

Optimal Partitioning under Memory Constraints for Minimizing Average Schedule Length *

Zhong Wang Virgil Andronache Edwin H.-M. Sha
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
Email: {zwang1,vandrona,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 have been devised to hide that performance gap - from intermediate fast memories (caches) to various prefetching and memory management techniques to manipulate 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 cache 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 times of approximately 25% over existing methods.

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 the regular computation or data intensive applications.

Prefetching data into the cache nearest to the CPU can effectively hide the long memory latency. Our goal is that the prefetch operation well overlay as much as possible with the CPU operation, so that the CPU can always keep busy without waiting for the operands. In order to implement the prefetch, we need to find the best balance between the CPU operations and the prefetch operations, given the primary cache size constraint. A consequence of the cache constraint is that we can not prepare an arbitrary amount of data in the first cache before the program execution. This paper proposes techniques that prefetch information into the cache at run-time based on compile time information and partition iteration space, thereby, circumventing the cache size constraint problem.

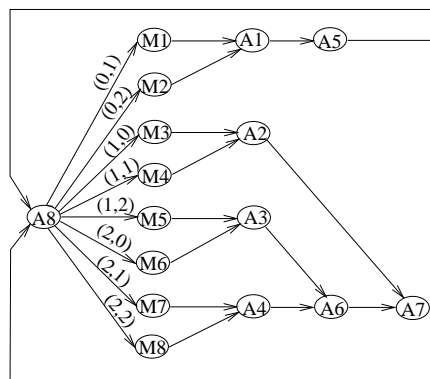
A lot of work has been done on hardware-based [4–6] and software-directed prefetching techniques [1, 2, 10–12]. Hardware-based prefetching, requiring some support unit connected to the cache, detects accesses with regular

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

```

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

```



(a) Fortran code representing an IIR filter

(b) equivalent MDFG representation

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

patterns and issues prefetches at run time. It is difficult for hardware-based prefetching technique to predict the memory references for complex access patterns. Software-directed approaches rely on compiler technology to insert explicit prefetch instructions. But the traditional software-directed approaches rarely have detailed consideration on how to schedule the loop body. The lack of detailed analysis usually results in an unbalance between ALU and memory schedule. Moreover, A number of real-time application specific systems such as image processing, DSP systems, etc, require such detailed analysis.

On the other hand, several multi-dimensional loop pipelining techniques have been proposed [7, 8] to minimize ALU time. For example, we can use multi-dimensional retiming techniques to explore more parallelism, to the extent of achieving full-parallelism for uniformed nested loop. These techniques, however did not consider the data fetching latency in their algorithms.

This paper will combine these two aspects, instruction level parallelism and reduce the memory access latency, to balance and minimize the overall schedule for uniform dependency loops. The 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 are combined to produce 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 in a high-level programming language code, as well as a typical DSP problem. The Fortran code derived from the IIR filter equation

$$y(n_1, n_2) = x(n_1, n_2) + \sum_{k_1=0}^2 \sum_{k_2=0}^2 c(k_1, k_2) * y(n_1 - k_1, n_2 - k_2) \quad \text{for } k_1, k_2 \neq 0$$

is shown in Figure 1(a). An equivalent Multi-dimensional Data Flow Graph is presented in Figure 1b. 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 unit and 4 memory unit 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 . Using the traditional list scheduling algorithm, the average schedule length will get to 23 CPU clock cycles when memory accessing 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 dominate 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 more better performance can be obtained, but still needs 7 CPU clock cycles of average schedule length. Therefore, our algorithm can make a large improvement.

The new algorithm exceeds the performance of existing algorithms by both optimizing both ALU schedule and memory schedule. the paper presents methods of deciding the best partition to achieve the 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 is a 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 this paper, we will be representing the code sequences to be optimized as MDFGs.

Definition 1 *A multi-dimensional data flow graph (MDFG) $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \in V \times V$ is the set of dependence edges, d is a function from E to Z^n , representing the multi-dimensional delay vector between two nodes, where n is the number of dimensions, and t is a function from V to the positive integers, representing the computation time of each node.*

The execution of all nodes in V one time is an *iteration*, i.e. the execution of one instance of the loop body. Iterations are identified by a vector i , equivalent to a multi-dimensional index, starting from $(0, 0, \dots, 0)$. An edge with delay $(0, 0, \dots, 0)$ represents an intra-iteration dependency, and an edge with non-zero delay $d(e)$ represents an inter-iteration dependency, it means the execution of the current iteration will use data computed $d(e)$ iterations before.

A different graph that is derived from an MDFG and which is useful in determining the optimal schedule is the CDG.

