partition schedule as long as $y$ is large enough. Therefore, there exist
different such $x$ associated with different $y$. The following algorithm gives the
method to calculate the minimum second level memory requirements and the
corresponding second level partition size.

In this algorithm, the symbol $# pre_1$ denotes the number of prefetch operations
in one first level partition, $keep_{1,n}$ is the number of keep-n operations in one first
level partition; $T_{3-2}$ and $T_{2-1}$ are the fetching times from the third level memory
to the second level memory and from the second level memory to the first level

78
memory, respectively; and \( l_{ord} \) is the schedule length of the ordinary first level partition.

In Algorithm 1, after the value of \( y \) which can lead to the minimum second level memory requirement is determined, we calculate an \( x \) value that makes the first level schedule length equal to the second level schedule length using this \( y \). Then \( \lceil x \rceil \) is chosen as the actual \( x \). It can be seen from the proof of Theorem 4 that the difference between the first level and second level schedules is an increasing function of \( x \). The second level partition size calculated using Algorithm 1 guarantees a balanced two level partition schedule.

4.3.4 Algorithm

From the above discussion, the balanced two level partition schedule is a good point to select the second level partition size. It can provide us both the optimal average schedule length and second level memory requirement. After knowing the partition size in both levels, we can generate the schedule for both levels using our previous knowledge. Note that there is a difference between the arrangement of memory operations in the first and second levels. In the first level partition, we can only issue keep operations after the corresponding data are ready, so there may be some idle time in the memory part. In the second level partition, we can arrange the keep operations as soon as the corresponding data have been written back. Since there is no keep operation in the top row of the second level partition, there will be no such idle time in the second level memory.

To create the first level schedule, we duplicate the retimed schedule of one iteration in order to construct the ALU schedule. In the memory part, we first arrange prefetch operations, then keep operations. In the second level schedule,
Algorithm 3 Calculate the minimum second level memory requirement and the corresponding second level partition size

Input: A given first level partition schedule and the corresponding set of delay dependences

Output: The minimum second level memory requirement and the associated second level partition size

1. Use the delay dependence set to determine the width $w$ of the region consisting of only boundary partitions
2. Compute the amount of keep operations $\# \text{keep}\_\text{keep}$ and prefetch operations $\# \text{pre}\_\text{bound}$ in the bottom row of this region
3. Derive the function $f(y)$ to calculate the number of keep operations and $f'(y)$ to calculate the number of prefetch operations.
   \[
   \begin{align*}
   f(y) &= \# \text{keep}\_\text{keep} \times (y - 1) \\
   f'(y) &= \# \text{pre}\_\text{bound} \times (y - 1) + \sum n \times \# \text{keep}\_\text{bound} \\
   &+ \# \text{pre}\_1 \times w
   \end{align*}
   \]
4. Compute $l(y)$ which is the summation of all boundary partition schedule lengths.
5. Let the first level schedule length equal to the second level schedule length and derive the expression for the second level memory requirement $\text{mem}_{\text{req2}}$
   \[
   \begin{align*}
   l_{\text{ord}}(x - w)y + l(y) &= \# \text{pre}\_1 T_{3-2}(x - w) \\
   &+ \# \text{pre}\_1 (x - w)(y - 1) \\
   &+ f'(y) T_{3-2} + f(y) \\
   \text{mem}_{\text{req2}} &= 2\# \text{pre}\_1 (x - w) \\
   &+ \# \text{pre}\_1 (x - w)(y - 1) \\
   &+ 2f'(y) + f(y)
   \end{align*}
   \]
6. Calculate $x$ for a given $y$ using the first equation.
7. Substitute $x$ into the second equation. Calculate the value of $y$ which make $\text{mem}_{\text{req2}}$ minimum.
8. Calculate the value of corresponding $x$ and the second level memory size.
Algorithm 4 The two-level partition schedule

Input: An MDFG, the first level memory constraint
Output: The two level partition schedule

1. Retime the MDFG to get a compact schedule for one iteration. //Using multi-dimensional rotation scheduling algorithm.
2. Based on the delay dependences in the retimed MDFG, determine the first level partition shape.
3. Based on the first level memory constraint and the first level partition shape, calculate the first level partition size. //Using one-level partition algorithm,
4. Create the first level partition schedule.
5. Use this first level partition schedule and Algorithm 3 to compute the second level partition size.
6. Calculate the corresponding second level memory requirement.
7. Create the second level schedule.

the prefetch and keep operations are arranged in turn as we have mentioned above.

In this chapter, we illustrate the two-level partition scheduling with two dimensional iteration space, because it is the most general case in multi-dimensional DSP applications. It is important to note that this scheme can be extended to more dimensional iteration space with little modification. All the concepts and algorithms under more dimensional case are easily deduced. Take 3-dimensional loop as an example, the partition will become a cube whose each face is a parallelogram. A partition is delimited by three partition vectors: $P_x$, $P_y$ and $P_z$. All data dependences must lie inside these partition vectors to prevent the dependence cycle between different partitions. The partition execution sequence is along $X$ axis first, then $Y$ axis and $Z$ axis. The identification of memory operations and the algorithm still applied. The partition size which can achieve the balanced partition schedule is still the preferred choice.
4.4 Experiment

In this section, the effectiveness of the two-level partition technique is evaluated by running a set of DSP benchmarks. In the experiments we assume that the time to load data from the third to second level memories is 9 clock cycles, from the second to first level memories is 3 clock cycles, and from the third to first level memories is 12 clock cycles. We compared six different schemes. They are two-level partition algorithm, one-level partition algorithm in [84], pen-tiling algorithm in [61], PSP scheduling in [29], list scheduling and hardware prefetch scheme, respectively. Pen-tiling algorithm presents a scalable criterion to define optimal tiling. This criterion, related to the communication to computation ratio of a tile, only depends upon its shape, not its size. The pen-tiling algorithm solves a combinatorial problem to find a basic tile, then determines the final tile size depending on the first level memory size constraints. It assumes a two-level memory hierarchy in the system. In the experiments, we loose the memory size constraints for the pen-tiling algorithm to demonstrate that our partition method, which balances the computation and communication, can get the better result even under the tighter memory size constraints. PSP scheduling attempts to balance the computation and communication. Nevertheless, the situation with the first level memory size constraints was not considered in PSP scheduling. This algorithm can obtain the same results as our algorithm under no first level memory size constraints, but can not get feasible results when this constraint is imposed. Therefore, their experimental results are not shown in the table.

List scheduling is the most traditional algorithm. It is a greedy algorithm which seeks to schedule a MDFG node as early as possible while satisfying the data dependence and resource constraints. In our experiment, we use list scheduling
to schedule the ALU operations, but the memory is not partitioned. In hardware prefetching scheduling, we use the model presented in [15]. In this method, to take advantage of the data locality, the next block in the higher level memory is also loaded whenever a block is loaded from the higher level to the lower level memories. We use the multi-dimensional rotation scheduling algorithm to arrange the computations in the ALU schedule. Furthermore, the prefetching operations are added in the memory part. However, no partition is considered here.

The first table presents the results without the memory constraint in the first level. In this table, we use the same first level partition size as shown in the table in both the one-level and two-level partition scheduling algorithms. Also, they have the same average schedule length. The other two tables describe the results with memory constraints. In the last two tables, the relative memory constraints for all benchmarks are used due to the large differences among their memory requirements.
## TABLE 4.2

WITHOUT MEMORY CONSTRAINTS ASSUMING $T_{\text{prefetch}} = 12$

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Two-level</th>
<th>One-Level</th>
<th>pen-tiling</th>
<th>list</th>
<th>hardware pre</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$m_{\text{req}}$</td>
<td>size2</td>
<td>size1</td>
<td>len</td>
<td>$m_{\text{req}}$</td>
</tr>
<tr>
<td>IIR</td>
<td>1159</td>
<td>6 x 2</td>
<td>4 x 8</td>
<td><strong>6.031</strong></td>
<td>358</td>
</tr>
<tr>
<td>DPCM</td>
<td>3982</td>
<td>5 x 3</td>
<td>12 x 10</td>
<td><strong>4.008</strong></td>
<td>840</td>
</tr>
<tr>
<td>WDF</td>
<td>486</td>
<td>6 x 2</td>
<td>4 x 5</td>
<td><strong>4.05</strong></td>
<td>129</td>
</tr>
<tr>
<td>Floyd</td>
<td>477</td>
<td>3 x 2</td>
<td>7 x 4</td>
<td><strong>6.000</strong></td>
<td>216</td>
</tr>
<tr>
<td>2D</td>
<td>491</td>
<td>3 x 2</td>
<td>3 x 5</td>
<td><strong>12</strong></td>
<td>241</td>
</tr>
</tbody>
</table>
### Table 4.3

**Experimental Results with About 1/2 of Original Size**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>One-level</th>
<th></th>
<th>pen-tiling</th>
<th></th>
<th>Two-level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size1</td>
<td>$m_{req1}$</td>
<td>len</td>
<td>size</td>
<td>$m_{req1}$</td>
<td>len</td>
</tr>
<tr>
<td>IIR</td>
<td>1 x 6</td>
<td>175</td>
<td>10</td>
<td>4 x 4</td>
<td>193</td>
<td>10.125</td>
</tr>
<tr>
<td>DPCM</td>
<td>3 x 9</td>
<td>416</td>
<td>5</td>
<td>7 x 7</td>
<td>427</td>
<td>5.347</td>
</tr>
<tr>
<td>WDF</td>
<td>1 x 5</td>
<td>74</td>
<td>5</td>
<td>3 x 3</td>
<td>89</td>
<td>6.111</td>
</tr>
<tr>
<td>FLOYD</td>
<td>2 x 3</td>
<td>109</td>
<td>8.333</td>
<td>3 x 3</td>
<td>128</td>
<td>7.889</td>
</tr>
<tr>
<td>2D</td>
<td>1 x 4</td>
<td>112</td>
<td>14</td>
<td>2 x 2</td>
<td>128</td>
<td>24</td>
</tr>
<tr>
<td>Benchmark</td>
<td>One-level</td>
<td></td>
<td></td>
<td>pen-tiling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>---</td>
<td>---</td>
<td>-----------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>size1</td>
<td>m_{req1}</td>
<td>len</td>
<td>size</td>
<td>m_{req1}</td>
<td>len</td>
</tr>
<tr>
<td>IIR</td>
<td>1 \times 3</td>
<td>85</td>
<td>15.67</td>
<td>2 \times 2</td>
<td>83</td>
<td>19.75</td>
</tr>
<tr>
<td>DPCM</td>
<td>2 \times 6</td>
<td>216</td>
<td>7.083</td>
<td>2 \times 2</td>
<td>205</td>
<td>9.062</td>
</tr>
<tr>
<td>WDF</td>
<td>1 \times 3</td>
<td>44</td>
<td>8</td>
<td>2 \times 2</td>
<td>52</td>
<td>9</td>
</tr>
<tr>
<td>FLOYD</td>
<td>1 \times 2</td>
<td>43</td>
<td>12.5</td>
<td>2 \times 2</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>2D</td>
<td>1 \times 2</td>
<td>65</td>
<td>25</td>
<td>2 \times 2</td>
<td>128</td>
<td>24</td>
</tr>
</tbody>
</table>
In our experiments, the five benchmarks used are *Infinite Impulse Response filter*, *Differential Pulse-Code Modulation device*, *Wave Digital filter*, *Floyd-Steinberg algorithm* and *Two Dimensional filter*. They are represented by “IIR”, “DPCM”, “WDF”, “Floyd” and “2D”, respectively, in all tables. All the “size1” columns list the partition size of the first level in both the one-level and two-level partition algorithms. The “size2” columns list the partition size of the second level in the two-level partition algorithm. All the “len” columns represent the average schedule lengths, and the “ratio” column in the first table denotes the improvement the two-level partition algorithm can obtain compared with list scheduling and hardware prefetching schemes. The $m_{req1}$ and $m_{req2}$ columns represent the memory requirements of the first and second level memories for each algorithm, respectively.

As we can see from the first table, list scheduling and hardware prefetching scheduling have much worse performance than the other three algorithms. The reason is that, in list scheduling, the schedule is dominated by a long memory schedule, which is far from the balanced schedule. In hardware prefetching scheduling, little compiler-generated information is available. Although the performance differs with data locality, it has on average the same performance as list scheduling. The one-level partition algorithm and pen-tiling algorithm can compete in performance with the two-level partition algorithm in the case without the memory constraints, which is due to the fact that a large first level memory size can efficiently hide the long memory access time. When memory constraints are added, the performance difference is obvious from the last two tables. Moreover, we can see from the last two tables that the two-level partition algorithm is superior to the one-level partition algorithm, and the one-level partition algorithm...
superior to the pen-tiling algorithm, which illustrates the importance of balancing the different schedules.

![Graphs showing the relationship between average schedule length and second memory size for IIR and DPCM](image)

**Figure 4.7.** The relationship between the average schedule length and the second memory size for IIR and DPCM

In Figures 5.5(a) and 5.5(b), the curves depict the relationship between the average schedule length and the second level memory requirement. Two different benchmarks, IIR and DPCM, are used with the first level memory constraints the same as in the second table. The $x$ axis in these two curves is the second level memory size, and the $y$ axis is the corresponding average schedule length. Around the threshold, it can be seen from the curves that a smaller second level memory size will degrade the performance greatly, while increasing the second level memory size will not result in much performance improvement. The memory size and the
corresponding second level partition size obtained from our algorithm is just this threshold to determine the two level schedule.

4.5 Conclusion

In this chapter, a new scheme that can obtain a minimal average schedule length under three levels of memory hierarchy was proposed. This algorithm not only explores the ILP among instructions by using retiming techniques, but combines it with data prefetching in both the first and second level memories to produce high throughput schedules. It uses partitions in both the first and second level memories. Through the study of the properties of the memory requirement and the schedule length in both levels, the algorithm gives a partition shape and size so that the overall minimal schedule length can be obtained. This scheme can take full advantage of the second level memory as compared to dynamic scheduling. Experiments on DSP benchmarks show that our scheme can always produce a minimal average schedule length.
CHAPTER 5

PARTITIONING WITH INITIAL DATA

The data to be prefetched can be classified into two groups, the intermediate and initial data. The intermediate data can serve as both left and right operands in the equations. Their value will vary during the computation. On the contrary, the initial data can only serve as right operands in the equations. They will maintain their value during the computation. Take the following equations as an example. The array $B$, $C$ can be regarded as the intermediate data and $A$ as the initial data.

\[
\]
\[
\]

The influence of both these two kinds of data should be deliberated in order to obtain an optimal overall schedule.

In our approach, both the intermediate and initial data are considered. For the intermediate data, we will restrict our study to nested loops with uniform data dependencies. On the other hand, it is difficult to implement uniformization for the initial data. Therefore, affine reference index is considered. The concept footprint \[1\] is used to denote the initial data needed for the computation of ALU units in one partition. Given a partition shape, this chapter presents an algorithm.
to find a partition size which can give rise to the maximum overlap between the adjacent overall footprints such that the number of memory operations is reduced to the largest extent.

The new algorithm in this chapter significantly exceeds the performance of existing algorithms [29, 87] due to the fact it optimizes both ALU and memory schedules and considers the influence of initial data. Taking the Wave Digital filter as an example, in a standard system with 4 ALU units and 4 memory units, assuming 3 initial data references exist in each iteration, our algorithm can obtain an average schedule length of 4.018 CPU clock cycles, which is very close to the theoretic lower bound of 4 clock cycles. The traditional list scheduling needs 22 clock cycles. The hardware prefetching costs 10 clock cycles. While the PSP algorithm in [29] can achieve some improvement, it still needs 8 clock cycles. Without the memory constraint, the algorithm in [87] has the same performance, 8 clock cycles. Our algorithm improves all the previous approaches.

It is worthwhile to mention that some works have been done on data layout technique [10, 70], which is used to reduce the cache coherency and conflict traffic. Our work should be regarded as another different layer which can be built upon the layer of data layout to get a better performance.

5.1 The theory about initial data

The overall footprint of one partition consists of all the initial data needed by one partition computation. Provided the execution is along the partition sequence, the initial data needed by the current partition computation have been prefetched to the first level memory at the time of previous partition. Also, the initial data needed by the next partition execution will be prefetched by the memory units.
during the current partition execution. For the overlap between the overall foot-
prints of the current and next partitions, they have already been in the first level
memory. The prefetch operations can be spared. Thus, the major concern for the
initial data is how to maximize the overlap between the overall footprints of two
consecutively executed partitions to reduce the memory traffic.

As mentioned in the introduction section, we consider affine reference for the
initial data. Given a loop index vector \( \vec{i} \), an affine reference index can be expressed
as \( \vec{g}(\vec{i}) = \vec{i}G + \vec{a} \), where \( G = [\vec{G}_1 \vec{G}_2] \) is a \( 2 \times 2 \) matrix and \( \vec{a} \) is the offset vector. The
footprint with respect to a reference \( A[\vec{g}_1(\vec{i})] \) is the set of all data elements \( A[\vec{g}_1(\vec{i})] \)
of \( A \), for \( \vec{i} \) an element of the partition. The overall footprint is the union of the
footprints with respect to all different references. For example, In Figure 5.1, the
partition is a rectangle with size \( 3 \times 4 \). The initial data references are \( B(i + j, i - j) \)
and \( B(2i + j, i - 2j) \). Their corresponding footprints are denoted by those integer
points marked by black \( X \) and \( O \), respectively. The overall footprint is the union
of these two footprints.

