Minimizing Average Schedule Length under Memory Constraints by Optimal Partitioning and Prefetching *

Zhong Wang  Timothy W. O’Neil  Edwin H.-M. Sha
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
Email: {zwang1,toneil,esha}@cse.nd.edu
Tel: (219)631-8803  Fax: (219)631-9260

Abstract

Over the last 20 years, the performance gap between CPU and memory has been steadily increasing. As a result, a variety of techniques has been devised to hide that performance gap, from intermediate fast memories (caches) to various prefetching and memory management techniques for manipulating the data present in these caches. In this paper we propose a new memory management technique that takes advantage of access pattern information that is available at compile time by prefetching certain data elements before explicitly being requested by the CPU, as well as maintaining certain data in the local memory over a number of iterations. In order to better take advantage of the locality of reference present in loop structures, our technique also uses a new approach to memory by partitioning it and reducing execution to each partition, so that information is reused at much smaller time intervals than if execution followed the usual pattern. These combined approaches - using a new set of memory instructions as well as partitioning the memory - lead to improvements in total execution time of approximately 25% over existing methods.

*.This work is partially supported by NSF grants MIP95-01006 and NSF ACS 96-12028 and JPL 961097
1 Introduction

Over the last twenty years developments in computer science have led to an increasing difference between CPU performance and memory access performance. As a result of this trend, a number of techniques have been devised in order to hide or minimize the latencies that result from slow memory access. Such techniques range from the introduction of single and multi-level caches to a variety of software and hardware prefetching techniques. One of the most important factors in the effectiveness of each of these techniques is the nature of the application being executed. In particular, compile time information can be particularly effective for regular computation or data intensive applications.

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 paper 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. This transformation (uniformization [7]) greatly reduces the complexity of the problem.

Prefetching schemes based on hardware [6,9,19], software [11,12], or both [5,13,24] have been
extensively studied. In hardware prefetching schemes, the prefetching activities are controlled solely by the hardware. In contrast, software prefetching schemes rely on compiler technology to analyze a program statically and insert explicit prefetch instructions into the program code. One advantage of software prefetching is that much compile-time information can be explored in order to effectively issue the prefetching instruction. Furthermore, many existing loop transformation approaches can be used to improve the performance of prefetching. Bianchini et al developed a runtime data prefetching strategy for software-based distributed shared-memory systems [1]. Wallace and Bagherzadeh proposed a mathematical model and a new prefetching mechanism. A simulation on the SPEC95 benchmarks showed an improvement in the instruction fetching rate [20]. In their work, 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 paper, too many prefetching operations may lead to an unbalanced schedule with a very long memory part. In contrast, our new algorithm gives more detailed analyses of both the ALU and memory parts of the schedule. Moreover, partitioning the iteration space is another useful technique for optimizing the overall system performance.

On the other hand, several loop pipelining techniques have been proposed. For example, 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 [21]. Wolf et al studied combinations of various loop transformation techniques, (such as fission, fusion, tiling, interchanging, etc) and presented a model for estimating total machine cycle time, taking into account software pipelining, register pressure and loop overhead [23]. 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 [14]. Modulo scheduling by Ramanujam [17] is a technique for exploiting instruction level parallelism (ILP) in the loop. It can result in high performance code but increased register requirements [10]. 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 [8,18]. None of the above research efforts, however, includes the prefetching idea or considers the data fetching latency in their algorithms.
DO 10 n1 = 1, N1
DO 20 n2 = 1, N2
    y (n1, n2) = x (n1, n2) + c (0, 1) * y (n1, n2 - 1) + 
    c (0, 2) * y (n1, n2 - 2) + c (1, 0) * y (n1 - 1, n2) + 
    c (1, 1) * y (n1 - 1, n2 - 1) + c (1, 2) * y (n1 - 1, n2 - 2) + 
    c (2, 0) * y (n1 - 2, n2) + c (2, 1) * y (n1 - 2, n2 - 1) + 
    c (2, 2) * y (n1 - 2, n2 - 2)
20 CONTINUE
10 CONTINUE

Figure 1: The IIR filter: (a) Code sequence (b) Equivalent MDFG

Loop tiling is a technique for grouping elemental computation points so as to increase computation granularity and thereby reduce communication time. Wolf and Lam proposed a loop transformation technique for maximizing parallelism or data locality [22]. Boulet et al introduced a criterion for defining optimal tiling in a scalable environment. In his method, an optimal tile shape can be determined by these criteria, and the tile size is obtained from the resources constraints [16]. Another interesting result was produced by Chame and Moon. They propose 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 [2]. Nevertheless, the traditional tiling techniques only concentrate on reducing communication cost. They do not consider how best to balance the computation and communication. There is no detailed schedule consideration in their algorithms.