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.*

A *schedule vector* of a CDG can be regards as the normal vector for a set of parallel hyperplanes, of which the iterations in the same hyperplane will be executed in sequence, a schedule vector of $(1, 0)$ means the row-wise execution sequence. An MDFG is said to be realizable that 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, no execution sequence which can satisfy the delay dependence exists. To be realizable, an MDFG must satisfy that there exists a schedule vector s for the CDG with respect to G , such that the innerproduct $s \cdot d(e) \geq 0$ for any $e \in E$, and no cycles exist in the CDG.

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 more efficient by dividing the iteration space into regions that better exploit spatial locality called partitions.

The key to our memory management technique is the way the data produced in each partition is 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 of the partition and the 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° , 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 *A pair of partition vectors that satisfy the following constrains is legal. For each delay vector d_e in the CDG, the following cross product¹ relations hold. $d_e \times P_x \leq 0$ and $d_e \times P_y \geq 0$*

Proof: Since the execution sequence 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 cycle delay dependencies. □

Given a set of dependence edges d_1, d_2, \dots, 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*² the other is the right-most vector in relation to which all vectors are in the clock-wise region. It is obvious that the left-most vector and right-most vector satisfy the property 2.1, they are a pair of legal partition vectors.

Because nested loops should follow the lexicographical order, vector $s = (0, 1)$ is always a legal scheduling vector. Thus the positive x-axis is always the legal partition vector if we choose $(0, 1)$ as 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. Therefore the partition shape is decided by these two vectors.

2.3 Memory unit operation

According to the above subsection, the total iteration space will be divided into partitions, and the execution sequence is determined by the partition. 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, each kind of partition corresponds to different kind of memory unit operation.

Definition 3 *The next partition is the partition which is adjacent to the current partition and lie on the right of the current partition along the x-axis. The second next partition is adjacent to and lies on the right of next partition, the definition of third next partition, fourth next partition, etc are similar. The other partitions are those partitions except the partitions listed above.*

From the introduction of CDG in section 2, a delay dependency going from a partition to another partition means the execution of destination partition will use some data computed in the execution of start partition. Depending on which kind of partition the endpoint of delay dependency is located in, different memory unit operations will be used to treat them.

Definition 4 *To the delay dependency that go into the next m^{th} partition, **keep_m** memory operation is used to keep this data in the first level memory for m partitions. To the delay dependency that go into the other partition, **prefetch** memory operation is used to fetch data in advance.*

¹The *cross product* $p_1 \times p_2$ is defined as the signed area of the parallelogram formed by the points $(0,0)$, p_1 , p_2 , and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. It is $p_1 \times p_2 = p_1.xp_2.y - p_1.y p_2.x$.

²The counterclockwise region of a vector P is the region found by sweeping the plane in a counterwise direction, starting at P and ending when the sweeping line becomes vertical. The definition of clockwise region is similar.

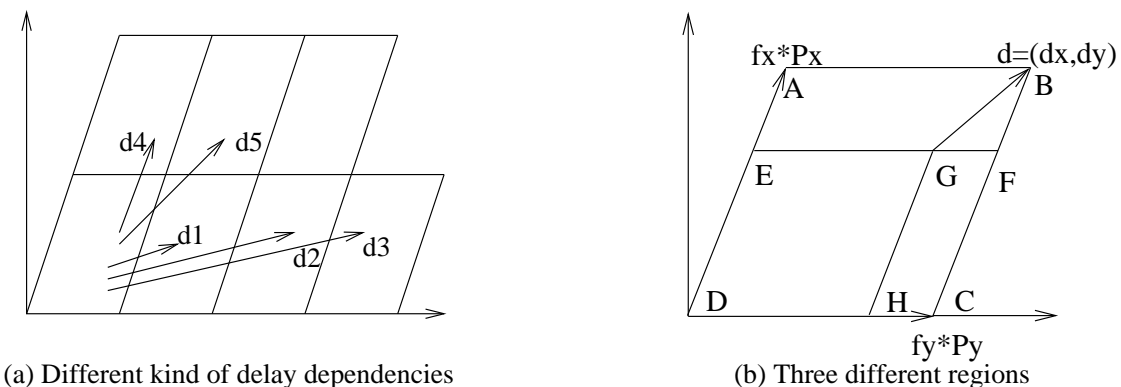


Figure 2: Different kinds of memory operations and its corresponding regions

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

Given a delay vector, a partition can be divided into three regions, shown in Figure 2(b). The region ABFE can be treated as prefetch area, in which the results will be used in other partitions, the region GFHC can be treated as keep area, in which the results will be used in next partition, while the region EGH D can be treated as inter area, in which the results will be used by some iterations in the same partition.

The reasons we have the above different kinds of memory unit operation is based on the two observations: 1). In the real loop, the delay dependency is not so long that the m of keep- m is too large. This imply the data kept in the first level memory must be used in the near future partition execution. 2). To fetch a data from the second level memory costs much more time than just to keep a data in the first level memory according to our memory arrangement.