In [1], Anant presents the concept uniformly generated references. Two refer-
ences \( A[\vec{g}_1(\vec{i})] \) and \( A[\vec{g}_2(\vec{i})] \) are said to be uniformly generated if

\[
\vec{g}_1(\vec{i}) = \vec{i}G + \vec{a}_1 \quad \text{and} \quad \vec{g}_2(\vec{i}) = \vec{i}G + \vec{a}_2
\]

If two references \( B_1 \) and \( B_2 \) are not uniformly generated, the overlap between
footprint with respect to \( B_1 \) of the current partition and that with respect to \( B_2 \) of
the next partition can be ignored because the overlap, if exists, diminishes rapidly.
Therefore, we need only consider the overlap between footprints with respect to
uniformly generated references of two consecutive partitions. Moreover, the offset
vector \( \vec{a} \) should satisfy that \( \vec{a} = m\vec{G}_1 + n\vec{G}_2 \), where \( m \) and \( n \) are integer constants.
Otherwise, no overlap between the footprints of consecutive partitions will exist even for the uniformly generated references.

The memory requirement should be taken into account when trying to maximize the overlap. The partition size cannot be enlarged arbitrarily only for the sake of increasing overlap. In such case, the larger partition means the larger overall footprint; i.e., the much more memory space will be consumed. Therefore, given a partition shape and a set of uniformly generated references, we try to derive some conditions of the partition size which should be met to achieve a reasonable maximal overlap. For the convenience of description, we introduce the following notations.

**Definition 13**  
1. Assuming the partition size is $\vec{S}$, $f(\vec{a}, \vec{S})$ is the footprint with respect to reference with offset vector $\vec{a}$ of the current partition, and $f(\vec{a}', \vec{S})$ is the footprint with respect to reference with offset $\vec{a}'$ of the next partition.

2. Given a set of uniformly generated references, the set $R = \{\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_n\}$
is set of offset vectors. Assuming the partition size is $\vec{S}$, $F(R, \vec{S})$ is the overall footprint of the current partition and $F(R', \vec{S})$ is the overall footprint of the next partition.

Figure 5.2. One dimensional line segments

One dimensional case can be regarded as a simplification to the two dimensional problem, in which the $f_y$ is always set to zero. It provides the theoretic foundation for the two dimensional problem. In the case of one dimension, a partition is reduced to a line segment and all vectors reduce to integer numbers. The partition size can be thought of as the length of the line segment. We use an example to demonstrate the problem we are tackling. In the Figure 5.2, there are three different offset vectors: 1, 2, 7. The solid lines represent the overall footprint of the current partition, and dotted lines denote that of the next partition. Then, we need to find the condition of the partition size, i.e., the length of the line segment, to achieve a maximal overlap. The figure shows the case when the length

*Note that the elements in the set $R$ are in lexicographically increasing order.
equal 5, which is the minimum length to obtain the maximum overlap between overall footprints.

In order to derive the theorem on the minimum value $S$ which can generate the maximum overlap, we first have the following lemmas. They are used to consider the overlap of two footprints of the consecutive partitions, as show in the Figure 5. The solid line is the footprint of the current partition and the dotted line is the footprint of the next partition.

![Figure 5.3. Two different relation between $a_1$ and $a_2$](image)

**Lemma 20** The minimum $S$ is $a_2 - a_1$ which makes the maximum intersection between $f(a_1', S)$ and $f(a_2, S)$, where $a_2 \geq a_1$.

**Proof:** According to the relation between $(a_1 + S)$ and $a_2$, there are two different cases.

- **Case 1**, as shown in Figure 5.3(a): $a_1 + S \leq a_2$, i.e., $S \leq a_2 - a_1$

  The intersection of is $(a_2, a_1 + 2S - 1)$. It can reach the maximum value $a_2 - a_1$ when $S = a_2 - a_1$. 

95
• Case 2. as shown in Figure 5.3(b): $a_1 + S > a_2$, i.e., $S > a_2 - a_1$

The intersection of two segments is $(a_1 + S, a_2 + S - 1)$. It has no relation to $S$. This means the size of intersection will not increase in spite of the increment of $S$. \hfill \Box

**Lemma 21** For the intersection between $f(a_1', S)$ and $f(a_2, S)$, where $a_2 \geq a_1$, it will keep constant, irrelevant to the value of $S$, as long as $S \geq a_2 - a_1$.

According to the definition, The $F(R, S)$ and $F(R', S)$ can be expressed as following.

$$F(R, S) = f(a_1, S) \cup f(a_2, S) \cdots \cup f(a_n, S)$$

$$F(R', S) = f(a_1', S) \cup f(r_2', S) \cdots \cup f(r_n', S)$$

The following lemma gives the expression of their intersection.

**Lemma 22** Let $C_m$ be the intersection $f(a_m, S) \cap f(a_{m-1}', S)$. Then the intersection of $F(R, S)$ and $F(R', S)$ is $\bigcup_{2}^{n} C_m$, where the number of integers in $R$ is $n$.

**Proof:** Let $A_m$ denote $f(r_m, S)$, and $B_m$ denote $f(r_m', S)$.

**Basis Step.** Let $n=2$. Then $F(R, S) = A_1 \cup A_2$ and $F(R', S) = B_1 \cup B_2$. The ending point of $A_1$ is less than the starting point of $B_1$ and $B_2$. The starting point of $B_2$ is greater than the ending point of $A_1$ and $A_2$. Thus, the only possible intersection is $A_2 \cap B_1$.

**Induction Hypothesis** Assume that, for some $n \geq 2$, $F(R, S) \cap F(R', S) = \bigcup_{2}^{n} C_n$. 

96
**Induction Step.** For $n + 1$, the added intersection is $A_{n+1} \cap (B_1 \cup B_2 \cup \cdots \cup B_n)$.

There are two different cases.

1. $a_{n+1} \geq (a_n + S)$. Then $A_{n+1}$ can only intersect with $B_n$.

2. $a_{n+1} < (a_n + S)$. Then $A_{n+1}$ can be divided into two parts, $A' = (a_{n+1}, a_n + S)$ and $A'' = (a_n + S, a_{n+1} + S - 1)$.

\[
A_{n+1} \cap (B_1 \cup B_2 \cup \cdots \cup B_n) = A' \cap (B_1 \cup B_2 \cup \cdots \cup B_n) \cup A'' \cap (B_1 \cup B_2 \cup \cdots \cup B_n) \subseteq \bigcup_2^n C_n \cup (A_{n+1} \cap B_n) = C_{n+1}
\]

Therefore, $F(R, S) \cap F(R', S) = \bigcup_2^{n+1} C_n$.

\[\square\]

**Theorem 23** Given the set $R = (a_1, a_2, a_3, \ldots, a_n)$, the maximum intersection between $F(R, S)$ and $F(R', S)$ can be achieved when $S = \max_{m=2}^n (a_m - a_{m-1})$.

**Proof:** When considering two adjacent $C_m$ and $C_{m-1}$, we have $C_m = A_m \cap B_{m-1}$, $C_{m-1} = A_{m-1} \cap B_{m-2}$. There is no common element between $B_{m-1}$ and $A_{m-1}$, neither is $C_m$ and $C_{m-1}$. According to the lemma 20 and lemma 21, the value $x \geq r_m - r_{m-1}$ can make segment $C_m$ largest. Moreover, each $C_m$ will not intersect each other. Therefore, the theorem is correct. \[\square\]

From Theorem 23 and Lemma 21, we can directly derive the following theorem.
**Theorem 24** For the overall footprints \( F(R, S) \) and \( F(R', S) \), their overlap will keep constant if the value of \( S \) continues to increase from the \( S \) value obtained by the Theorem 23.

![Figure 5.4. The stripe division of a footprint](image)

To maximize the overlap between \( F(R, \vec{S}) \) and \( F(R', \vec{S}) \) in the two dimension space, we can find that \( f_y \) element of the partition size is not so important as \( f_x \) element, since the intersection always increases when \( f_y \) is enlarged. We will determine the value of \( f_y \) based on other conditions. Therefore, the key is what is the minimum value of \( f_x \) to make the intersection maximum, given a certain \( f_y \).

At following, we discuss the situation with \( G \) a two dimensional identity matrix. If \( G \) is not an identity matrix, the same idea can be applied as long as \( \vec{a} = m\vec{G}_1 + n\vec{G}_2 \). The only difference is that the original \( XY \) space will be transformed to the new space by \( G \) matrix. An augment set \( R^* \) can be obtained based on a certain partition size of \( \vec{S} \) and the set \( R \) with the following method: \( a^*_i = a_i, \ a^*_{i+n} = a_i + f_yP_y.y, \) where \( n \) is the size of set \( R \) and
\( P_y = (P_y.x, P_y.y) \). Arranging all the points in set \( R^* \) with the increasing order of the \( Y \) element, the overall footprint of one partition can be divided into a series of stripes. Each stripe is determined by two horizontal lines which pass the adjacent two points in sorted \( R^* \). For instance, in the Figure 5.4, the \( R \) set is \{\((0, 0), (6, 1), (3, 2), (1, 3)\)\}. Assume the value of \( f_{yP_y.y} \) is 5, then the augment set \( R^* \) is \{\((0, 0), (0, 5), (6, 1), (6, 6), (3, 2), (3, 7), (1, 3), (1, 8)\)\}. After sorting, it will become \{\((0, 0), (6, 1), (3, 2), (1, 3), (0, 5), (6, 6), (3, 7), (1, 8)\)\}. The overall footprint consists of 7 stripes as indicated in the figure.

In each stripe, a horizontal line will intersect with left bounds of some footprints \( f(\vec{a}, \vec{S}) \). Thus, the two dimensional intersection problem of this stripe in the footprint can be reduced to one dimensional problem, which can be solved using the Theorem 23. Applying this idea to each stripe, we can solve the two dimensional overlap problem, as demonstrated in the following algorithm. The algorithm is obvious a polynomial-time algorithm, whose time complexity is \( O(n^2) \).

From the Lemma 21, the intersection will keep constant if \( f_x \) is greater than the value chosen by this algorithm, and will reduce with less \( f_x \). We can demonstrate this phenomenon by two examples. The set \( R \) for the first example is \{\((0, 1), (5, 3), (3, 1), (4, -1), (2, -2)\)\} and the partition shape is \((1, 0) \times (0, 1)\). It is the partition shape for Wave Digital filter. The set \( R \) for the second example is \{\((0, 2), (3, 5), (1, 3), (-1, -1)\)\} and the partition shape is \((1, 0) \times (-3, 1)\). It is the partition shape for Two Dimensional filter. The Figure 5.5(a) and 5.5(b) show the varying trends of footprint intersection with the value of \( f_x \) and \( f_y \) for two examples, respectively.
Algorithm 5 Calculating the minimum $x$ to make the overlap maximum

**Input:** The set $R$ and the shape of the partition

**Output:** The $f_x$ to make the overlap maximum under a certain $f_y$

1. Set $f_x$ to 0.
2. Based on the set $R$ and partition shape, choose a $f_y$ such that the product $f_y * P_y,y$ is larger than the difference between the largest and least $b$ element of all vectors in set $R$.
3. Using the $f_y$ above, generate the augment set $R^*$
4. Sort all the value in the $R^*$ in increasing order according to the $b$ element and kept them in a event list.
5. Use a horizontal line to sweep the whole iteration space. When an event point is met, insert the corresponding set $f(\vec{a},\vec{S})$ in a visiting list, if the event point is the lower bound of the footprint. Otherwise delete the corresponding $f(\vec{a},\vec{S})$ from the list.
6. Calculate the intersection point of this line with the left bound and right bound of each set in the visiting list, respectively. Use Theorem 23 to derive a $f'_x$ value to make the intersection in current stripe maximal.
7. Replace $f_x$ with $f'_x$ if $f'_x > f_x$.

---

(a) 2D

(b) WDF

Figure 5.5. The tendency of intersection with $f_x$ and $f_y$
5.2 The overall schedule

The overall schedule can be divided into two parts – ALU and memory schedules, as seen in Figure 5.6. The ALU part schedules the ALU computation. The memory part schedules the memory operations – prefetch and keep, so that the data for the computation can always be found in the first level memory. For the ALU schedule, the multi-dimensional rotation scheduling algorithm [58] is used to generate a static schedule for one iteration. Then the entire ALU schedule can be formed by simply replicating this schedule for each iteration in the partition. The schedule obtained by this way is the most compact schedule since it only considers the ALU hardware resource constraints. The overall schedule length must be longer than it. Thus, this ALU schedule provides us a lower bound for the overall schedule. This lower bound can be calculated by \#len_{iteration} \times \#nodes, where \( len_{iteration} \) represents the schedule length obtained by multi-dimensional rotation scheduling algorithm for one iteration, and \#nodes denotes the number of iteration nodes in one partition. Our objective is to find a partition whose overall schedule length can be very close to this lower bound.
5.2.1 Balanced overall schedule

Different from the ALU schedule, the memory schedule is considered as an integrate for the entire partition. It consists of two parts: memory operations for initial data and intermediate data. Each part consists of the prefetch and keep operations for the corresponding data. Because all the prefetch operations have no relation to the current computation, they can be arranged from the beginning of the memory schedule part. On the contrary, the keep operation for intermediate data can only be issued after the corresponding computation has finished. The keep operations for initial data can be issued as soon as they have been prefetched. The memory part schedule length is the summation of these two parts’ schedule lengths.

For the intermediate data, the calculation of the number of prefetch and keep operations can refer to [87]. For the initial data, they can be prefetched in blocks. This kind of operation can fetch several data at one time and costs only a little longer time than general prefetch operation. To calculate the number of such operations, we first have the following observation.

Property 4 As long as \( f_y P_y G_2 \), the projection of footprint size along the direction \( G_2 \), is larger than the maximum difference of \( \bar{a} G_2 \), \( \forall \ \bar{a} \) belongs to a uniformly generated offset vector set, the overall footprint will increase at a constant rate with the increment of \( f_y \), so does the number of prefetch operations for initial data.

Note the requirement in the above property guarantees that the partition is large enough, such that the footprint with respect to a offset vector can intersect with the footprint with respect to all other offset vectors belonging to the same uniformly generated set.
Suppose a two dimensional vector can be written as $\vec{a} = (a.x, a.y)$. Given a certain $f_x$, the number of prefetch operations for initial data for any $f_y$, which satisfying the condition in the above property, is $Pre_{Base, ini} + (f_y - f_{y_0}) \times Pre_{incr, ini}$, where $f_{y_0} = \left\lceil \frac{y_0}{(P_y G)_y} \right\rceil$, $y_0$ is the maximum difference of $(\vec{a}G)_y$ for all offset vectors, $Pre_{Base, ini}$ denotes the number of such operations for a partition with size $f_x \times f_{y_0}$, and $Pre_{incr, ini}$ represents the increment of number of prefetch operations when $f_y$ is increased by one.

The keep operations for the initial data can be issued after they have been prefetched. The number of such keep operations is $Keep_{Base, ini} + (f_y - f_{y_0}) \times Keep_{incr, ini}$, where $y_0$ and $f_{y_0}$ have the same meaning as above. $Keep_{Base, ini}$ denotes the number of keep operations for a partition with size $f_x \times f_{y_0}$, and $Keep_{incr, ini}$ represents the increment of keep operations when $f_y$ is increased by one.

In order to understand what is a good partition size, we first need the definition of the balanced overall schedule. It is also gives the balanced overall schedule requirement.

**Definition 14** A balanced overall schedule is a schedule for which the memory schedule is at most one unit time of keep operation longer than the ALU schedule.

To reduce the computation complexity and simplify the analysis, we add a restriction on the partition size: the partition size is large enough that no data dependence can span more than two partitions.

1. There is no delay dependency which can span more than two partitions along the $Y$ coordinate direction, that is $f_y \times P_y.y \geq d_y, \forall d = (d_x, d_y) \in D$. 

103
2. There is no delay dependency which can span more than two partitions along the X coordinate direction, that is \( f_x > \max\{d_x - d_y P_{y,x} \} \).

As long as these constraints on minimal partition size are satisfied, the length of prefetch and keep parts for intermediate data in memory schedule increases slower than the ALU schedule length when partition size is enlarged. At this time, if a partition size cannot be found to meet the balanced overall schedule requirement, it means that the length of the block prefetch part for initial data increases too fast. Due to the property of block prefetch, increasing \( f_x \) will increase the number of block prefetch only by a small number, while increase the ALU part by a relative large length. Therefore, a partition size which satisfy the balanced overall schedule requirement can be found. The following algorithm determine the partition size to obtain the balanced overall schedule.

After the optimal partition size is determined, the operations in ALU and memory schedules can be easily arranged. For the ALU part, it is the duplication of the schedule for one iteration. For the memory part, the memory operations for initial data are allocated first, then are the memory operations for intermediate data, as we discussed above.