This paper will combine these 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.

Consider the example in Figure 1. It can be seen as a nested loop coded in a high-level programming language, as well as a typical DSP problem. The Fortran code derived from the IIR
filter equation

\[ y(n_1, n_2) = x(n_1, n_2), + \sum_{k_1=0}^{2} \sum_{k_2=0}^{2} c(k_1, k_2) \ast y(n_1 - k_1, n_2 - k_2) \text{ for } k_1, k_2 \neq 0 \]

is shown in Figure 1(a). An equivalent multi-dimensional data flow graph is presented in Figure 1(b). The graph nodes represent two kinds of operations: nodes denoted by an 'A' followed by an integer are additions, while an 'M' followed by an integer represents 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, while \(d_y\) corresponds to the innermost loop.

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 paper 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 [3] 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 optimizing 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 [15]. This paper 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.

Section 2 will introduce the terms and basic concepts used in the paper. In Section 3, the theoretical concepts that form the basis of the paper are presented. The next section describes the algorithm that will be used to implement the new constructs, while Section 5 presents comparison
of the new technique with a number of existing approaches. A summary section that reviews the main points concludes the paper.

2 Basic Framework

In this section, we review some concepts that are essential to the implementation of the algorithm. The architecture model and the framework of the algorithm are also illustrated in this section.

2.1 Multi-dimensional data flow graph (MDFG)

In a nested loop, an iteration is the execution of the loop body once. It can be represented by a graph called a 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, \( d \) is a function from \( E \) to \( \mathbb{Z}^n \) representing the multi-dimensional delay vector between two nodes, and \( n \) is the number of dimensions (the depth of the 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.

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. All iterations constitute the iteration space, which 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.

2.2 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 \square

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.

2.3 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 3 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\_m or prefetch operation will be used.

**Definition 4** Delay dependencies that go into the next m\textsuperscript{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.

![Different kind of delay dependencies](image1)

![Three different regions](image2)

**Figure 2**: Different kinds of memory operations and its corresponding regions

For instance, for the delay dependencies in Figure 2(a), d\textsubscript{1} needs a keep\_1 operation; d\textsubscript{2} needs a keep\_2 operation; d\textsubscript{3} needs a keep\_3 operation; and d\textsubscript{4} and d\textsubscript{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.

### 2.4 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.

![Figure 3: An example of memory arrangement](image)

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. 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 communi-
cation 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.

2.5 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
   - else, calculate the memory requirement when $f_x = 1, f_y = F_y$. //see section 4.2 and section 2.3
4. If this size is larger than the first level memory constraint
   - then print("no suitable partition exists")
   - stop
5. For each delay vector $d = (d_x, d_y)$, calculate the projection on the x-axis along $P_y$ direction, $ld = d_x - d_y \times \frac{P_u \times}{P_{u-y}}$.
6. Let $f_x = ld$, and for each ld, calculate the memory requirement. //see section 4.4, theorem 13
7. Find the interval whose left endpoint satisfy the memory constraint, but whose right endpoint doesn’t.
8. Repeatedly increase $f_x$ within this interval until it reaches the memory constraint.
9. 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 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 Theoretic foundation

The main theme of this paper 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 4. 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 4: Two adjacent partitions](image)

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

**Definition 5** A basic partition is a parallelogram with the following properties:
1. Two its sides are parallel to the x-axis, each with these length $f_x*|P_x|$.

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

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

**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 $m = \left\lceil \frac{d_x - d_y \cdot \frac{f_x}{f_y}}{f_x} \right\rceil$. 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 $n = \left\lceil \frac{d_x - d_y \cdot \frac{f_y}{f_x}}{f_y} \right\rceil$ 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 operations can be calculated 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(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(a) and 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(a) when $m = 1$. In Figure 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-parallellograms, AJnE and JBFn. 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.

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

**Definition 6** 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., $w = t/m$ 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 $I = n = w \times h$.  