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

Property 1 *Those delay dependencies with same starting node and different ending node can be classified into the following three classes:*

- a) *Ending at a node in the same partition, we can merge their memory unit operation.*
- b) *Ending at nodes in the different partitions and need keep operations, in which case we use their longest keep operation to represent all of them.*
- c) *Other situations except the above two, we can not merge memory unit operation corresponding to these delays.*

2.4 Architecture model

Our technique is designed for using in a system which has one or more processors with memory hierarchy, and this paper will focus on the memory management of the first level memory. We assume that as is the case with existing systems, the access time for the first level memory is significantly less than for the other memory levels. During a program's execution, if one instruction requires data which is not in the first level memory, the processor will have to fetch data from the other memory levels, which will cost much more time. Thus the use of a prefetch system that loads data in the first level memory before its explicit use can minimize the overall cost of accessing the other memory level. On the other hand, we can not prefetch arbitrary data into the first level memory, since it has limited size. Therefore, the goal of our algorithm is to overlap the memory access and program execution as much as possible, and satisfy the first level memory size constraint at the same time.

Our scheme is a software-based method, in which some special prefetch instruction are added to the code at compiler time by compiler. When the processor encounter these instructions during program execution, it will pass

them to the special hardware – *memory unit* to process it. *memory unit* is in charge of putting data in the first level memory before the execution of one partition need to reference them. Two types of prefetch instructions, *prefetch* and *keep*, are supported by memory units. The *prefetch* instruction prefetches the data from the second level memory to the first level memory; the *keep* instruction keeps the data in the first level memory for the use of later partition’s execution. Depending on the partition size and different delay dependencies, the data will need to be in the first level memory for different amount 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.

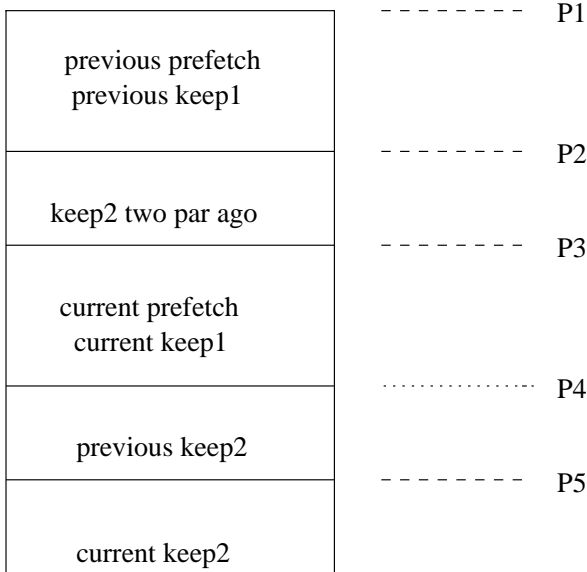


Figure 3: the example of memory arrangement

When arranging data in memory, we can allocate memory into several regions for different operation to put data in, Pointers to each of these regions will be kept in different circular lists – one each for the keep & 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. In the current partition execution, we can arrange data in memory as seen in figure 1. We have two circular lists: {p1,p3} and {p2,p4,p5}. When we go into the next partition execution, we only move forward the list element one step, thus get the list {p3, p1} and {p4, p5, p2}. We still obey the same rule to store different kinds of data.

2.5 Framework of the algorithm

In the algorithm, we divide the partition schedule into two parts: the ALU part and the memory part. In 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.

Algorithm

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

Output: A partition size with the optimal average schedule length.

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 constrain
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 exist")
stop
 6. For each delay vector $d = (d_x, d_y)$, calculate it's projection on the x-axis along P_y
direction, which is $ld = d_x - d_y \times \frac{P_y \cdot x}{P_y \cdot y}$,
 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 the right endpoint doesn't.
 8. Repeat increase f_x within this interval, until it reach the memory constraint.
output the partition size at this time.
-

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 execution, ready in the first level partition. The sequence of memory operation in memory part schedule from top to bottom is prefetch operation, keep_1 operation, keep_2 operation, etc.

3 Theoretic foundation

The main concern of this paper is with the division of iteration space into distinct partitions that can be effectively used in the execution of loop structures. In this section, we focus on calculating the number of elements that need to be kept in memory after the execution of each partition. 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 nonnegative 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 iteration space. Here assume AB is lower bound and AD is left boundary. Partitions are then defined by four parameters which describing its lower and left boundaries:

1. $P_x = (P_x \cdot x, P_x \cdot y)$ - a two dimensional vector that determines the orientation of the lower boundary of the parallelogram. In this approach its value is always (1,0).
2. $P_y = (P_y \cdot x, P_y \cdot y)$ - a two dimensional vector 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 **basic vectors** of the partition.
3. $f_x = \frac{|AD|}{|P_x|}$
4. $f_y = \frac{|AB|}{|P_y|}$

Given four parameters of a partition, a partition size and shape can be determined, and it can be divided into several basic partitions.