The memory requirement for a partition consists of four parts, the memory requirement for the calculation of in-partition data, the memory for prefetch operations for intermediate data, the memory for keep operations for intermediate data and the memory for those operations for initial data. The memory consumption calculation for in-partition data can refer to [87]. For the other part memory requirements, they can be computed simply by multiply the number of operations with the memory requirement of each operation. The memory requirement for a prefetch operation is 2. One is used to store the data prefetched by
Algorithm 6 Find a balanced overall schedule

**Input:** The ALU schedule for one iteration, the partition shape \( P_x \times P_y \) and the initial data offset vector set \( R \)

**Output:** A partition size which can generate a balanced overall schedule

1. Based on the information of initial data, use algorithm 1 to calculate the minimum partition size \( f'_x \) and \( f'_y \)
2. Using two above conditions on partition size to calculate another pair of minimum \( f''_x \) and \( f''_y \)
3. Get a new pair \( f_x = \max(f'_x, f''_x) \) and \( f_y = \max(f'_y, f''_y) \).
4. Using this pair \( (f_x, f_y) \), calculate the number of prefetch operations, block prefetch operations and keep operations.
5. Calculate the ALU schedule length to see if the balanced overall schedule requirement is satisfied.
6. If it is satisfied, this pair \( (f_x, f_y) \) is the partition size. Otherwise, increase \( f_x \) by one, use the balanced overall schedule requirement to find the minimum \( f_y \). If such \( f_y \) does not exist, continue increasing \( f_x \) until the feasible \( f_y \) is found. Use them as partition size.
7. Based on the partition size, Output the corresponding ALU part schedule and memory part schedule

the previous partition and consumed in the current partition, the other stores the data prefetched by the current partition and consumed in the next partition. As the same rule, the keep operation will take 2 memory locations, too. The block prefetch operations will take \( 2 \times \text{block\_size} \) memory locations.

5.3 Experiment

In this section, we use several DSP benchmarks to illustrate the effectiveness of our new algorithm. They are WDF, IIR, DPCM, 2D and Floyd, as indicated in the tables, which stand for *Wave Digital filter, Infinite Impulse Response filter, Differential Pulse-Code Modulation device, Two Dimensional filter and Floyd-Steinberg algorithm*, respectively. These are DSP filters in common usage in real DSP applications. We applied five different algorithms on these benchmarks: list scheduling, hardware prefetching scheme, partitioning algorithms in \([29, 87]\) and
our new partition algorithm (Since it has been shown in [87] that loop tiling technique cannot outperform partitioning algorithms, we don’t compare the result of loop tiling in this section). In list scheduling, the same architecture model is used. However, the ALU part uses the traditional list scheduling algorithm, and the iteration space is not partitioned. In hardware prefetching scheduling, we use the model presented in [6]. In this model, whenever a block is accessed, the next block is also loaded. The partitioning algorithms in [29, 87] assume the same architecture model as ours. They partition the iteration space and execute the entire loop along the partition sequence. However, they do not take into account of the influence of the initial data.

In the experiment, we assume a ALU computation, a keep operation of one clock cycle, a prefetch time of 10 CPU clock cycles and a block prefetch time of 16 CPU clock cycles, which is reasonable when the big performance gap between CPU and the main memory is considered. The first table presents results with only one initial data with the offset vector (1, 1), and the second table is results with three initial data with the offset vector set \{(1, 1), (2,-2), (0,3)\}. Note all these three initial data references are uniformly generated. From the discussion in Section 4, the overall footprint is only the simple summation of the footprint with respect to different uniformly generated reference sets. In the tables, the par vector column determines the partition shape. The list column lists the schedule length for list scheduling and the improvement ratio our algorithm can get compared to list scheduling. The hardware column lists the schedule length for hardware prefetching and our algorithm’s relative improvement ratio. Since the algorithm in [29] will get the same result as the algorithm in [87] when there is no memory size constraint, we merge their results into one column Partition algo.
### Table 5.1

**Experimental Results with Only One Initial Data**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Par Vector</th>
<th>New algo</th>
<th>Partition algo</th>
<th>list</th>
<th>hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Px Py</td>
<td>size</td>
<td>m_r len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDF</td>
<td>(1,0) (-3, 1)</td>
<td>4 x 7</td>
<td>221 4.107</td>
<td>4 x 4</td>
<td>143 5.312</td>
</tr>
<tr>
<td>IIR</td>
<td>(1,0) (-2, 1)</td>
<td>4 x 9</td>
<td>407 6.028</td>
<td>4 x 7</td>
<td>350 6.893</td>
</tr>
<tr>
<td>DPCM</td>
<td>(1,0) (-2, 1)</td>
<td>8 x 10</td>
<td>736 4.01</td>
<td>8 x 8</td>
<td>628 4.891</td>
</tr>
<tr>
<td>2D</td>
<td>(1,0) (0,1)</td>
<td>3 x 5</td>
<td>233 12</td>
<td>3 x 4</td>
<td>207 12</td>
</tr>
<tr>
<td>Floyd</td>
<td>(1,0) (-3,1)</td>
<td>7 x 5</td>
<td>301 6.057</td>
<td>4 x 4</td>
<td>174 6.312</td>
</tr>
</tbody>
</table>
### TABLE 5.2

**EXPERIMENTAL RESULTS WITH THREE INITIAL DATA**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>par Vector</th>
<th>New algo</th>
<th>Partition algo</th>
<th>List</th>
<th>hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vx</td>
<td>Vy</td>
<td>size</td>
<td>m, r</td>
<td>len</td>
</tr>
<tr>
<td>WDF</td>
<td>(1,0)</td>
<td>(-3, 1)</td>
<td>8 x 7</td>
<td>474</td>
<td>4.018</td>
</tr>
<tr>
<td>IIR</td>
<td>(1,0)</td>
<td>(-2, 1)</td>
<td>5 x 13</td>
<td>772</td>
<td>6.015</td>
</tr>
<tr>
<td>DPCM</td>
<td>(1,0)</td>
<td>(-2, 1)</td>
<td>8 x 14</td>
<td>1207</td>
<td>4.001</td>
</tr>
<tr>
<td>2D</td>
<td>(1,0)</td>
<td>(0,1)</td>
<td>4 x 5</td>
<td>346</td>
<td>12</td>
</tr>
<tr>
<td>Floyd</td>
<td>(1,0)</td>
<td>(-3,1)</td>
<td>8 x 6</td>
<td>526</td>
<td>6</td>
</tr>
</tbody>
</table>
In the Partition algo and new algo columns, the size column is the size of partition presented with the multiple of partition vectors. The $m_r$ column represents the corresponding memory requirement and the $len$ column is the average scheduling length for corresponding algorithms. The ratio column is the improvement our new algorithm can get relative to the corresponding algorithms.

The list scheduling and hardware prefetching schedule the operations based on the iteration, which will result in the much longer memory schedule. It is this dominant memory schedule that leads to an overall schedule which is far away from the balanced schedule. Thus, lots of ALU resources are wasted waiting for the data. Their much worse performance compared with the partitioning technique can be seen from the tables.

Although the traditional partitioning algorithms consider the balance of ALU and memory schedules for intermediate data, they lack of the consideration for the initial data. The time consumption to load the initial data is a rather significant influence factor for one partition. The lack of such consideration will result in an unbalanced overall schedule. The memory latency cannot be efficiently hidden. This is the reason why traditional partitioning algorithms get the worse performance than our new algorithm. It also explains the results that the performance will become worse as the initial data references increase. Our new algorithm considers both data locality and the initial data. Therefore, the much better performance can be achieved through balancing the ALU part and memory schedule.
5.4 Conclusion

In this chapter, a new scheme that can obtain a minimal average schedule length under the consideration of initial data was proposed. The theories and an algorithm on initial data were presented. The algorithm explores the ILP among instructions by using software pipelining techniques and combines it with data prefetching to produce high throughput schedules. Experiments on DSP benchmarks show that our scheme can always produce a better average schedule length than existing methods.
CHAPTER 6
PARTITIONING WITH MULTIPLE LOOP NESTS

It is usual that a program includes several loop nests in the consecutive order. A lot of computational overhead is caused by the repetitive access to array data elements. Separately partitioning each loop will not take full advantage of the data reuse occurred among different loop nests, thereby incur much more memory references than multiple loop partition. This is demonstrated by the experiments in Section 5. Therefore, the traditional loop partition technique is not efficient for multiple loop nests. Multiple loop partition technique deals with multiple loop nests together and partitions them at the same time, such that the data locality can be exploited to the largest extent.

An important problem related to group computation in the loop nest is that much more cache interference may be incurred. Chame and Moon [12] propose a new tile selection algorithm for eliminating self interference and simultaneously minimizing capacity and cross-interference misses. Data padding, which changes the array store sequence in memory, is an effective technique to address the cache interference problem. Rivera and Tseng [69] develop and evaluate heuristics for guiding two data-layout transformations, inter- and intra-variable padding. Data padding technique is utilized in multiple loop partition to eliminate the cache conflict when the first level memory is a direct mapped cache.
6.1 Program model

Multiple loop nests are represented by Loop Dependence Graph (LDG), as defined below.

**Definition 15** A loop dependence graph (LDG) $G = (V, E, D_L)$ is an edge-weighted directed multigraph, where $V$ is the set of loop nests, $E \in V \times V$ is the set of dependence edges between the loop nests, and $D_L$ is a function from $E$ to $\mathbb{Z}^n$, representing the set of loop dependency vectors between two nodes, where $n$ is the number of loop dimension.

![Diagram of LDG](a) LDG

![Program segment](b) Program segment

**Figure 6.1.** The code and its LDG
The LDG is a directed multigraph because multiple dependences can exist from one loop nest to another arising from different variables. Figure 6.1(a) shows an example of LDG and Figure 6.1(b) shows the corresponding code segment. In this example, \( V = \{A, B, C, D\} \) and \( E = \{e_1, e_2 : (A, B), e_3, e_4 : (B, C), e_5 : (C, C), e_6 : (C, D), e_7 : (A, C), e_8 : (D, A)\} \) where, \( D_L(e_1) = (2, 1), D_L(e_2) = (1, 1), D_L(e_3) = (0, 1), D_L(e_4) = (0, -2), D_L(e_5) = (1, 0), D_L(e_6) = (0, -1), D_L(e_7) = (0, 1), D_L(e_8) = (2, 1) \). Note that multiple edges exist between nodes A, B and B, C, due to the different array elements access. The idea of multiple loop partition is illustrated with multiple two-dimensional nested loops. The weight of edges in the corresponding LDG can be written as \( D_L = (D_L[1], D_L[2]) \). The two different target program models are shown in the Figures 6.2(a) and 6.2(b), respectively.

```
DO K1 i = 0, n
  DO K2 j=0, m
    ...
  K2 CONTINUE
  DO K3 j = 0,m
    ...
  K3 CONTINUE
  ...
K1 CONTINUE
```

```
(a) Nested Multiple Loops

DO K1 i = 0, n
  DO K2 j=0, m
    ...
  K2 CONTINUE
  DO K3 j = 0, m
    ...
  K3 CONTINUE
  ...
K1 CONTINUE
```

(b) Multiple Nested Loops

Figure 6.2. Program Model
Both two models comprise of several uniform loop nests without the intervening code between them. All the loop nests are conformable with each other. Conformability is a binary equivalence relation. Two loop nests are said to be conformable if their corresponding loops have the same type (serial in this chapter) and identical iteration space size (loop bounds). The data dependences between loop nests are constant, as well as lexically forward (from the earlier loop to the later). Program transformations may be used to obtain a program segment that belongs to the above models. For instance, code motion may be employed to obtain a sequence of loops with no intervening code. Loop permutation may be used to ensure that loop nests are conformable.

The difference between two models is that multiple sequential loops have the same outer-most loop in the nested multiple loops program model, while the loops in the multiple nested loops model are relatively independent. Thus, there will be some loop-carried data dependence introduced by the outer-most loop in the first model, which may cause a dependence cycle in the corresponding LDG. To guarantee the correctness of the execution order, there should not exist the dependence that flows backwards with respect to the iteration execution order. A set of nested multiple loops is illegal if no execution order exists to satisfy all the data dependence. The following lemma ensure the legality in the first program model.

**Lemma 25** A set of nested multiple loops is legal if each cycle in its corresponding LDG satisfy $W(c) \geq (1, -\infty)$ * and $D_L(e)[1] >= 0, \forall e \in E$, where $W(c)$ is the weight summation for all edges in a cycle.

*In this chapter, all the comparison between two vectors is based on the lexicographic order*
On the contrary, there is no data dependence from a loop nest to its ancestor in the second program model. Its LDG is always acyclic. The existence of cycle in LDG brings a little more complication on the in-partition execution order, which is discussed in the next section.

6.2 Partitioning multiple loop nests

In case of partitioning multiple loop nests, a partition consists of several sub-partitions. Figure 6.3 plots an example of a single partition. In a partition, all the iterations which belong to the same iteration space constitute a subpartition. In this figure, $L_1$, $L_2$ and $L_3$ are the iteration spaces for the first, second and third loop nests, respectively. $SP_1$, $SP_2$ and $SP_3$ are the subpartitions in $L_1$, $L_2$ and $L_3$, respectively.

![Diagram of partition and subpartitions](image)

Figure 6.3. Partition and subpartitions

Multiple loop partition exploits not only the data locality inside each iteration space, but also the data locality among different loop nests. Taking Figure 6.4 as
an example, $SP1$ and $SP2$ represent the subpartitions in iteration space $L1$ and $L2$, respectively. The dark circles denote the iterations. Iterations $I1$, $I2$, and $I3$ lie in $SP1$, while iterations $I4$ and $I5$ lie in $SP2$. Therefore, data dependences $d1$ and $d2$ belong to the dependences inside a single loop nest, $d3$ and $d4$ belong to the dependence between different loop nests. Assuming iterations $I2$ and $I3$ are in different rows, so are iterations $I4$ and $I5$. They all need the outcome from the computation of iteration $I1$. According to the general row-wise order, the result of iteration $I1$ will be fetched from the memory for 4 times. On the contrary, this result is still in the first level memory and no fetch is needed with the multiple loop partition technique. With the row-wise partition execution order, differently grained in-partition execution order can be applied. The coarse-grained in-partition execution order is shown in Fig 6.5(a). The first subpartition is finished first, then the second subpartition, until the last subpartition’s execution ends. On the other hand, we can execute the iterations alternatively in

![Figure 6.4. Different data dependences](image)

Figure 6.4. Different data dependences
the loop nests. Figure 6.5(b) shows this fine-grained in-partition execution order. After the execution of the first line of iterations in each subpartition along loop sequence, we jump to the first subpartition and begin to execute the second line. A partition is finished until all the lines of iterations are executed in this manner.

![Diagram of coarse-grained and fine-grained execution orders](image)

Figure 6.5. In-partition execution order

The first principle of the selection of in-partition execution order is that such order cannot violate the data dependence. For the multiple loop nests program model, both orders are feasible because of the acyclic lexically forward dependence. This is not the case for the nested multiple loops program model. Some data dependence cannot be satisfied by the coarse-grained execution order due to the cycle in LDG.

There are also some other considerations on the in-partition execution order. For example, different efficiency can be achieved for the different data access patterns in the loops. The coarse-grained order performs better when each loop
nest has its own input data set, while fine-grained order is preferred when the loop
ests have a common input data set. In such case, some compiler switches can be
ed to designate a better in-partition execution order.

6.3 Multiple loop partition scheduling

Multi-dimensional retiming technique [59] is used in multiple loop partition
to transform the iteration space, such that the iteration space can be partitioned
legally. A multi-dimensional retiming $r$ is a function that redistributes the nodes in
the iteration space. A new LDG is created after the retiming. The retiming vector
$r(u)$ of a node $u \in V$ represents the offset between the original iteration containing
$u$, and the one after retiming. The weight of the edge changes accordingly to
preserve dependence, i.e., $D^r_L(u, v) = D_L(u, v) + r(u) - r(v)$, where $D^r_L$ and $D_L$
is the weight after and before retiming, respectively. Also, $D^r_L(c) = D^r_L(c)$ for each
cycle $c \in LDG$.

In the nested multiple loops program model, it is stated in Lemma 25 that
$D_L[1]$ element of the weight in LDG should be no less than zero for any edge in
LDG. Provided the fine-grained execution order is used, there are two cases for the
data dependences. If $D_L[1] > 0$ or $D_L[1] = 0, D_L[2] \geq 0$, these data dependences
can be preserved using the fine-grained execution order. If $D_L[1] = 0, D_L[2] < 0$,
it is reverse to the flow of execution along $X$ axis and will be violated after the
partition. Therefore, we should eliminate all the data dependence that belong to
the second case. This can be accomplished by the retiming technique [59] proved
by the following theorem.

**Theorem 26** Given a legal LDG, there exists a set of retiming vector $S(\vec{r})$ such
that a legal retimed LDG is obtained with $d_i \geq (0, 0)$ for any edge in this graph.
Proof: Assuming an edge $e : u \rightarrow v$, its weight after retiming is $D'_L = D_L + r(u) - r(v)$. The inequality $r(v) - r(u) \leq D_L$ must be true to make $D'_L \geq (0, 0)$. Therefore, a feasible solution of the following integer programming problem can serve as a retiming vector set, where $r_n$ denote the retiming vector on $n^{th}$ node.

\[
\text{ILP:} \quad r_1 - r_2 \leq D_L(1,2) \\
r_2 - r_3 \leq D_L(2,3) \\
\vdots \\
r_{n-1} - r_n \leq D_L(n,n-1)
\]

This special IP problem can be solved using graph theory. A directed graph can be constructed with each node denoting a variable, and edges between the corresponding nodes whose weight equal to $D_L$. An additional source node is also added. This node has a zero weight edge to any node in the original graph. If there is no cycle with weight less than $(0, 0)$, The single source shortest path is a feasible solution. Lemma 25 has shown that for all cycle in the graph, $W \geq (1, -\infty)$, which ensure the existence of a feasible solution.

\[\square\]

A retiming vector set can be obtained using Bellman-Ford single source shortest path algorithm. The only modification is to replace all the weights, real numbers in the original algorithm, by weight vectors i.e., $D_L$ in the LDG.

In the multiple loop nests program model, there is no certainty that $D_L[1] \geq 0$. The data dependence with $D_L[1] < 0$ will also be violated after the partition. This violation can still be overcome by retiming. Because the LDG for this program model is acyclic, the following algorithm can be used to get rid of all of the violations.