14
For any integral-area keep parallelogram with width $w = t/m$ and height $h = l \cdot m$, 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 $w = p \cdot m$ 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 $p \cdot m$. Thus, the total number of points is $l \cdot (p \cdot m + q) = w \cdot h$.

For ease of notation, we introduce the following definitions.

**Definition 7** Let $\text{frac}(a)$ 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 $\frac{1}{n}$ with $l$ an integer, define

$$
\delta_l(m, n) = \begin{cases} 
\left\lfloor \frac{m}{n} \right\rfloor - 1 & \text{if } \text{frac}(a) \notin [0] \cup (1 - \text{frac}(\frac{m}{n}), 1) \\
\left\lfloor \frac{m}{n} \right\rfloor & \text{otherwise}
\end{cases}
$$

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 $\lfloor \text{length}(K) \rfloor - 1$, 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 $\lfloor \text{length}(K) \rfloor + 1$ points. Thus $K'$ contains exactly $\lfloor \text{length}(K) \rfloor - 1$ points with integer coordinates and $K$ will contain $\lfloor \text{length}(K) \rfloor$ such points. From the definition of $\delta_t$, 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 $m = \left\lfloor \frac{d_x - d_y}{f_x \cdot P_x \cdot f_y} \right\rfloor$, 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, $R1$ 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.

**Proof:** As shown in Figure 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 $mp = \lfloor \frac{d_y}{P_y} \rfloor + 1$. 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 $p = (m \cdot P_x, mp \cdot P_y)$

Once $p$ is known, the first step of the proof is completed by letting $fp = (p \cdot x - d_x, p \cdot y - d_y)$. 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(b), all results within region AJnE are treated with keep\(_m\) \(-\) \(1\) 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 \(m = \frac{d_k - d_y \times \frac{n_{x,y}}{f_{x,y}}}{f_{x,y}}\), 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.

4 Algorithms

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

4.1 Scheduling the ALU

In the ALU schedule, the multi-dimensional rotation scheduling algorithm [15] 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.

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 6 (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 7 (a). The set \{D\} is a down-rotatable set. Retiming node D using the retiming function \( r(D) = (1,0) \) (see Figure 6 (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 7 (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 6 (c), we see that node A can be rotated down and pushed into its earliest control step 4. This result is seen in Figure 7 (c). Thus we can get a schedule with a length of only two control steps.

Figure 6: (a) Initial MDFG (b) After retiming \( r(D) = (1,0) \) (c) After retiming \( r(A) = (1,0) \)

After the schedule for one iteration is produced, we can simply duplicate it for each node in the partition to get the ALU part of the schedule. Suppose the number of nodes in one partition is \#nodes and the schedule length of one iteration is \( \text{len}_{\text{per-iteration}} \). The ALU schedule’s length
will be $\text{len}_{\text{per-iteration}} \times \#\text{nodes}$, which is the least amount of time the program can execute without memory access interference. Therefore, this is the lower bound of the partition schedule. The goal of our algorithm is to make the overall schedule as close to this lower bound as possible while satisfying the first level memory space constraints.

### 4.2 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 8(a) and 8(b). $AB$, $CD$ and $EF$ are parallel to the $x$-axis and have length $f_x - d_x + d_y \frac{P_{u_x}}{P_{u_y}}$. $AD$, $BC$ and $EF$ are parallel to the vector $P_y$. If $2d_y \leq P_{u_y} \cdot f_y$, the length of $BC$ is $d_y \sqrt{\frac{P_{u_x} \cdot y^2 + P_{u_x} \cdot x^2}{P_{u_y}}}$. as seen in Figure 8(a). On the other hand, if $2d_y > P_{u_y} \cdot f_y$, the length of $FC$ is $(P_{u_y} \cdot f_y - d_y) \frac{\sqrt{P_{u_x} \cdot y^2 + P_{u_x} \cdot x^2}}{P_{u_y}}$, as seen in Figure 8(b). The memory requirement can be decided by the parallelogram and the corresponding delay vector.

![Figure 8: the parallelogram decided by the delay vector](image)

**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 $d_x - d_y \frac{P_{u_x}}{P_{u_y}} \leq f_x$.

1. If $2d_y \leq P_{u_y} \cdot f_y$, then the memory requirement for storing intermediate partition data is equal to the number of integer points in the parallelogram $ABCD$ plus the number of integer points on the line $AH$ (see Figure 8(a)).