After applying the retiming vectors obtained above, all the data dependence
Algorithm 7 Find a feasible retiming vector set

Input: An LDG graph
Output: A retiming vector set of LDG

Sort the nodes in topological order
FOREACH Node $i$ in LDG do
  $r(i) = (0, 0)$
ENDFOR

FOREACH Node $i$ in LDG do
  Construct a vector $NV = (NV[1], NV[2])$, which satisfy $NV[1] \leq DL[1], NV[2] \leq DL[2], \forall$ incoming edges with weight $DL$.
  IF $NV \geq 0$ THEN $r(i) = (0, 0)$ ELSE $r(i) = NV$ ENDIF
  Update all the outgoing edges’ weight to $DL' = DL + r(i)$.
ENDFOR

satisfy $DL \geq (0, 0)$, thereby are automatically preserved by the partition execution order. The partition shape is only determined by the data dependence $d_e$ in the MDFG of each loop nest. Therefore, we can find a legal subpartition shape for each loop based on Lemma 1 in Chapter 3. Due to the lexically forward data dependence, vector $(1, 0)$ can always serve as $P_x$ for each subpartition, thereby the $P_x$ of the partition. As about $P_y$, it is one of $P_y$ of all the subpartitions, which has the largest angle with vector $(1, 0)$. It is not difficult to verify that all the data dependences conform to Lemma 1 of Chapter 3 under such partition shape.

As the same principle of partitioning an individual nested loop, the determination of the partition size depends on the relation between the memory and ALU schedules. A partition size which can balance the ALU and memory schedules is a preferred size. To derive the memory schedule, all the prefetching operations need to be identified.

The number of prefetching operations depends on the data dependence distance and the partition size. A certain data dependence decide a prefetching region, in which all data need to be prefetched in advance. Figure 6.6 shows two adjacent partitions along the partition execution direction and two different data
dependences. $DL$ is a data dependence between two loop nests, and $de$ is a dependence inside a loop. Each of them decide a prefetch region in the partition, as shown in the figure.

Figure 6.6. Prefetch and keep regions

In the algorithm to decide the partition size, a partition size is denoted by $f_x$ and $f_y$. Then the actual partition size is $f_x$ along $X$ axis and $\|f_y \times P_y\|$ along the $P_y$ direction. The lengths of memory and ALU schedules are denoted by $L_{mem}$ and $L_{ALU}$ respectively. The first “for” loop in the algorithm calculates the minimum partition size requirement. This requirement means that no data dependence will span two or more partitions. This constraint is used to reduce the computation and simplify the analysis. The “while” loop in the algorithm calculates the partition size which can achieve a balanced partition schedule. The theory behind these steps is: increase the partition size along $P_y$ direction can maintain the number of prefetch operation. If the memory schedule is longer than ALU schedule at the start step, enlarge the partition along $P_y$ direction.
only increases the number of iterations in the partition, which implies the ALU schedule will increase faster than the memory schedule if the partition size along $P_x$ direction is large enough, thereby guarantees the existence of the balanced partition schedule and the termination of the algorithm. In the algorithm, we use the variable “icount” and a predefined constant “MI” to ensure the $f_x$ is large enough to find a balanced partition schedule.

Another property of a balance partition schedule can be seen from the algorithm. From the partition size found by the algorithm, decreasing the partition size will lead to the longer memory schedule than the ALU schedule. On the other hand, increasing the partition size furthermore can make the difference between the memory and ALU schedules larger. Therefore, this partition size determined by the algorithm is a best balance point between the memory and ALU schedules.

We prefer a small partition size for the convenience of data padding, which is discussed in the next section.

### 6.4 Data padding

In the above discussion, the first level memory is always assumed full-associative. In the case of the low-associativity cache, some data with different memory addresses may map to the same cache location. The references to these data will lead to conflict misses. Data padding is an effective technique to eliminate such cache interference. It can be classified as inter- and intra-variable padding. Inter-variable padding is applied to several variables. Some gaps are inserted between different variables to eliminate the cross interference among these variables. On the other hand, intra-variable padding is applied to a single variable so as to eliminate the self-interference.

Figure 8 is the example of inter- and intra-variable padding. In Figure 7.3(a),
Algorithm 8 Multiple loop partition scheduling

**Input:** The data dependences $d$ (including both $d_e$ in MDFG and $D_L$ in LDG) in multiple loop nests

**Output:** The partition size
/*find the partition shape*/
For each loop nest, find the subpartition shape $P_{x,s}, P_{y,s}$
Set $P_x = (1, 0)$. Choose $P_y$ as the most anti-clockwise vector relative to $P_x$ in all $P_{y,s}$.
/*find the minimum $f_x$ and $f_y$/
$f_x = 0, f_y = 0$.
FOREACH $d = (d[1], d[2])$ /* vector $d$ can be regarded as $d_e$ or $D_L$ */ do
  if $f_x < (d[1] - d[2] * p_y[1]/p_y[2])$ then
  end if
    $f_y = \lceil d[2]/p_y[2] \rceil$
  end if
ENDFOR
/*determine the partition size*/
Calculate $L_{mem}$ and $L_{ALU}$ under the partition size $f_x, f_y$.
icount = 0
WHILE $L_{ALU} \geq L_{MEM}$ do
  $f_y ++$;
icount ++;
  Calculate $L_{mem}$ and $L_{ALU}$ under the partition size $f_x, f_y$.
  if icount $\geq$ MI then
    $f_x ++$;
  end if
ENDWHILE

Schedule the ALU and memory operations in ALU and memory units, respectively.
unit-stride references to $A(i)$ and $B(i)$ provide spatial locality, leading to cache 
reuse. However, if $A$ and $B$ are separated by a multiple of the cache size and the 
cache is direct mapped, every access to $B(i)$ will map to the same cache line as 
the previous $A(i)$ access and vice versa. As a result, every reference will produce 
a conflict miss. To regain the benefits of spatial reuse, inter-variable padding may 
be applied as shown in the figure. In Figure 7.3(b), the computation exhibits much 
spatial and temporal reuse between the references to $A$. However, if the column 
size of $A$ is a multiple of the cache size, columns of $A$ will conflict, eliminating 
reuse between references to $A$. To regain spatial reuse, intra-variable padding 
applied to the columns of $A$ can change its internal layout so that columns no 
longer map to conflicting cache locations.

![Figure 6.7. Data padding illustration](image)

It is worth noting that our partitioning technique is different from the tradi-
tional loop tiling technique, which majorly applies the optimization on the source program. Such high level optimization will bring the problem that the small tile size is not applicable, since overhead due to tiling cannot be compensated by the benefit from tiling. Instead, our partitioning technique is computational finer-grained than the tiling techniques. It has the detailed schedules for the function units, such that the compiler can directly map the schedule to operations on the corresponding function units. In such case, we can avoid the overhead incurred by the partition as well as possess the benefit of the small size partition. With the small size partition, the task to determine a data padding pattern without cache conflict becomes much easier.

![Diagram](image)

Figure 6.8. Data set for different kinds of data

(a) input only  
(b) recurrence

To apply data padding, the data set which is kept in the cache during the time of a partition’s execution should be determined first. This data set comprises
of the prefetched data, which are used in the current partition and prepared during the time of last partition’s execution, and the prefetching data, which are prepared in the cache for the next partition’s execution. When a single array is considered, the determination of the data set depends on the characteristic of this array, whether this data array only serves as the input data. If it is only the input (not modified during the computation and can only appear as right operand), the data set is shown in Figure 6.8(a). Given the current partition with left-lower corner at \((i, j)\), each data dependence on this array will decide a parallelogram which includes the data to take part in this partition’s computation. In the figure, the dash line parallelogram \(A'B'C'D'\) denotes the corresponding region determined by \(d_2 = (2, 2)\), while the dotted line parallelogram \(ABCD\) is determined by \(d_1 = (3, 1)\). Because of the uniform of data dependence, the parallelograms determined by \(d_1\) and \(d_2\) have the same shapes and sizes as the current partition, with their lower left corner at \((i - 2, j - 2)\) and \((i - 3, j - 1)\), respectively. As the same rule, the parallelogram \((BEFC \text{ and } B'E'F'C')\), lie on the right of and adjacent to the corresponding regions \((ABCD \text{ and } A'B'C'D')\), include all the data of this array needed by the next partition’s computation. Therefore, the data set is the union of all four regions \(ABCD, BEFC, A'B'C'D',\) and \(B'E'F'C'\).

If this data array appears in the recurrence equations (modified during the computation and can appear as an left operand), its data set is shown in Figure 6.8(b). The parallelogram \(ABCD\) and \(A'B'C''D'\) have the same meaning as above. The difference is due to the recurrence of the data array. According to the data dependence, all the data, lying in the current partition parallelogram except of the prefetch region, are computed in the current partition and reused in the next
partition. These data should also be maintained in the cache for either current or next partition’s computation. The data to be prefetched for the next partition are only those data which cannot be covered by the current or next partition, which are $EGFC$ and $EGF'C'$ determined by $d_1$ and $d_2$, respectively. Thus, the data set for such array is the union of $ABCD, A'B'C'D', EGF'C, EGF'C'$, and $OPEQ$.

Let the height of a data set be the number of rows in this data set, then this height can be easily calculated through the above method of determining the data set. For instance, the heights are 7 in both figures. In a data set with the height $H$, all the rows are numbered from the bottom to top as 1 to $H$.

As about the relation between the data set of current and next partitions, it is obvious that they have the same shape and size. Their cache locations satisfy the following property.

**Property 5** For each array’s data set, when the execution proceeds from the current partition to the next partition, each cache block the will shift right a cache distance $\lambda$, where $\lambda = |P_x|$.

We can consider only one partition and apply data padding to prevent conflict during this partition execution time. The above property guarantees that there is no cache conflict in the other partitions execution time.

For a certain partition size, the data set for each data array can be decided. All these data sets should be kept in the cache during the execution of a partition. However, with direct mapped cache, conflict miss may occur due to the fact that different elements may map to the same cache location. Our objective is to eliminate all the cache conflict misses. Inter- and intra- variable data padding can be

\[\text{For any array, each row of the data set occupies a block of continuous cache locations. This data set corresponds to several blocks of data in the cache.}\]
combined to accomplish this goal. Intra-variable padding is applied first to distribute the data set of an array in the cache, such that the space between any two blocks of data is larger than a certain value. We call this certain value as the safe distance and the requirement that the space must be larger than the safe distance as safe distance constraint. The safe distance is large enough to accommodate all array data from the rows with the same row number. The data layout in the cache after data padding is shown in Figure 6.9. Suppose that a data array has the data set $DS$, whose height ($HL$) is the largest. Then the cache is partitioned into $HL$ parts. Each row from $DS$ (light shaded rectangle in the figure) lies at the head of each part. For other data sets, $i^{th}$ rows lie in the $i^{th}$ part. As long as the safe distance constraint is satisfied, the $i^{th}$ part is large enough to contain all such data. For example, if we have three data arrays $E$, $F$ and $G$, they all have $8 \times 4$ rectangle data sets. Assume the cache size is 512, cache line size is 4, then the safe distance is 36 according to Algorithm 9. If the array row size is 256, their cache data layout of one partition is shown in Figure 6.10. In the figure, each rectangle is labeled by an array name, followed by a row number. It represents the cache positions occupied by this row. The number beneath the line denotes the corresponding cache position.

It is not necessary to calculate the pad size for each array to satisfy the safe distance constraint. The workload can be reduced due to the uniform data dependence.

**Theorem 27** When intra-padding is applied, if a pad size can satisfy the safe distance constraint for the array with the largest height, this pad size can be applied on other arrays without violating the safe distance constraint.

**Proof:** Because of the uniform data dependence, the arrays can be regarded as
owning the same dimensions and sizes. Since the data dependence is quite small relative to the array size, the cache distance between two adjacent rows of data set is mainly decided by the array size. The distance between two adjacent rows of any data set can be thought of as the same.

On the other hand, the larger the height is, the more number of cache blocks in the cache, which means that more constraints are needed to meet to maintain the safe distance. Combining this statement with the above statement, we can easily verify the correctness of the theorem.

We first give a definition for the convenience of describing the algorithm.

**Definition 16** For a row of $\beta$ elements, the function $MaxC(\beta)$ is defined to be the maximal possible number of cache lines to include this row of data.

For example, if each cache line consists of 4 elements, $MaxC(3) = 2$ and $MaxC(8) = 3$. 

Figure 6.9. Data layout in the cache

Figure 6.10. An example of data layout in cache
In Algorithm 9, the safe distance is calculated in the first three steps. It is the maximal possible cache lines occupied by the data from all rows with the same number. Note that the cache distance in this chapter is a circular cache distance. There are two cache distance value between any two blocks. We adopt the one with the smaller absolute value as the cache distance. For instance, if the cache size is 512, the cache distance between 1 and 510 is 3, instead of 509. In the data set, the start position of a row is the column number of its leading element. Each row may have a different start position. Suppose all the start positions are same, i.e., all rows start at the same column, the following observation is easily deduced.

Property 6 Assume the cache distance between $n^{th}$ and $m^{th}$ rows of data set is $\alpha$ and the height of the data set is $H$. Then for any two rows $l^{th}$ and $k^{th}$, where $l, k \in [1, H]$, their distance equal to $\alpha$ as long as $|l - k| = |n - m|$.

Thus, we can simply ensure that the first row of data set has the enough cache distance with all other rows to maintain the safe distance constraints. If the start positions are different, the safe distance constraint can still be satisfied after the difference of start positions are considered. In the algorithm, without loss of generality, the first row of data set is put at the beginning of the cache. The cache size is $CS$. The time complexity of the algorithm is $O(H^2)$.

Inter-variable padding can be used to eliminate the cross-interference between different arrays. The pad size should be selected such that no two arrays conflict in the cache. We use the concept adjacent data arrays to derive the pad size. Two data arrays are said to be adjacent if their corresponding data set satisfy: two rows, with the same row number, lie side by side, as shown in Figure 6.9. $DR$ means the space between the two cache blocks, which come from the different data sets but have the same row number. When $DR \leq 2 \times C$, they are said to lie side
Algorithm 9 Calculate the intra-pad size \( p \)

**Input:** The partition size and shape, data dependences

**Output:** The intra-pad size

1. For each data array, determine the data set according to the partition and data dependences. Find the array with the largest height data set. Calculate its data set size and start position of each row of the data set. /* \( SP_k \) is used to denote the start position of \( i^{th} \) row and \( DSize \) is used to represent the size of a data set*/
2. In all the data sets, find all the rows which have the same row number. Suppose the number of such rows is \( n \), and each one has size \( s_i \), calculate the summation \( S = \sum_{i=1}^{n} MaxC(s_i) \).
3. Scan through all possible rows, choose the maximum \( S \) as the safe distance \( DIS_{safe} \).

   - If \( n \times DIS_{safe} + DSize > CS \) then
     - quit /* no padding pattern can be found*/

/* In the following, \( A, B \) are used to record the minimum and maximum difference of start positions. */

**FOREACH** \( m = 1, n - 1 \) do
  - \( A[m] = Min(SP_k - SP_{k-m}) \) \( \forall k \in [2, n] \);
  - \( B[m] = Max(SP_k - SP_{k-m}) \) \( \forall k \in [2, n] \);
  - Calculate \( d[m] \) as the distance between cache locations of the first and \( m^{th} \) rows of data .

**ENDFOR**

**FOREACH** \( i \in [1, n] \) do
  - if \( d[i] + A[i] < DIS_{safe} \) or \( d[i] + B[i] < DIS_{safe} \) then
    - \( p = MAX((DIS_{safe} - A[i] - d[i])/i, (DIS_{safe} - B[i] - d[i])/i) \).
  - end if

/* The following loop eliminates the possible new violation of safe distance constraint introduced by the above pad*/

**FOREACH** \( m \in [1, i] \) do
  - if \( d[m] + A[m] + m \times p < DIS_{safe} \), then \( p = (DIS_{safe} - A[m] - d[m])/m \).
  - if \( d[m] + B[m] + m \times p < DIS_{safe} \), then \( p = (DIS_{safe} - B[m] - d[m])/m \).

**ENDFOR**

**ENDFOR**
by side. Due to the safe distance constraint, the space between two blocks from the same arrays is large enough to accommodate all the blocks from other arrays. Therefore, The inter-pad size can be determined as long as these two arrays are adjacent.

6.5 Experimental results

In this section, the effectiveness of multiple loop partition scheduling technique is evaluated by running a set of simulations on four benchmarks. Example is the example code shown in Figure 6.1(b). Jacobi is a PDE solver. LL18 is the eighteenth kernel from the Livermore Loops benchmark. Tomcatv is extracted from the 101.tomcatv in SPEC95 benchmark. We compared the $AET/Iter$ by applying three different schemes on these benchmarks. $AET/Iter$ (Average execution time per iteration) is the summation of all the average execution times for one iteration in each loop nest. Because of the comfortability of the loop nests, AET/Iter reflects the overall effectiveness of each scheme.

In the simulation environment, we have 4 ALU and 4 memory units. The first level memory is 32KB in size and full associative. Each data is float type which consumes 8 bytes. The computation time for each operation is simplified as one unit time. The data are prefetched in block with size 64 bytes. This operation takes 10 units. The assumption is reasonable considering the big gap between the CPU and memory speeds.

We first compare multiple loop partition scheduling and the traditional partitioning technique, i.e., partition each loop nest separately. The former is superior to the latter, due to the fact that the better data locality is exploited and the repetitive array element accesses are reduced. Figures 6.5 shows how average
cache miss/Iteration changes with the cache block size (denoted by the number of float data in a block) for LL18 kernel under the above simulation environment. We can see that the number of average cache miss for the traditional partitioning technique is always two times the number of our technique. This observation can be explained by the number of array references per iteration. Partitioning separately has two times memory reference as many as multiple loop partition.

The simulation results are listed in Table 1. Three methods compared are Original, which is the original loop without any transformation, Fusion, in which we apply the loop fusion on the benchmarks, and MLP, which is multiple loop partition. Because it has been shown in [85] that software pipelining and loop tiling cannot outperform the traditional partitioning technique, we didn’t compare their results. The column \#L denotes the number of loop nests in each benchmark. The size column is the partition size. We also list the theoretic lower bound in the \(L_{\text{bound}}\) column. As mentioned in Section 3, the ALU schedule
is obtained by Multi-dimensional Rotation Scheduling Algorithm \[58\] without the consideration of the memory load latency. The ALU schedule length per iteration provides the lower bound for computation under the resource constraints. Comparing the AET/Iter, we can find that partitioning technique can achieve the best performance. Accredit to increasing the data locality appeared between different loop nests, the loop fusion perform better than the original loop. The multiple loop partitioning not only takes advantage of the benefit of loop fusion, but also explores more data reuses which exist in the different iterations. Moreover, it takes the balance of computation and memory reference into consideration, thereby prevents the wasted time caused by the dominant memory reference time. Therefore, the multiple loop partition scheduling technique can efficiently exploit the data locality.

<table>
<thead>
<tr>
<th></th>
<th>#L</th>
<th>Original</th>
<th>Fusion</th>
<th>MLP</th>
<th>size</th>
<th>(L_{bound})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>4</td>
<td>8.5</td>
<td>5.41</td>
<td>\textbf{2.31}</td>
<td>8 \times 8</td>
<td>2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2</td>
<td>5.25</td>
<td>4.0</td>
<td>\textbf{2.14}</td>
<td>4 \times 8</td>
<td>2</td>
</tr>
<tr>
<td>LL18</td>
<td>3</td>
<td>20.85</td>
<td>13.44</td>
<td>\textbf{11.5}</td>
<td>4 \times 4</td>
<td>11</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>5</td>
<td>25.25</td>
<td>20.38</td>
<td>\textbf{18.21}</td>
<td>4 \times 4</td>
<td>18</td>
</tr>
</tbody>
</table>
The performance of multiple loop partition scheduling is very close to the lowerbound, which demonstrates that the memory latency is hidden very efficiently. Their difference is due to the iterations lie on the boundary of iteration space. These iterations can be regarded as the preface, which is the preparation stage of the normal partition execution.

![Figure 6.12. Experiment result of data padding](image.png)

If the first level memory is direct mapped cache, the average execution time will become larger due to the conflict misses. However, the same result as Table 6.1 can be achieved after data padding is applied, which is illustrated in Figure 6.12. In this figure, No padding denotes the result without data padding, while Data Padding shows the result after data padding. Lower bound has the same meaning as above.
In tomcatv benchmark, there are five loop nests with the same iteration space size. However, the first three and the last two loop nests should be thought of as in two independent groups, since the step direction is contrary in different groups. The first group is in the form as \( do 10 i = 1, N, 1 \), while the second group is in the form of \( do 10 i = N, 1, -1 \). Because loop fusion can be regarded as a special case of multiple loop partition when the partition size is 1, The problem to group loop nests is essentially the same as the fusible loop problem [51].

6.6 Conclusion

In this chapter, a new scheme which combines loop partitioning and data padding was proposed. The new algorithms consider the case with multiple consecutive loop nests and partition them at the same time, such that the data locality can be exploited to the largest extent. Moreover, a good balance detailed partition schedule is found so as to efficiently hide the memory latency. In order to deal with low-associativity cache, data padding is utilized to extend the application of our technique. Experiments show a significant improvement over the existed approaches.
In this chapter, we discuss a rotation scheduling algorithm to optimize the loop performance in the clustered architecture, where both register files and data caches are distributed among clusters. It has been shown that rotation scheduling is one of the best software pipelining techniques in the centralized architecture [13, 59]. Nevertheless, the traditional rotation algorithm does not consider the register constraint or cluster configuration, and thus limits the usage of this algorithm. We apply the idea of rotation scheduling to the clustered architecture by extending it to handle the register constraint. The proposed algorithm can efficiently explore the parallelism at the level of distributed function units and registers. It can handle arbitrary clustered datapath configuration and, along with schedule latency minimization, can effectively handle the register constraints. Previous methods [3, 72, 73] treat communication BUS as a type of hardware resource and inter-cluster communication operation as same as other operations during scheduling. On the contrary, we collect all the MOVE operations (used to move operands between clusters) and schedule them at the later stage of the scheduler. With all the global information gathered, the MOVE operations can be scheduled in a more efficient way such that the register resource can be better utilized. The algorithm
to schedule MOVE operations is presented and the register usage improvement can be seen from the experimental results.

Another important aspect in clustered architecture is the distributed cache. Distributed caches allow multiple clusters to perform memory operations simultaneously while a single cache only provides very limited parallel memory operations. The consideration of distributed cache comprises of two aspects: which cache the spilled variables should be put in and how to arrange the data in the distributed caches. The first problem comes up when the register resource is restrictive. By judiciously inserting spill code and taking advantage of the local cache of each cluster, our technique can satisfy the register constraint without significantly increasing the schedule latency. The second problem is solved in our technique by the incorporation of data padding [69]. To guarantee that the exploited data locality is not canceled by the cache conflict misses, which may severely compromise the schedule for VLIW architecture due to the lockstep execution, we form an integer linear programming (ILP) problem. By solving this ILP, the least pad size to eliminate the cache conflict misses can be obtained.

Previous research seldom deals with the clustered VLIW architecture that has distributed data caches. To our best knowledge, [73] is the only one having such consideration. The comparison illustrated in the experimental results section shows that our algorithm is superior to [73] on both schedule latency and register usage.

7.1 Architecture model

In this paper, we use the architecture model similar to that of $Lx$ [28], as shown in Figure 7.1. It is a multi-cluster architecture. Each cluster is composed of a mix
of register files and function units. A single program counter and a unified I-cache control all the clusters, so that they run in lockstep. Inter-cluster communication, achieved by explicit register-to-register move, is compiler-controlled and invisible to the programmer. To increase the parallelism and efficiency, each cluster has its own local data cache, thereby multiple data caches exist in the architecture.

To establish main memory coherency in the presence of multiple memory access, a MESI-like synchronization mechanism can be used for multiple independent caches. This protocol is completely transparent to the ISA. Both the coherence and the bus arbitration are managed by the hardware.

Regarding memory accesses, a load/store issued by a cluster first tries its local data cache. If the data is found, the access is satisfied with minimum latency. Otherwise, the access is solved by the MESI-like synchronization protocol.

![Figure 7.1. Lx architecture model](image_url)
7.2 Scheduling algorithm

The problem we are attacking in this chapter is defined as follows:

**Problem:** Given a DFG and a clustered datapath configuration, find a schedule that minimizes the schedule length and satisfies the register constraint. Insert spill codes if necessary to reduce the register requirement. Find the suitable data padding pattern to eliminate cache conflict misses.

From this definition, we can see that the solution for such problem can handle all major aspects of the clustered architecture, i.e., cluster configuration, register constraint, inter-cluster communication and distributed caches.

7.2.1 Scheduler framework

The framework of the scheduler is shown in Figure 7.2. The input are the data flow graph and system specifications, i.e., clustered datapath configuration, cache configuration and register constraint. The output is a minimized feasible schedule for such a system.

As the first step, a k-way partition algorithm is used to allocate the DFG nodes to different clusters. The objective of this algorithm is to minimize the communication cost and balance the nodes’ cost in each cluster. Because a generic k-way partition algorithm does not consider the influence of the critical path on the schedule length, it is difficult to associate a good partition with a compact schedule length when scheduling acyclic code. However, in cyclic code scheduling, most critical paths can be broken up by a suitable retiming. Hence a k-way partition algorithm is used as the first step to form a good initial partition. The following points should be kept in mind when implementing the k-way partition module.
Figure 7.2. The scheduling algorithm framework

**Point 1:** Any edge between two nodes corresponds to an communication cost of 1 regardless of its weight, since the weight is only related to timing information.

**Point 2:** The balanced cluster load means that the load is evenly distributed to each cluster for every node type, while the minimized communication cost is found from the global view.

**Point 3:** Strongly connected component (SCC) with small overall latency should be treated specially such that they are put into the same cluster with a high probability.

We use an example to help illustrate the last two points. Assume an architecture with two clusters, each of which has an ALU and a multiplication unit. For a DFG which has 2 multiplication (cost 3) nodes and 6 addition (cost 1) nodes, a good partition will allocate 3 additions and 1 multiplication to each cluster.
A partition, with 2 multiplications in one cluster while 6 additions in the other cluster, is not acceptable though the cluster cost is balanced. As of the combination of 1 multiplication and 3 additions, it should be decided by taking into consideration all the communication cost between any two nodes to minimize the communication cost. For point 3, a SCC of a DFG is a subgraph such that for every pair of nodes \( n_i \) and \( n_j \) belonging to this SCC, there exists a path \( n_i \rightarrow n_j \) and \( n_j \rightarrow n_i \). A SCC with small overall latency always exerts an extra restriction for the schedule. Retiming cannot change the overall latency of such a cycle. If a critical path exists in this kind of SCC, it cannot be broken up by retiming. Therefore, it is beneficial to put such SCC in one cluster.

K-way partition is an NP-complete problem and many heuristics algorithms exist. We modified the algorithm in [40] due to its easy implementation and fast speed. Point 2 is handled by artificially assigning special costs to different kinds of operations. We take care of Point 3 by a post-processing step after the DFG is partitioned. Although the k-way partition algorithm we used is fairly simple, it is good enough to generate a partition from which an efficient schedule can be derived. However, with a better k-way partition heuristic algorithm, some improvements of the final schedule may be expected.

After the DFG is partitioned, an initial schedule can be generated by the force-directed list scheduling module [60]. This initial schedule can be optimized by repeatedly applying rotation and partial rescheduling. Step 1 and 2 are repeated several times until a minimal schedule length is reached.

In the clustered architecture, rotation is performed on the basis of each cluster. That is to say, an operation is only rescheduled to another available control step

---

*Any other scheduling technique, e.g., other kinds of list scheduling, can be used in this framework. We choose force-directed list scheduling because it can handle the resource constraint and operation concurrency effectively.
in the same cluster. Only allowing operation rotation in the same cluster avoids the change of the inter-cluster move operations. With such a rotation scheme, the assignment of variables to clusters does not vary with rotation, thereby the communication workload on the communication bus remains constant with rotation. However, the variable lifetime may be changed by rotation. In the rest of this section, we discuss in detail the last three procedures in our framework.

7.2.2 Move operation scheduling

When an operation requests a variable in another cluster, an inter-cluster MOVE operation is needed. The schedule of MOVE operations influences the maximum register requirement significantly. Through a better MOVE operation schedule, register loads may be distributed to clusters more evenly such that the overall register requirement is reduced. By considering the MOVE operation scheduling after rotation, we have a comprehensive view of the global register usage and can use it to achieve more efficient register usage. The greedy MOVE scheduling algorithm is shown in Algorithm 1.

The register requirement of a schedule can be derived by a register usage map. In our algorithm, we divide the register usage map into two parts, cluster part for all except MOVE operations, and move part for only MOVE operations. In the move part, the lifetime of a MOVE operation starts from the last control step when the variable to be moved is alive in the producer cluster and ends at the first control step when the variable is needed in the consumer cluster.† The MOVE operation can be scheduled at any control step belonging to its lifetime. Scheduling

†In the case of two MOVE operations deal with the same moving variable but have different control steps, a and b in the consumer cluster, the MOVE operation with the later control step can be omitted, and the span of lifetime b−a (assuming b ≥ a) is counted into the cluster part register usage map.
a MOVE at different control steps will have different impact on the register usage map of the move part. For example, scheduling a MOVE operation whose lifetime is $1 \rightarrow 5$ at control step 2 will add 1 to the move part register usage map of the producer and consumer clusters at control steps 1, 2 and 2, 3, 4, 5, respectively, while scheduling it at control step 4 will add 1 at control steps 1, 2, 3, 4 and 4, 5, respectively. If the maximum register requirement of the cluster part occurs at the control step 3 in the consumer cluster, we prefer to scheduling this MOVE at control step 4 since it will not increase the overall register requirement.

\begin{algorithm}
\caption{MOVE operation scheduling algorithm}
\label{alg:move-scheduling}
\textbf{Input:} MOVE operations and the schedule after rotation.
\textbf{Output:} MOVE operation schedule which can generate the efficient register usage
\begin{enumerate}
\item Derive the register usage map for cluster part.
\item Sort all the MOVE operations in the increasing order of its lifetime and put them into list $L$.
\item \textbf{WHILE} ($L$ is not empty) \textbf{do}
\begin{enumerate}
\item Pop out the head $h$ of $L$
\item \textbf{FOREACH} (control step $i$ in $h$’s lifetime) \textbf{do}
\begin{enumerate}
\item Calculate $F(i)$
\end{enumerate}
\textbf{ENDFOREACH}
\end{enumerate}
\item Sort all $F(i)$ in the increasing order and put it into list $Force\_list(h)$
\item Tentatively schedule $h$ in the control step $Con$ with the least force.
\textbf{ENDWHILE}
\item In all the control steps with Bus resource conflict, find the MOVE operations $h$ with the smallest current force. Reschedule it to a available control step with the next smallest force in the $force\_list(h)$. If no control step is available, increase the schedule length by one, and reschedule this MOVE.
\item Update the $Force\_list(h)$ for all other MOVE operations and repeat the third step until there is no BUS resource conflict.
\end{enumerate}
\end{algorithm}
Suppose the largest register requirement in the cluster part is \( \text{MAX} \). If a MOVE operation is scheduled at a certain step \( \text{Con} \), a new move part register usage map can be derived. In each control step \( i \), we find the maximum value \( M_i \) for all clusters at this control step from the overall register usage map. The force of scheduling MOVE at \( \text{Con} \) at this control step is defined as \( F(\text{Con})_i = M_i - \text{MAX} \), and the force of scheduling MOVE at \( \text{Con} \) is defined as \( F(\text{Con}) \), which is the largest of \( F(\text{Con})_i \). We always try to schedule the MOVE operation to a control step \( b \) which has the least force \( F(b) \), where \( b \) belong to the MOVE operation’s lifetime.

The greediness of this Algorithm lies in that it always tries to find the local optimal control step to schedule a MOVE operation. It is easy to verify that the time complexity of this algorithm is \( O(m^2n^2) \), where \( m \) is the schedule length and \( n \) is the number of MOVE operations.

7.2.3 Spill code insertion

If the register requirement still exceeds the constraint after the first four steps in Figure 5, spill codes need to be inserted to satisfy the register constraint. Spill code insertion has been considered in many papers [20, 98]. None of them has dealt with distributed cache. In the clustered architecture with distributed cache, the consideration of spill code insertion includes two aspects: which variable is spilled and which cache should the spilled variable be put in.

Spill code is added through a pair of memory operations: writing back to the local cache and loading it later. Each cluster has its own local caches, which can be accessed much faster than main memory and remote caches. To improve the efficiency, a spilled variable is always written back to the local cache of its
consumer, i.e., the variable has been moved to the desired cluster before spilled out to the cache.

If a cluster demands more registers than available, several steps are carried out to get rid of the violation.

1. Find a variable with the longest lifetime and its spilling can help reduce the register pressure.

2. If the variable’s lifetime is longer than the overall cost of writing back and loading from the local cache, the corresponding spill code is inserted.

3. Otherwise, the schedule length is increased by one. The operations in the critical control step, whose register requirement exceeds the constraint, are redistributed into two control steps and the MOVE operations is rescheduled to reflect the schedule change.

4. The above steps are repeated until all the register constraints are satisfied.

These steps are effective in reducing the register pressure since the general register constraint violation is often caused by long variable lifetimes that span several iterations. It is advantageous to swap them out of the register file to decrease the register usage. If no such long lifetime variable exists and the register limit is still surpassed, the register resource is really tight. The only way to satisfy the constraint is to increase the schedule length.

7.2.4 Data padding

Cache conflicts may occur when loading data from main memory to a cache or spilling variables from a register file to a cache. In both cases, the execution has to be stalled (in the later case, the execution is stalled when the spilled variable is
Data padding is an effective technique to eliminate cache interference. It can be classified as inter- and intra-variable padding. Inter-variable padding is applied to several variables. Some gaps are inserted between different variables to eliminate the cross interference among these variables. On the other hand, intra-variable padding is applied to a single variable so as to eliminate the self-interference.

Figure 7.3. Data padding illustration

Figure 7.3 is an example of inter- and intra-variable padding. In Figure 7.3(a), unit-stride references to $A(i)$ and $B(i)$ provide spatial locality, leading to cache reuse. However, if $A$ and $B$ are separated by a multiple of the cache size and the cache is direct mapped, every access to $B(i)$ will map to the same cache line as the previous $A(i)$ access and vice versa. As a result, every reference will produce a conflict miss. To regain the benefits of spatial reuse, inter-variable padding may
be applied as shown in the figure. In Figure 7.3(b), the computation exhibits much spatial and temporal reuse between the references to $A$. However, if the column size of $A$ is a multiple of the cache size, columns of $A$ will conflict, eliminating reuse between references to $A$. To regain spatial reuse, intra-variable padding applied to the columns of $A$ as shown can change its internal layout so that columns no longer map to conflicting cache locations.