2. If $2d_y > P_{u_y} \cdot f_y$, 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 8(b)).
Proof: From Figure 8 (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 8(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 \(1\) 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 \(2\) 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 $f_x > d_x - d_y \frac{P_u x}{P_y}$, the memory requirement of the keep operation for the delay dependence will not change when $f_x$ decreases. When $f_x \leq d_x - d_y \frac{P_u x}{P_y}$, the memory requirement of the keep operation will decrease.

**Proof:**

If $f_x > d_x - d_y \frac{P_u x}{P_y}$, 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 $f_x \leq d_x - d_y \frac{P_u x}{P_y}$, let $m = \left\lfloor \frac{d_x - d_y \frac{P_u x}{P_y}}{f_x \times P_x} \right\rfloor$. The numbers of keep$_m(m - 1)$ 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$_m(m - 1)$ 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 keep$_m(m - 1)$ operations in R1 is $n_1 = \#BP \ast P_y y \ast (m \ast f_x - d_x + \frac{d_y}{P_y} P_y x)$, where $\#BP = f_y - \left\lfloor \frac{d_y}{P_y} \right\rfloor$ denotes the number of basic partitions in R1.

- The number of keep$_m(m - 1)$ operations in R2 is $n_2 = \sum_{i=0}^{\left\lfloor \frac{d_u y}{P_y} \right\rfloor - d_y - 1} \delta_i(a, b)$, where $a$ and $b$ satisfy the relation $a/b = m f_x - d_x + \frac{d_y}{P_y} P_y x$. 

22
• The number of keep-\(m\) operations in \(R1'\) is \(n_3 = \#BP \times P_y \times P_y, x - n_1\).

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

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

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

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

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

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

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

**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.
4.3 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 8** 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 \#pre be the number of prefetch operations, \#keep the number of keep operations, and \#iter the number of iterations in a partition. \(N_{mem}\) is the number of memory units and \(N_{ALU}\) the number of ALU units. \(T_{keep}\) and \(T_{pre}\) are the keep operation time and prefetch time, respectively, and \(L_{ALU}\) 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 \(N_{ALU} \leq N_{mem}\), \(T_{ALU} \geq T_{keep}\). Then

\[
\left\lfloor \frac{\#pre}{N_{mem}} \right\rfloor * T_{pre} + \left\lfloor \frac{\#keep}{N_{mem}} \right\rfloor * T_{keep} \leq L_{ALU} * \#iter + T_{keep}
\]  

(1)

**Proof:** In the memory part of the schedule, the length of the prefetch part is \(\left\lfloor \frac{\#pre}{N_{mem}} \right\rfloor * T_{pre}\), and the length of the keep part is \(\left\lfloor \frac{\#keep}{N_{mem}} \right\rfloor * T_{keep}\). The length of the ALU part of the schedule is \(L_{ALU} * \#iter\). 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 \(T_{keep}\) control steps to schedule those potential keep operations which correspond to the computational nodes in the last control step in the ALU part. Since \(N_{ALU} \leq N_{mem}\) and \(T_{ALU} \geq T_{keep}\), a legal memory schedule is guaranteed. Therefore, the length of the memory part of the schedule is at most \(T_{keep}\) control steps longer than that of the ALU part. \(\Box\)
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. \( f_x > \max[d_x - d_y \frac{P_{xy}}{P_{x,x}}] \) 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 \( T_{\text{keep}} \), we have a balanced schedule. On the other hand, if the memory part is more than \( T_{\text{keep}} \) 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. \( \square \)

### 4.4 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. \( \text{mem}_{\text{keep}} = \text{mem}_{\text{keepbase}} + \sum_{n=1}^{m} PL_n \times (f_y P_y \cdot y - dy) + (\# \text{interval} - m - 1) \times f_x \times (f_y P_y \cdot y - dy) \),

where \( \# \text{interval} \) represent the overall number of intervals.

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

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.