In [50], Naraig et al use inter-variable padding to partition the cache into equal parts in order to prevent cache conflict misses. This simple method is not applicable in our technique because several blocks of the same array maybe required to exist in the cache at the same time. Only a change of the starting address of each array cannot guarantee no conflicts between these blocks. Therefore, we combine inter- and intra-variable data padding to eliminate the cache conflict misses. In the following, we consider the case with a direct-mapped cache.

The objective of data padding is to let the data blocks for all the array accesses of one iteration coexist in the cache without any overlap among them. The data blocks, which can coexist during one iteration, are composed of the data blocks loaded from the main memory for calculation and the data blocks spilled out of the register file. Due to the uniform data dependence, we can extract data padding patterns for any iteration to prevent conflict misses, such that the same pattern can achieve the same effect for any other iteration. Furthermore, the overall pad size should be as small as possible to spare the memory space. We will formulate an integer linear programming problem (ILP) to help achieve our goal.

As the first step, we gather the information of the starting address of all data blocks. This can be accomplished by calculating the relative distance of each data block from the current iteration index. For instance, assuming the current
iteration index is \((i, j)\), the starting address of the data block for access \((i+3, j-4)\), and \((i - 1, j)\) are \(3 \times row_{size} - 4\) and \(-1 \times row_{size}\), respectively, where \(row_{size}\) is the array row size. Actually, these are not accurate if we consider that the data block’s starting address is a multiple of the cache block size. However, as long as we make the cache distance between two addresses larger than two cache block size, the corresponding two cache blocks will have no overlap no matter where these two addresses are located in their data blocks.

\[
\begin{align*}
\text{Minimize} & \quad \sum_{i=1}^{n-1} pinter_i + \sum_{i=1}^{n} n_i \times pintra_i \\
\text{*** Intra variable constraint ***} & \\
& \vdots \\
& \quad cache_{size} - 2 \times line_{size} \geq (db_m^i - db_n^i) \% cache_{size} \geq 2 \times line_{size} \\
& \vdots \\
\text{*** Inter variable constraint ***} & \\
& \vdots \\
& \quad cache_{size} - 2 \times line_{size} \geq (db_m^i - db_n^i + Addr_i - Addr_j \\
& + \sum_{l=j+1}^{i-1} pinter_l + \sum_{l=j}^{i-1} n_l \times pintra_l) \% cache_{size} \geq 2 \times line_{size} \\
& \vdots \\
& \quad pinter_1, \ldots, pinter_n, pintra_1, \ldots, pintra_{n-1} \geq 0
\end{align*}
\]

Figure 7.4. ILP problem for data padding
Suppose the starting address for each array are $Addr_1, Addr_2, \ldots, Addr_n$, the inter-pad size for a sequence of arrays are $pinter_1, pinter_2, \ldots, pinter - n - 1$ and the intra-pad size for arrays are $pintra_1, pintra_2, \ldots, pintra_n$, the ILP problem can be summarized as in Figure 7.4.

In the constraints, $db^i_m$ means the relative starting address of $m$th data block in $i$th array. In its calculation, the $row_{size}$ is the array row size after inter-variable padding, i.e., $pintra_i$ plus the original row size. $cache_{size}$ is the size of cache, while $line_{size}$ is the cache block size. The array $l$ has $n_l$ number of rows. The constraints together ensure that the distance between any two data blocks are larger than two cache block sizes. Intra-variable constraints are applied on two data blocks of the same array, while inter-variable constraints are applied on two data blocks of two different arrays.

The objective function aims at minimize the total pad size. In the clustered architecture, the data are distributed into different data caches, The number of constraints is reduced compared to the centralized cache. For instance, for a program with 16 arrays and only one data block from each array, the number of constraints is 240 for the centralized cache and 48 for the 4-distributed cache (assume 16 data blocks are evenly distributed into distributed caches), respectively. In general, this ILP problem size is small enough to be solved quickly. In the case of the problem size is really large, the less memory space efficient greedy algorithm can be used.

7.3 Previous work

Previous research work related to cluster scheduling can be classified into two categories: scheduling with computational DAGs and scheduling with cyclic code.
In [23], Desoli developed a two-phase binding algorithm called Partial Component Clustering. Ozer et al [57] presented a greedy binding/scheduling algorithm, which binds and schedules the ordered operations to a priority list of clusters in one step. The priority of the cluster is determined by several heuristics. Lapinskii and Jacome [39] proposed an algorithm whose initial schedule explores tradeoffs between in-cluster operation serialization and delays associated with inter-cluster data movement. An iterative scheme can be applied on this initial schedule to compact the schedule further. These acyclic code scheduling algorithm cannot be directly applied to the cyclic code, since they can not take advantage of the inter-iteration data dependences.

Software pipelining [36] is efficient in scheduling cyclic code through moving operations among the iterations such that a shorter Initiation Interval (schedule length) can be achieved. Many work on software pipelining have been done in the centralized architecture, such as Modulo scheduling [44], rotation scheduling [13] and retiming scheduling [2]. Here we briefly review algorithms that target clustered architecture. Sanchez et al [72] proposed the modulo scheduling algorithm which performs the cluster assignment and instruction scheduling in a single pass and considers the possible improvements by loop unrolling. Akturan and Jacome [3] developed the CALIBeR framework for clustered embedded VLIW processors. The framework tries to optimize the schedule length as well as minimize the register usage and code size. It is the state-of-art clustered scheduling algorithm without considering distributed cache. To our best knowledge, the technique in [73] is the only clustered scheduling approach which consider both the distributed register file and caches. It use Cache Miss Equations [31] to predict the influence of allocating memory operations and modulo scheduling to derive the compact
schedule. Since solving Cache Miss Equation is NP-complete, they adopted the heuristic solver in their approach.

7.4 Experimental results

The effectiveness of our technique is illustrated by running a set of benchmarks from [3]. We compared our results with CALiBeR in [3] and Modulo scheduling in [72]. The experimental results for the latter two methods are extracted from [3]. Since the benchmarks are the same, the comparisons are fair. Table 1 lists the benchmarks and the clustered datapath configuration. In this table, the corresponding benchmarks from left to right are Lattice Filter, 2 Cascaded Biquad Filter, Avenhous Filter, 4 Cascaded FIR Filter, AR Filter, 4 Cascaded Biquad Filter, DCT-DIT. The number of nodes in the DFG is listed after the benchmark’s name. The clustered datapath is specified in the form of number of ALU (a), multiplication (m) and load/store (x) units.

In order to evaluate the efficiency of the scheduler itself, we show the experimental results in Figure 7.5. The schedule length for 4 different schedulers are listed. They are cluster and register aware rotation scheduling (rotation), CAL-iBeR from [3] (CALiBeR), modulo scheduling in [72], and the RS-FDRA in [2] (Centra). RS-FDRA is a centralized architecture software pipelining scheduling algorithm. It is used to provide a criteria to show how well the other three schedulers can perform in clustered architecture. It has been shown in [2] that this algorithm and rotation scheduling can always achieve the best results in the centralized architecture. In a clustered architecture, after the function units are partitioned into clusters, the schedule length cannot get better due to the extra cost introduced by
<table>
<thead>
<tr>
<th></th>
<th>LF (16)</th>
<th>2CBF (16)</th>
<th>AF (20)</th>
<th>4CFF (32)</th>
<th>AR (34)</th>
<th>4CBF (32)</th>
<th>DCT (48)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>C2</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>Cluster1</td>
<td>1a</td>
<td>2a</td>
<td>1a</td>
<td>2a</td>
<td>1a</td>
<td>2a</td>
<td>2a</td>
</tr>
<tr>
<td></td>
<td>1m</td>
<td>2m</td>
<td>1m</td>
<td>2m</td>
<td>1m</td>
<td>2m</td>
<td>2m</td>
</tr>
<tr>
<td>Cluster2</td>
<td>1a</td>
<td>2a</td>
<td>1a</td>
<td>2a</td>
<td>1a</td>
<td>2a</td>
<td>2a</td>
</tr>
<tr>
<td></td>
<td>1m</td>
<td>2m</td>
<td>1m</td>
<td>2m</td>
<td>1m</td>
<td>2m</td>
<td>2m</td>
</tr>
<tr>
<td>Cluster3</td>
<td></td>
<td>1a</td>
<td>2a</td>
<td>3a</td>
<td>3a</td>
<td>3a</td>
<td>3a</td>
</tr>
<tr>
<td></td>
<td>1m</td>
<td>1m</td>
<td>1m</td>
<td>1m</td>
<td>1m</td>
<td>1m</td>
<td>1m</td>
</tr>
<tr>
<td>BUS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
inter-cluster communication. Therefore, the results from RS-FDRA can be regarded as the lower bound for other algorithms.

CALiBeR is selected for comparison because it is the start-of-art software-pipelining algorithm in clustered architecture. The modulo scheduling in [72] is selected for comparison because it is the baseline scheduling algorithm of [73].

If the register resource is not restricted, we can see from Figure 7.5 that all three clustered scheduling algorithms are quite effective in that they can reach the lower bound in most cases. However, from the pointer view of register cost, as shown in Figure 7.5, our technique requires the least number of registers than the other two schedulers. The register usage comparison demonstrates that the global MOVE operation schedule can indeed reduce register cost.

When the register resource is restrictive, spill codes have to be inserted to swap some variables out of the register file. CALiBeR does not consider the register constraint, thereby cannot obtain a schedule under the rather restrictive register constraint. Through judiciously inserting spill code, our technique can reduce the register requirement by 1/3 - 1/2 with almost the same schedule length, as shown in Figure 7.6. Moreover, our technique can still get a reasonable result if the register constraint becomes tighter. In such a case, the schedule length is expected to become longer to modify the register requirement distribution and accommodate the swapping time between the register file and local cache.

To show the importance of data padding, we use Figure 7.7 as an example. In this figure, we show the number of cache conflicts occurred during one iteration of a nested loop, swim(200), which is extracted from the swim benchmark of spec98. In the benchmark specification, all the array sizes are 513 × 513. Figure 7.7 shows that the cache conflict misses vary significantly with slight variations of array
Figure 7.5. The schedule length and register requirement without register constraint
Figure 7.6. The schedule length and register requirement if register resource is restricted. NRC and RC represent the performance without register constraint and with register constraint, respectively.
size and cache configuration. In the clustered architecture, cache conflict misses will severely degrade the system performance due to the lockstep execution of VLIW instructions. Benefits can be gained from data padding by eliminating these conflict. For instance, with a 2K cache and 8 cache block size, without data padding, there exist 10 cache conflicts out of 25 memory accesses in each iteration. These cache conflicts can be eliminated either by intra-padding (change the array size to 515) or inter-padding (inserted 3 pad between each arrays). The performance can be improved by eliminating cache conflicts.

![Cache Conflict Misses](image)

**Figure 7.7.** The number of cache conflicts under different array size and cache size

As we mentioned before, the algorithm in [73] is the only existing algorithm
which can handle the distributed cache. It takes the distributed cache into consideration by merging the scheduling of memory operations into their baseline algorithm, and solving the Cache Miss Equation to determine the cluster and control step to which a memory operation belongs. The method to deal with spill codes is not discussed in their paper. Although Cache Miss Equation solver can improve the cache performance to some extent, it cannot eliminate cache conflict misses, whose existence will severely compromise the schedule. As shown in Figure 7.5, our scheduler outperforms the baseline algorithm in [72] in both schedule length and register requirement. Moreover in our algorithm, cache conflict misses are eliminated through data padding. Therefore, We believe that our scheduler is better than the algorithm in [73].

7.5 Conclusion

We have presented a partition/scheduling framework suitable for compilers targeting clustered VLIW processors. A novel global inter-cluster communication scheduling algorithm is proposed to efficiently utilize the register resource. To satisfy the restrictive register constraint, spill codes are wisely inserted. Furthermore, Data padding is incorporated into our technique to fully take advantage of the distributed cache. The proposed algorithm can handle arbitrary clustered configuration and, along with latency minimization, can effectively handle register constraint. The experimental results demonstrate that our technique is superior to the existing algorithms.
CHAPTER 8

ENERGY AWARE VARIABLE PARTITIONING AND INSTRUCTION SCHEDULING FOR MULTI-BANK MEMORY ARCHITECTURES

We present our approach to variable partitioning and memory access operation scheduling in the presence of multi-bank memory and multiple memory operating modes for maximizing energy saving without sacrificing performance. We reveal some observations to help categorize different cases. A novel memory access graph model, which simultaneously captures potential energy savings as well as potential performance improvements, is proposed to overcome the weakness of previous techniques. Based on this model, we have devised an iterative technique to find best energy saving while satisfying the performance constraint. Experimental results show that our technique can achieve an average code size improvement of 14.15% over the unoptimized programs (for benchmarks in DSPStone [101]) and 7.71% over the programs optimized by the original SPAM compiler [74]. Code size improvements translate directly to shorter execution times. Such improvements are quite significant compared with those obtained by existing approaches. In terms of energy saving from memory modules, our results on average outperform those from SPAM by around 47%. The experiments on the benchmark programs also show that our algorithm runs much more efficiently than the original algorithm of SPAM.

159
8.1 Problem Formulation and Related Work

In this section, we briefly discuss essential features of the non-orthogonal architecture. We then formulate our problem and review related work.

8.1.1 Target Architecture and Problem Formulation

Our target architecture consists of multiple memory banks and a heterogeneous register set. Associated with each memory bank is an independent set of address bus, data bus and address generation unit (AGU). Figure 8.1 shows an example of such an architecture, that of Motorola DSP56000. DSP56000 has three sets of register files (\{X0, X1\}, \{Y0, Y1\}, \{A, B\}) and two memory banks (X, Y). We will use this architecture in our experiments. However, our algorithm can be easily extended to architectures with a homogeneous register set or more memory banks.

We consider memory modules used in the memory banks to have two operating modes, i.e., the active mode and the low-current mode (standby or sleep) [52]. The operating mode transition is controlled by memory controller, whose states can be modified through a set of configuration registers [21]. The detailed discussion of controlling memory operating mode transition is beyond the scope of this thesis. In the active mode, a memory module performs normal read/write while in the low-current mode, the memory module does not perform any memory operation and consumes much lower current than in the active mode. A memory module can switch between the two operating modes by incurring certain time overhead. The memory module supply current during the mode transition is the same as the active mode. For instance, for a Rambus RDRAM module [65], it takes a negligible amount of time to switch from the active mode into the standby (low-
current) mode with the dynamic energy consumed in a cycle\(^*\) being reduced from 3.57 nJ to only 0.83 nJ, but it takes 2 clock cycles to switch back to the active mode. For a Micron SyncBurst SRAM module [52], it takes 2 cycles to switch the module from the active mode into the snooze (low-current) mode with the dynamic energy consumed in a cycle\(^†\) changing from 5.61 nJ to only 0.17 nJ, and it takes another 2 cycles to switch back to the active mode. Clearly, in order to save energy by putting a memory module in the low-current mode, the consecutive idle time should be long enough to compensate for the transition time overhead. Furthermore, it is more beneficial to lump the idle times to a single long idle period than disperse them. This presents some unique challenges to the problem we want to solve, which is formally defined as follows.

\(^*\)The dynamic energy in a cycle is obtained from the measured supply current values associated with memory modules documented in the data sheets (for a 3.3 V, 2.5ns cycle time, 8 MB module).

\(^†\)The dynamic energy in a cycle is calculated for a 3.3V, 5.0ns cycle time, 1 MB module.
**Definition 17** Given a program (in the form of an intermediate code) and a non-orthogonal architecture specification, generate an instruction schedule which maximizes the memory operation parallelism and energy saving.

It is not difficult to envision that increasing parallelism could have an adverse effect on energy saving. Our goal is to devise a methodology to trade off performance, i.e., operation parallelism and energy saving in the Pareto optimal solution set.

8.1.2 Related Work

To our best knowledge, no existing work has investigated the problem defined in Definition 17. However, a number of researchers have studied the different aspects of this problem, e.g., maximizing memory operation parallelism, exploiting the memory module operating modes, etc. We briefly review them below to help clarify our unique contributions.

8.1.2.1 Related work on operation parallelism

Previous related work can be roughly divided into two main categories: those that use compacted intermediate code as the starting point (e.g., [17, 45, 71, 75]), and those that start with uncompacted intermediate code (e.g., [41, 100]). *Compa**cted* intermediate code refers to the intermediate code that is compacted or scheduled by some heuristics such as list scheduling, to increase the instruction level parallelism without considering the data dependency. Since scheduling is done prior to exploring memory bank assignments, it is obvious that some memory-operation-pair combinations may be left out of consideration no matter which heuristic is used to compact the code. Thus, the approaches in the first
category often fail to exploit many optimization opportunities. Techniques in the second category overcome this problem by using the uncompacted code to explore all possible pairs of memory operations as long as there are no dependencies between them. Therefore, We adopt the same philosophy as these techniques, i.e., starting with the uncompacted code.

Most existing techniques to explore parallelism adopt some graph model for variable partitioning. A major distinction between those techniques lies in the graph model definition. Reviewing these graph models can help explain why these models are not adequate.

Given a program represented by a control data flow graph (CDFG), an undirected graph can be constructed to model the relationship among the variables in the program. The nodes in the graph represent all the local variables stored in memory. Partitioning the nodes in the graph into different groups then leads to partitioning the corresponding variables to different memory banks. The effectiveness of such an approach relies on modeling edge weights properly to capture all relevant information.