5 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 [4]. 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 [16], 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, $V_x = f_xP_x$ and $V_y = f_yP_y$. 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 con-
straints. 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition</th>
<th>m_2eq</th>
<th>len</th>
<th>list</th>
<th>base</th>
<th>m_2eq</th>
<th>len</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDF</td>
<td>(4,0)</td>
<td>4.06</td>
<td>10</td>
<td>(4,0)</td>
<td>4.06</td>
<td>0%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
<tr>
<td>IR</td>
<td>(6,0)</td>
<td>6.02</td>
<td>34</td>
<td>87.20%</td>
<td>6.03</td>
<td>0.2%</td>
<td>245</td>
<td>37</td>
</tr>
<tr>
<td>DPCM</td>
<td>(12,0)</td>
<td>4.01</td>
<td>23</td>
<td>83%</td>
<td>4.01</td>
<td>0%</td>
<td>455</td>
<td>37</td>
</tr>
<tr>
<td>2D</td>
<td>(3,0)</td>
<td>12.2</td>
<td>55</td>
<td>77.26%</td>
<td>12</td>
<td>0%</td>
<td>350</td>
<td>51</td>
</tr>
<tr>
<td>MDFG</td>
<td>(3,0)</td>
<td>4.01</td>
<td>40</td>
<td>89.88%</td>
<td>5.51</td>
<td>27.23%</td>
<td>453</td>
<td>32</td>
</tr>
<tr>
<td>Floyd</td>
<td>(4,0)</td>
<td>6</td>
<td>30</td>
<td>80%</td>
<td>6</td>
<td>0%</td>
<td>140</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 1: Experimental results without memory constraints assuming T_prefetch =10

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition size</th>
<th>m_2eq</th>
<th>len</th>
<th>list</th>
<th>base</th>
<th>m_2eq</th>
<th>len</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDF</td>
<td>(3,0)</td>
<td>6.04</td>
<td>40</td>
<td>(40)</td>
<td>7.08</td>
<td>14.69%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
<tr>
<td>IR</td>
<td>(3,0)</td>
<td>4.01</td>
<td>40</td>
<td>(40)</td>
<td>4.01</td>
<td>0%</td>
<td>455</td>
<td>37</td>
</tr>
<tr>
<td>DPCM</td>
<td>(3,0)</td>
<td>5.26</td>
<td>80</td>
<td>(80)</td>
<td>7.75</td>
<td>32.13%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
<tr>
<td>2D</td>
<td>(2,0)</td>
<td>6.04</td>
<td>40</td>
<td>(40)</td>
<td>6.04</td>
<td>14.69%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
<tr>
<td>MDFG</td>
<td>(2,0)</td>
<td>5.26</td>
<td>80</td>
<td>(80)</td>
<td>7.75</td>
<td>32.13%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
<tr>
<td>Floyd</td>
<td>(2,0)</td>
<td>6.04</td>
<td>40</td>
<td>(40)</td>
<td>6.04</td>
<td>14.69%</td>
<td>10</td>
<td>(4,0)</td>
</tr>
</tbody>
</table>

Table 2: Experimental results when reducing the available memory to 2/3 of the optimal

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Partition size</th>
<th>m_2eq</th>
<th>len</th>
<th>list</th>
<th>base</th>
<th>m_2eq</th>
<th>len</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDF</td>
<td>(1,0)</td>
<td>6.83</td>
<td>10</td>
<td>(4,0)</td>
<td>8.67</td>
<td>111</td>
<td>21.22%</td>
<td>10</td>
</tr>
<tr>
<td>IR</td>
<td>(2,0)</td>
<td>6.83</td>
<td>10</td>
<td>(4,0)</td>
<td>8.67</td>
<td>111</td>
<td>21.22%</td>
<td>10</td>
</tr>
<tr>
<td>DPCM</td>
<td>(4,0)</td>
<td>6.24</td>
<td>10</td>
<td>(4,0)</td>
<td>9.02</td>
<td>232</td>
<td>42.86%</td>
<td>10</td>
</tr>
<tr>
<td>2D</td>
<td>(1,0)</td>
<td>6.24</td>
<td>10</td>
<td>(4,0)</td>
<td>9.02</td>
<td>232</td>
<td>42.86%</td>
<td>10</td>
</tr>
<tr>
<td>MDFG</td>
<td>(1,0)</td>
<td>6.24</td>
<td>10</td>
<td>(4,0)</td>
<td>9.02</td>
<td>232</td>
<td>42.86%</td>
<td>10</td>
</tr>
<tr>
<td>Floyd</td>
<td>(1,0)</td>
<td>6.24</td>
<td>10</td>
<td>(4,0)</td>
<td>9.02</td>
<td>232</td>
<td>42.86%</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3: Experimental results when memory requirement is 1/2 of the original

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 paper 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. 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.

6 Conclusion

In this paper, 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.

References