A straightforward way of assigning edge weights is to connect two nodes with an edge of weight 1 if the two corresponding variables do not have data dependencies and the memory operations involving the variables can potentially overlap [41] (as accessing such two variables in parallel may decrease the schedule length). However, such potential parallelism may not be always realizable due to certain timing constraints on the associated memory operations. (Recall that the operations are to be scheduled later for uncompacted code.)

The authors in [100] introduced the concept of possibility weight to capture the likelihood of parallelizing pairs of instructions. The model does improve on
the simple graph model above, but it still has some deficiencies. For instance, to
derive the edge weight between a pair of variables, they simply sum the possibility
weights of all pairs of memory operations involving this variable pair in the entire
procedure. Simply adding the possibility weights from different pairs of memory
operations cannot correctly capture the scheduling freedom difference between the
operation pairs. We will discuss these deficiencies in more detail in Section 4.

None of the existing graph models consider how to exploit serialism in in-
struction execution to trade off performance for energy saving. In this chapter,
we propose to use two lists to describe the edge weight in the graph model. By
introducing one more dimension to the graph edge weight, we not only capture
the serialism information among operations, but also overcome the deficiencies of
previous models.

8.1.2.2 Related work on energy saving

A number of research results have been published regarding saving energy
through exploiting operating mode changes. The key idea is to distribute idle
times judiciously through good scheduling. This can be achieved at various ab-
straction levels or design stages. For example, an on-line, low-power, task schedul-
ing algorithm for multiple devices is presented in [46]. An operating system (OS)
based solution is proposed in [22] where the OS scheduler manages power mode
transitions by keeping track of module accesses for each process in the system.
Several papers have been published to exploit the benefit of memory operating
mode control. In [21], a compiler-directed scheme is presented to reschedule the
basic blocks such that longer consecutive memory idle times can be obtained. The
techniques in [8, 47] consider data organization in multi-bank memory such that
data accesses can be concentrated in a small number of banks while other banks can be left in the low-current mode. No instruction scheduling is considered by these works.

Our work focuses on the instruction level. By integrating energy consideration into the instruction scheduling stage, we can achieve additional energy saving without sacrificing performance. Note that our work complements the above mentioned energy saving techniques since it can be applied together with these other techniques.

8.1.2.3 Other related work

Some research related to multiple memory banks concerns with memory partitioning [8, 96]. Given the memory access pattern of a class of programs, memory partitioning finds the best memory bank configuration, e.g., the number of memory banks, the size of each bank, the number of ports for each bank, etc., from the viewpoint of instruction level parallelism or energy saving. It is a different problem from the one considered in this chapter in that the memory configuration is given in our architecture model, and we concentrate our effort on variable partitioning and instruction scheduling among memory banks. It is worth noting that the work in [96] deploys the model of conflict graph and conflict probability, which derives the graph information from uncompacted intermediate code. Though it addresses a different problem, it also shows that working on uncompacted code reveals more optimization opportunities.
8.2 Idle Time Exploration

As mentioned earlier, memory operating mode transition does not come for free. Extra clock cycles are needed to change between the active and low-current modes. Therefore, to exploit the low-current mode, longer consecutive idle times are more desirable for a memory bank. However, variable partitioning and instruction scheduling with only performance considerations may not lead to the best schedule in terms of idle time distribution. For example, for the data flow graph (DFG) in Fig 8.2(a), a schedule with only performance consideration is shown in Fig 8.2(b), while better schedules with respect to both performance and energy are shown in Fig 8.2(c) and 8.2(d). In Fig 8.2(b), the memory modules cannot be switched to the low-current mode because all idle times are too short. In the latter two schedules, the idle slots are put together such that one or more memory modules can change to the low-current mode during the idle periods. In Figure 8.2(c), both memory banks can be put into low-current mode during the control steps $3 \rightarrow 5$ under the assumption that Rambus RDRAM is used, while in Figure 8.2(d), the second memory bank can be put into low-current mode during the control steps $4 \rightarrow 6$ under the assumption that a Micron SRAM module [52] is adopted. Thus, we gain energy saving without affecting the schedule performance.

Memory operation scheduling for energy saving is tightly related to that for maximum parallelism, but their different goals can lead to totally different schedules. For example, one could easily sacrifice all the parallelism by putting all variables in one memory bank, which gives the longest idle times for other memory banks. Therefore, a tradeoff exists between energy saving and performance. We consider the problem of maximizing the energy saving without degrading the performance (i.e., program execution time).
Figure 8.2. (a) An example DFG, where $L$ (resp., $S$) followed by an integer $i$ represents a $LOAD$ (resp., $STORE$) operation on variable $i$. Other nodes are non-memory operations. Edges denote the precedence constraint between operation. (b) Schedule with only performance consideration. (c) Schedule when mode transition time is 2 cycles. (d) Schedule when mode transition time is 4 cycles.
In the following, we examine an ideal scenario in which no register constraint exists. In other words, all the variables can be loaded at the earliest time and stored at the latest time. The importance of this case will become clear in Section 5, where we will show that an operation schedule can be regarded as the ideal scenario after the mobility is calculated with the register constraint in mind. Given a control data flow graph (CDFG) representing the behavior of a program segment, assume that the desired schedule length is $t$, the number of memory operations in the $i^{th}$ memory bank is $n_i$ and the overhead for memory module mode transition is $m$ clock cycles. For a given $t$, there exist three cases depending on the relationship of $t$, $n_i$ and $m$.

**Case 1**: $\min (t - n_i) > m, \forall i$

Maximal energy saving can be readily achieved by Lemma 28. The correctness of Lemma 28 is easy to prove and is omitted.

**Lemma 28** If $\min (t - n_i) > m, \forall i$, by simply pushing the LOAD (resp., STORE) operations to the beginning (resp., end) of the schedule, the maximal energy saving is achieved.

The schedule in Fig 8.2(b) belongs to this case when the operating mode transition time is 2 cycles. The schedule with optimal energy saving can be readily obtained by Lemma 28 and is shown in Fig 8.2(c).

**Case 2**: $\min (t - n_i) \leq m, \exists i$ and $t \geq \frac{n + m}{N}$

In the above conditions, $N$ denotes the number of memory banks, and $n$ is the total number of memory operations, i.e., $n = \sum_{i=1}^{N} n_i$. These two conditions mean that consecutive idle times, which are long enough to change memory module to low-current mode, can be formed in some but not all of the memory banks. To improve energy saving, one may consider moving memory operations between
banks to serialize more operations in one or more banks while leave other banks with longer idle times. The goal is then to maximize “serialism” without degrading the performance. The example in Figure 8.2(d) illustrates such thought for the SRAM memory module. The desire to increase serialism in this case complicates the variable partitioning problem.

**Case 3:** \( t < \frac{n+m}{N} \)

\( N \) and \( n \) have the same meaning as in Case 2. No further optimization can be obtained in this case. So long as the schedule length is maintained, not enough idle time can be formed in any memory module.

For a given problem, deciding the schedule length is not an easy task. Even if we have a schedule, Case 2 still presents quite a challenge. In the following, we present our approach to tackle the problem.

### 8.3 Graph Modeling Approach

As mentioned in Section 2.2.1, the edge weight assignments introduced in the previous works all fail to capture some important information, which may lead to suboptimal solutions. In this section, we describe our edge weight assignment approach by examining the requirement of a desirable graph model and analyzing deficiencies of previous graph models [100].

The construction of the graph model is based on the CDFG representation of an application. The information about memory-operation scheduling freedom can be derived from CDFG and fed into a later stage to assign the graph edge weights.

From CDFG representation of a program, one can readily derive both the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules, considering the constraints of computation units. Let the control steps of a memory operation,
a, be \( t_s(a) \) and \( t_t(a) \) according to ASAP and ALAP, respectively. The mobility, i.e., the scheduling freedom of \( a \), defined as \([t_s, t_t]\), represents the time interval in which \( a \) can be scheduled without introducing additional delay. Only when the mobilities of two memory operations have some overlap may parallelizing the two corresponding variables be beneficial (in terms of improving performance).

Clearly, the larger the overlap between two mobilities, the higher the potential of the two variables being able to be parallelized. If the mobilities of two operations are both small and their overlap is relatively large, parallelizing the corresponding variables is more likely to improve the schedule length. In other words, if such variables are put in the same bank, accessing the two variables is forced to be sequentialized which is very likely to increase the overall schedule length. The work in [100] assigns a possibility weight defined below to an edge to model this property.

**Definition 18** Given two memory operations, \( a \) and \( b \), let their mobilities be \([t_s(a), t_t(a)]\) and \([t_s(b), t_t(b)]\), and the maximum overlap between these two mobilities be the interval \([t_1, t_2]\). The possibility weight assigned to the edge between the two variables accessed in operations \( a \) and \( b \) is \( \frac{t_2 - t_1 + 1}{(t_t(a) - t_s(a) + 1)(t_t(b) - t_s(b) + 1)} \).

Fig 8.3(b) shows an example of this possibility weight assignment for the memory operations given in Fig 8.3(a) which captures the memory operation mobilities for the DFG in Fig 8.2(a). In Fig 8.3(a), there are 6 variables and 8 memory operations. The line beneath or above each operation represents its mobility. For example, \( L3 \) has a mobility of \([0, 1]\). In Fig 8.3(b), more than one possibility weight may be associated with an edge. These come from different pairs of memory operations. For instance, between variables \( v_1 \) and \( v_5 \), \( 1/5 \) comes from \((L1, L5)\) pair, while \( 1/20 \) from \((L5, S1)\) pair. In [100], such numbers are simply added together.
Figure 8.3. (a) Memory operation mobility graph for DFG in 8.2(a) (b) Memory Access Graph with the first dimensional weight, each node number corresponds to the associated variable number. (c) Memory Access Graph with the second dimensional weight.

Moreover, if the same operation pattern (e.g., overlapping of $L_1$ and $L_5$) occurs in another mobility range\(^\dagger\), the possibility weight is again added to the edge weight between $v_1$ and $v_5$.

One deficiency of the possibility weight model in [100] is associated with simple summation of the possibility weights mentioned earlier. Consider a simple example of an edge possibility weight of 1. This may come from two operations such as $L_1$ and $L_2$ in Fig 8.3(a). It may also come from two occurrences of the operation pair such as $L_3$ and $L_4$ in Fig 8.3(a). Though both edges have a weight of 1, it is not difficult to see that the variables in the first case should be given a higher priority to be parallelized since the schedule length will definitely be increased if the two variables are put in the same memory bank (while the variables in the second case have an additional slack cycle). To overcome this problem, we advocate to maintain as a list the possibility weights from different operations involving

\(^\dagger\)A mobility range is a period of consecutive scheduling steps which may cover several variables’ mobility. In this chapter, whenever we talk about two different mobility ranges, we refer to independent non-overlapping mobility ranges.
the same variable pair instead of adding them together. (We will discuss how to manipulate this list later). For example, a list, which contains two elements, i.e., \( \frac{1}{5} \) and \( \frac{1}{20} \), is maintained to reflect the parallelism weights for edge between variables \( v_1 \) and \( v_5 \) in Figure 8.3(b).

Another problem with the possibility weight model is that it does not distinguish mobility overlaps within a single mobility range from those in different mobility ranges. Consider the following example. Given three memory operations, \( a, b, c \), in one mobility range, each has the same mobility \([0, 1]\). The corresponding graph model contains an edge with weight \( \frac{1}{2} \) between the variables in \( a \) and \( b \) and between those in \( a \) and \( c \). Assume in the same procedure, memory operations, \( a', b' \) in a different mobility range, have the mobility \([3, 4]\), and memory operations, \( a', c' \) have the mobility \([7, 8]\) in yet another mobility range. Then, the associated graph has an edge with weight \( \frac{1}{2} \) between the variables in \( a' \) and \( b' \) and between those in \( a' \) and \( c' \). Obviously, variables in \( a, b, c \) should be given a higher priority to be parallelized than variables in \( a', b', c' \) since putting the former in one memory bank will definitely introduce an additional delay (while putting the latter in the same bank does not necessarily introduced additional delay since operations on variables in \( a', b' \) are in a different mobility range from those on variables in \( a', c' \)). However, the model in [100] treats the two groups indiscriminately.

Besides the above shortcomings, the possibility weight model have no consideration about energy saving because the work in [100] focuses only on performance. From the point of view of energy saving, one would prefer to serialize memory operations as much as possible so as to leave more “long” idle intervals for the low-current mode (see the discussion of Case 2 in Section 3). Clearly, this preference toward serialism may run against the requirement of improving performance.
To capture the trade-off between the desire of parallelism and that of serialism, we propose to use two lists of weights. The first one is the one discussed above, i.e., the list of possibility weights, which are referred to as *parallelism weights*. The second one is a new one and the weights are referred to as *serialism weights*. The goal of serialism weights is to model the possibility of serializing a pair of operations without sacrificing performance. To derive the serialism weight, observe that given a certain mobility range, the more operations in the range, the more difficult it is to serialize the operations without increasing the total delay. Take the example above, serializing three operations $a, b, c$ increases the schedule length, while serializing $a', b' \text{ (or } a' \text{ and } c')$ has no negative effect. Based on this observation, we formally define the serialism weight as follows.

**Definition 19** Assume the mobilities of two memory operations, $a$ and $b$, are $[t_s(a), t_l(a)]$ and $[t_s(b), t_l(b)]$, respectively, their union is $[t_1, t_2]$, and the number of operations whose mobilities are contained in $[t_1, t_2]$ is $n$. The serialism weight for the edge between the variables accessed in $a$ and $b$ is $\frac{t_2 - t_1 + 1}{n}$.

An example of the serialism weight is shown in Figure 8.3(c). Note that similar to the parallelism weights, more than one serialism weight may be associated with an edge due to the multiple occurrences of the memory operations involving the corresponding variable pair. Take the serialism weight between variables $v_1$ and $v_5$ as an example, $\frac{5}{6}$ comes from $(L_1, L_5)$ pair (five operations exist in mobility range $[0, 5]$ which is the shortest range to cover both $L_1$ and $L_5$’s mobility), while $\frac{9}{8}$ comes from $(L_5, S_1)$ pair (eight operations exist in mobility range $[0, 8]$). A list is also maintained for each edge to record these serialism weights.

We now formally define the *Memory Access Graph* (MAG) used in our approach.
**Definition 20** A Memory Access Graph (MAG), $G = (V, E, F)$, is a multi-weighted undirected graph, where $V$ is the set of nodes representing the variables in the given code, $E \in V \times V$ is the set of edges, $F = (\vec{w}_p, \vec{w}_s)$ is a function from $E$ to $R^{2m}$ representing the weight lists between the corresponding two nodes, $\vec{w}_p$ and $\vec{w}_s$ are the parallelism and serialism weight lists, respectively.

Though we are able to capture the requirements of performance and energy saving through introducing both parallelism and serialism weights, we need to be able to use them effectively in partitioning the variables. The problem of variable partitioning for $k$ memory banks is equivalent to the maximum $k$-cut problem, which is NP-complete [5]. A number of excellent heuristics exist for solving the maximum $k$-cut problem [5]. To use such heuristics, we need to reduce the two lists of weights associated with an edge to a single weight value. To reflect the trade-offs between performance and energy saving, we use a weighted sum formula to compute the average weight of an edge. Specifically, the average weight of an edge $e(i, j)$ is defined as

$$w(i, j) = \sum_{h=1}^{m(i, j)} \lambda_p w_p(h, i, j) - \lambda_s w_s(h, i, j)$$

(8.1)

where $\lambda_p, \lambda_s$ are two coefficients representing the trade-offs between parallelism and serialism, $w_p(h, i, j)$ (resp., $w_s(h, i, j)$) is the parallelism (resp., serialism) weight associated with the $h^{th}$ pair of operations involving variables $i$ and $j$ and $m(i, j)$ is the total number of such pairs. The rationale behind the subtraction used in (Eqn. 1) is that $w_p(h, i, j)$ and $w_s(h, i, j)$ can be viewed as measures of two opposite forces, parallelism and serialism. Different coefficient values, $\lambda_p$ and $\lambda_s$, reflect the preference between the two forces, and hence help trade off performance with energy saving.
As we can see from the definition, the parallelism weight can be regarded as the local measurement which captures the scheduling information between two memory operations, while the serialism weight can be regarded as the global measurement which reflects the scheduling freedom for memory operations belonging to the same mobility range. The integration of these two factors through (Eqn. 1) provides a complete measure of benefit which can be obtained from a certain variable partition.

Each member of $\vec{w}_p$ is always smaller than 1, while that of $\vec{w}_s$ may be larger than 1. In order to ensure these two values are in the same range, each member $w_s(i, j)$ is normalized with the formula $w_s(i, j) = \frac{w_s(i, j) - w_{\text{min}}}{w_{\text{max}} - w_{\text{min}}}$, where $w_{\text{min}}$ (resp., $w_{\text{max}}$) is the minimum (resp., maximum) value of all $w_s(i, j)$ values in the corresponding mobility range to which this $w_s(i, j)$ belong. Keeping a linked list to record all weights enabled us to do this normalization on the basis of mobility range.

It is important to point out that the average weight defined in (Eqn. 1) indeed can overcome the shortcomings mentioned earlier in the model introduced in [100]. Consider the shortcoming associated with simply adding possibility weights for the two discussed scenarios that the possibility weight is not able to distinguish, under our average weight model, the edge weight in the first case is $(\lambda_p - \frac{1}{2}\lambda_s)$, while the edge weight in the second case is $(\frac{1}{2}(\lambda_p - \lambda_s) + \frac{1}{2}(\lambda_p - \lambda_s))$. (We assume that no other variables have operations overlap with the mobilities under consideration.) Given the same $\lambda_p$ and $\lambda_s$ values, the former is always greater than the latter, which correctly reflects the fact that it is more beneficial to put the two variables in the first case to separate memory banks as they have more stringent timing requirement. For the shortcoming due to mobility range differentiation, according
to our model, the average weight on the edge between $a$ and $b$ and that between $a$ and $c$ is $(\frac{1}{2}\lambda_p - \frac{2}{3}\lambda_s)$, while the average weight on the edge between $a'$ and $b'$ and that between $a'$ and $c'$ is $(\frac{1}{2}\lambda_p - \lambda_s)$, (assuming no other operations overlap these mobilities). Again, the edges between $a$, $b$ and $c$ have a larger weight for a given pair of coefficients, and hence the variables associated with these operations are favored for putting into separate banks (which is exactly what one would like to see).

There are still several unanswered questions. For example, how should one select the coefficients in the average weight definition for a given application? Also, how does one handle the register constraints? We shall discuss these issues in the next section where we present our complete algorithm.

8.4 Algorithm

Our variable partitioning and instruction scheduling algorithm is intended to be used in the back end of a compiler to optimize the intermediate code. The algorithm operates on the CDFG representation of a given piece of intermediate code. As the first step, it calculates the mobility for each operation with the register constraint in mind. Then, the MAG is constructed based on the mobility information. With this MAG, the steps of average weight calculation, maximum cutting, variable partitioning and instruction scheduling are iterated for a number of times to find the best values of $\lambda_p$, $\lambda_s$. The algorithm framework is shown in Algorithm 11.

In Algorithm 11, $T_{\text{schedule}}$ represents the number of consecutive idle cycles for the current schedule, while $T_{\text{max}}$ represents the maximal value of all $T_{\text{schedule}}$. $\lambda'_s$ is used to remember the value of $\lambda_s$ in the previous loop iteration.
Algorithm 11

**Input:** Intermediate Code, Register Constraints

**Output:** Code optimized for energy and performance

1. Derive the CDFG from the intermediate code. Calculate the mobility for each operation.
2. Construct the memory access graph (MAG) //refer to Section 4
3. $\lambda_p = 1, \lambda_s = 0, \lambda'_s = 0$§, calculate the average weight for each edge and schedule the program
4. Set the minimum schedule length $L_{min}$ as the schedule length of the current schedule, $T_{max} = 0$.

**WHILE** ( ) **do**
5. Find the Maximum Cut. Allocate variables to memory banks according to the cut result.
6. Schedule the program according to the above allocation result while maximizing the consecutive idle time. Set $L_{schedule}$ and $T_{schedule}$ //refer to Section 3
7. **if** $L_{schedule} - L_{min} \leq \phi$ and $T_{schedule} - T_{max} \leq \eta$ **then**
   - $N_{stable} = +$;
   - Record the corresponding variable partition and schedule;
**end if**
8. **if** $L_{schedule} - L_{min} \leq \phi$ and $T_{schedule} - T_{max} > \eta$ **then**
   - $L_{min} = \min(L_{min}, L_{schedule})$,
   - $T_{max} = \max(T_{schedule}, T_{max})$,
   - $\lambda_s = \frac{\lambda_p + \lambda_s}{2}$,
   - $N_{stable} = 0$
**end if**
9. **if** $L_{schedule} - L_{min} > \phi$ **then**
   - $\lambda_s = \frac{\lambda_s + \lambda'_s}{2}, \lambda'_s = \lambda_s, N_{stable} = 0$
**end if**
10. **if** $N_{stable} \geq \sigma$ **then**
    - break;
**end if**
11. Recalculate the average weight of MAG.

**ENDWHILE**
12. Output the corresponding variable partition and schedule
$L_{\text{schedule}}$ and $L_{\text{min}}$ represent the current and minimum schedule lengths, respectively. $\phi$ is a user specified parameter to indicate the latency constraint and defined as the allowed difference between the final and minimum schedule lengths. $\eta$ is a user-defined threshold to measure whether $T_{\text{schedule}}$ has a significant change. The algorithm will finish after $T_{\text{schedule}}$ have not shown significant changes for $\sigma$ number of loops.

In Line 1 of Algorithm 1, we use the technique in [99] to deal with the heterogeneous register set and register constraint. By dividing the register mapping into two stages, register allocation (before scheduling) and register assignment (after scheduling), the algorithm in [99] obtains the benefit, but avoids the difficulty of considering register mapping and scheduling together. A heterogeneous register set is dealt with by transforming physical registers into virtual registers such that all virtual registers can be regarded as homogeneous. The concept of virtual registers provides a powerful methodology to check if a feasible register assignment exists for a specific schedule without the necessity to generate one. This allows the flexibility of considering the register constraint during scheduling, by simply checking if enough virtual register resources are available in each schedule step. Lots of efforts can be saved by determining the detailed register assignment for the final schedule instead of every possible intermediate schedule. The approach in [99] is similar to the register class concept in [37], but with the advantage that the number of available virtual registers can be derived to constrain the mobility of each variable.

In Line 5 in Algorithm 1, the well known maximum spanning tree (MST) algorithm [62] is used as the maximum-cut heuristic. Then variables are allocated to the memory banks under the rule that two variables having a MAG edge belong-
ing to the cut, must be in different banks. Note that any heuristic maximum-cut algorithm can be used at this step. MST algorithm is preferred because of its popularity and easy implementation.

The WHILE loop of the algorithm is used to find a point where the maximal energy saving is achieved for the desired performance. Because of the opposite forces of parallelism and serialism, more parallelism (larger $\lambda_p$ and smaller $\lambda_s$) may bring better performance and less energy saving, while more serialism (smaller $\lambda_p$ and larger $\lambda_s$) may bring more energy saving, but a possible deteriorated performance. Therefore, we introduce a process analogous to the binary search into the algorithm, trying to reach the best trade-off point, i.e., a set of $\lambda_p$ and $\lambda_s$ values to achieve maximal energy saving for a given desired performance. In the 8th step, if the performance is still in the desirable scope and more energy saving can be achieved through the last change of $\lambda_s$, we then push $\lambda_s$ further toward the direction of serialism in the hope of getting more energy saving without degrading the performance. On the other hand, if the performance degrades too much, we move back $\lambda_s$ toward parallelism in the 9th step in the hope of recovering the performance but still maintaining the gained energy saving. Finally, when the alteration of $\lambda_s$ becomes too small to bring any meaningful change on either performance or energy saving, the algorithm exits from the WHILE loop.

The complexity of the algorithm depends on two factors, the schedule length ($L$) and the number of variables ($N$). In the WHILE loop, step 6 takes $O(L^2)$ time, while MST algorithm can be done in $O(N^2 \log N)$ since there are at most $O(N^2)$ edges in an MAG graph. Therefore, the algorithm complexity is $O(w(L^2 + N^2 \log N))$, where $w$ is the number of iterations of the WHILE loop body. It is shown in the experimental section that $w$ is normally quite small for reaching the
The algorithm can be easily extended to other systems with different architectural parameters. For instance, if a system has a different register file set, the technique in [99] can still be used. The only difference will be the number of available virtual resources. By replacing the MST algorithm with some polynomial maximum k-cut heuristic [32], this algorithm framework can be extended to the system with more memory banks.

MOVE main.(y) R0
MOVE main.(ph), R1
MOVE main.(px), R2
MOVE main.(px2), R3
DO #15 TMP14
MOVE X:(R1)-, X0
MOVE X:(R2), X1
MOVE X:(R0), A
MAC X0 X1 A
MOVE A X:(R0)
MOVE X:(R3)-, X0
MOVE X0, X:(R2)-
TMP14:

MOVE main.y R4
MOVE main.(ph), R0
MOVE main.(px), R1
MOVE main.(px2), R2
MOVE Y:(R4) A
DO #15 TMP14
MOVE X:(R0)-, X0
move X:(R1), X1
MAC X0 X1 A X(R2)-, X0
MOVE X0 X:(R1)-
TMP14:
MOVE A Y:(R4)

(a) (b)

Figure 8.4. An example to illustrate the result of applying Algorithm 1. (a), The original assembly code given by SPAM compiler. (b). The assembly code after applying Algorithm 1.

Fig 8.4 shows an example assembly code which is the loop segment of FIR filter
in DSPStone benchmark suite [101]. Fig 8.4(a) is the assembly code obtained from SPAM. Fig 8.4(b) is the result after applying Algorithm 1, shown in the instruction format of opcode, operands, and two possible parallel move fields. There are 10 nodes and 34 edges in the MAG graph for the FIR filter. It takes 4 iterations to reach the final result. The reader is referred to [83] for the complete code example. In this example, variable y is put into Y memory bank due to the global variable partition consideration. The loop body length is reduced from 7 to 4 clock cycles. The improvement is achieved by increasing the program parallelism (see instruction 9 in Fig 8.4(b)) and memory operation scheduling (moving the operations of loading (resp., storing) variable y to the beginning (resp., end) of its mobility according to Lemma 3.1, thus out of the loop body). The final code is a trade-off between energy saving and schedule length. If the performance is the only emphasis, the loop body can be further reduced to 3 clock cycles by moving data array px to Y memory bank.

8.5 Experimental results

We have implemented our algorithm in the SPAM compiler environment to replace the simulated annealing algorithm [75] originally used by SPAM project [74]. The Benchmarks used are from DSPstone benchmark suite [101], which contains C source code for various DSP kernels: Least Mean Square (B1), FIR (B2), N Real Update (B3), IIR Biquad (B4), Convolution (B5), N Complex Update (B6), 2-Dimensional FIR (B7), Matrix Multiplication (B8), 1st Adapted Predictor (B9) and Tone Detector (B10) routine in ADPCM. The intermediate code is generated by SUIF front end followed by SPAM code generation program, then fed as input to our algorithm to obtain the optimized assembly code.

181
Figure 8.5. Assembly code size results. The results are normalized with respect to the SPAM result.

Figure 8.6. Execution time of assembly code. The results are normalized with respect to the SPAM result.
The assembly code size results are shown in Fig 8.5. The data include the original code size (Original), code size generated by the constraint-graph method (SPAM), code size generated by [100] (Inde Graph\(^\dagger\)) and code size generated by our algorithm (VPIS). Fig 8.5 reveals that the methods of independence graph and our algorithm can both perform better than SPAM. This improvement can be attributed to exploiting more potential memory operations parallelism. Accredited to our comprehensive graph model and judicious selection of weight coefficients, our algorithm demonstrates a superior performance to the method of independence graph, as demonstrated in Fig 8.5. The execution time of the assembly code is correlated to the code size [41], since the assembly code can be directly mapped to the schedule for the basic block. Moreover, due to the existence of loops is DSP benchmark, we are able to observe even larger improvements when comparing the execution time of assembly code. The results are shown in Figure 8.6.

We compare the energy saving results of our algorithms with SPAM. Results from Inde Graph methods are not included in this comparison, since it does not have the energy saving consideration. In fact, it can be regarded as a special case of our algorithm with the restriction of \(\lambda_p = 1, \lambda_s = 0\). As a simple comparison, we examine the generated assembly code. By counting the number of consecutive idle cycles (must be larger than the operating mode transition time) for the two schemes in each basic block, the absolute number of idle cycles in which the memory module can be put into low-current mode is obtained by summing all such numbers in the entire procedure. The ratio of these idle cycles to the overall code size can be calculated. The average improvement of this ratio is 19.84%. This ratio comparison can give us the initial estimation of at least how much improve-

\(^\dagger\)Their paper uses a greedy heuristic algorithm, similar to MST algorithm to partition the variables. Their results reported in this section are obtained from MST algorithm for the sake of fairly comparing graph model.
ment can be achieved from the algorithm. With the multiple execution times of
loop bodies, a larger improvement should be expected, which is demonstrated in
the following comparison.

As a more elaborate comparison, we have simulated the execution of the gen-
erated assembly code with Sim56000 (Motorola’s DSP56000 simulator). From the
run profile, all the usable idle times are added together to get the total idle cycles
during which the memory module can be put into low-current mode. By dividing
this total idle cycles by the benchmark’s total execution clock cycles, the ratio
of memory energy saving are derived for all benchmarks. Note the upper bound
for this ratio equals to the number of memory banks. The results are shown in
Figure 8.7.

Our algorithm achieves larger improvements for data-intensive (e.g., computa-
tion intensive loop body) than control-intensive (e.g., procedure call, procedure
initialization) application code, since data-intensive code involves more ALU op-
erations which operate only on the register file. Furthermore, data intensive code
are usually executed many times such as the computation loops in DSP applica-
tions. Based on these facts, we can see larger energy saving in Figure 8.7 than
the initial estimation given above. The average improvement of our approach over
SPAM is 47.55%.

One concern may be raised about the algorithm execution time because of the
loop in the algorithm to find the best trade-off point. Due to the fact that variable
partitioning is not very sensitive to the change of average coefficients \( \lambda_p \) and \( \lambda_s \)
(small change in these two coefficients does not change the variable partition), the
algorithm generally can find the best trade-off point in at most twenty loops. We
ran the program in SUN Ultra Sparc2 and the algorithm execution time compar-
Figure 8.7. Percent of low-current cycles over the total execution clock cycle

is shown in Figure 8.8. In the figure, we normalized the algorithm execution time by the simulated annealing (SA) algorithm (adopted by SPAM) execution time which is given in the unit of second on the top of each benchmark.

This figure shows that our algorithm takes much less time than the SA-based algorithm. Moreover with more complicated programs, the constraint graphs in SPAM become larger and each step in the annealing process takes longer time. The SA algorithm execution time increases significantly with the increase in the constraint graph size. While our algorithm, by contrast, does not have to deal with the large graph for many times (at most twenty loops for our experiments). Therefore, the execution time improvement becomes more obvious for larger benchmarks.
8.6 Conclusion

A variable partitioning and instruction scheduling algorithm is proposed to exploit the architecture with multiple memory banks and heterogeneous register set. The algorithm takes into account both instruction level parallelism and reducing system energy. A novel graph model is presented to capture both parallelism and serialism scheduling information. With such a model, the maximum instruction level parallelism can be exploited to improve the schedule performance. The idle intervals of the memory module are maximized under the constraint of the schedule performance, such that the system energy is reduced by changing memory modules to low-current mode for longer time. Experimental results demonstrate that our algorithm outperforms the previous techniques.

As future work, the novel graph model presented in this chapter can be extended to other architectures besides multiple memory bank architecture, such as clustered architecture and distributed systems. One common characteristic for all
these systems lies in the trade-off between parallelism and serialism because of energy saving consideration, resource constraints, etc. For example, in a clustered architecture, it is important to balance the workload to all clusters to increase the performance. On the other hand, to reduce energy consumption, it is desirable to reduce the bus communication between clusters and put some clusters in low-power mode. Therefore, a graph model which can capture all such information is essential to a good scheduler. How to extend our proposed graph model to cope with other architectures is worth more research effort.
CHAPTER 9

FUTURE WORK

This thesis presents our work on software partitioning and scheduling optimizations. The algorithms targeting different architecture models are elaborated. To measure the performance of our algorithms, we use a set of well known benchmarks. We apply our algorithms and some other well-known or state-of-art algorithms to these benchmarks. The result and their comparison are demonstrated in this thesis.

Another interesting topic in scheduling optimization is the address offset assignment problem. In order to improve the performance, modern embedded processors always have specialized address generation units (AGU) to facilitate the memory address calculation. The AGU normally provides simple address address register operation (typically, plus or minus an immediate value or a value in offset register) in parallel with the memory access operation. Since address generation on AGU does not use datapath resources, a high instruction-level parallelism can be achieved by generating many auto-increment/decrement addressing modes to access variables. With this special feature, comes the problem for compiler how to determine the relative location of variables in memory to maximize the utilization of auto-increment/decrement modes to generate a very compact address code.

A lot of research has been conducted on the offset assignment problem. This problem was first studied by Bartley [7] and Liao et. al [43]. They identified the
problem as two classes. The simple offset assignment (SOA) problem considers only one address register, while the general offset assignment (GOA) problem handles more than one address register. The problem is modeled as a graph and the objective is to find the maximum weight path cover (MWPC), which is proved to be a NP-complete problem. A heuristic is used to solve both SOA and GOA. Leupers et. al. [42] extended Liao’s work by proposing a Tie-break heuristic for SOA and a variable partitioning strategy for GOA. Atri et. ali [4]. further improved the heuristics by algorithm called Incremental-Solve-SOA. Sudarsanam et. al. [76] studied the offset problem in the presence of auto-increment/decrement feature that varies from $-1$ to $+1$ with $k$ address registers. In all these works, the variable access sequence is obtained from the scheduled operations. All algorithms only concentrated on finding the variable layout, so that the least number of extra address calculation operations is needed except auto-increment/decrement. Obviously, more improvement should be expected if we can optimize both the schedule and memory layout.

Some greedy algorithms are presented to optimize both schedule and memory layout. Rao et. al. [66] attempted to reorder the variable access sequence through algebraic transformations on the expression trees. Choi et. al. [18] propose the algorithm to incrementally schedule the operation and determine the memory layout. A common deficiency in these two techniques are their greediness. The decision on schedule is always made based on the local information gathered by the current schedule step. It is very debatable that such algorithms may fail because of the lack of global information.

Combining the the above observations, a good offset assignment algorithm should tightly couples two steps of operation scheduling and memory layout. More
importantly, global variable access information should be extracted beforehand and utilized during the entire process of determining the schedule and memory layout.


193