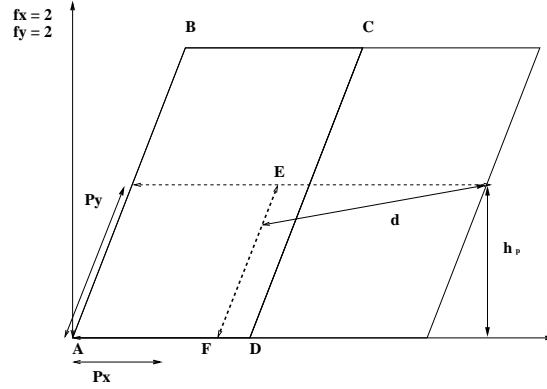


Figure 4: Two adjacent partitions

Definition 5 A basic partition is a parallelogram with the following properties:

1. A pair of its sides is parallel to the x-axis. Each of these sides has length $f_x * |P_x|$
2. A pair of its sides is parallel to P_y and each of these sides has length $|P_y|$

For ease of notation, let h_p denote the height of a basic partition. h_p is then equal to $P_y \cdot y$. Once the iteration space has been divided into such regions, the next step in the optimization process is the determination of a repetitive pattern that the inter-iteration dependency vectors follow in terms of these partitions. Each dependency vector $d = (d_x, d_y)$ is considered individually. Two cases can be distinguished.

Case 1: $d_x \leq f_x * P_x \cdot x$

Then, the only partitions that are involved in the memory allocation process are the current partition and next partition. Given a dependency vector (d_x, d_y) , it is obvious that iterations in the first d_x 'columns' of the next partition will incur incoming dependency vectors from the current partition. The exception to this are those nodes found in the first d_y rows. Thus, the total number of results that needs to be kept in memory for use in the next partition is $(f_y * h_p - d_y) * d_x$ when $f_y * h_p \geq d_y$ and 0, otherwise.

Case 2: $d_x > f_x * P_x \cdot x$

In this case, the process of determining the number of results that are kept in memory becomes a multi-step process, as follows:

1. Compute $np = \lceil \frac{d_y}{h_p} \rceil$.
2. Compute $m = \lceil \frac{d_x - d_y \frac{P_y \cdot x}{P_y \cdot y}}{f_x * P_x \cdot x} \rceil$. Let partition p' is then m^{th} partitions in the future
3. Find the coordinates of the upper left corner of partition p'
4. Find the node n , in the current partition that maps to the node computed in step 3 under the dependency considered. Once n is known, the location of the iterations whose results need to be kept in memory is determined. The results in all but the last np highest basic partitions need to be kept. The np -highest partition is subjected to an additional restriction regarding the iterations whose results should be kept in memory. These iterations are found below a horizontal line through the point n .
5. Calculate the total number of results that need to be kept in memory for use in the $(m-1)^{\text{th}}$ and m^{th} next partitions, according to the results derived further in this section.

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

Definition 6 An integral-area keep parallelogram is a parallelogram that satisfies the following properties:

1. A pair of its sides is parallel to the x-axis.
2. Its non-horizontal sides are either vertical (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, $w = t/m$, is a multiple of the inverse of the slope numerator, 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, $h = l * m$, and is a multiple of the slope

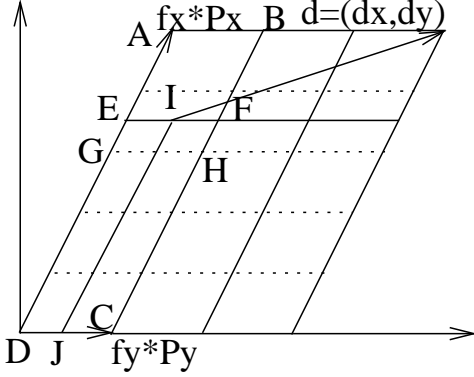


Figure 5: The division of a partition

In the above case 2, a partition can be divided into two regions ABEF and EFCD as seen in Figure 5. In the figure, the partition size is $f_x P_x \times f_y P_y$, the dotted line give the boundary of each basic partition. For a delay vector $d = (d_x, d_y)$, the nodes in region EFCD will be treated with keep operations. Based on the point n , this region consists of two sub-parallelogram, EIJD and IFCJ, each will map to different future partitions according to this delay vector. In Theorem 4 & 5, we determine how many nodes in these two sub-parallelograms, respectively. As a prerequisite for calculating the number of keep operations needed for each sub-parallelogram, we have the following lemma.

Lemma 2 *Let R be an integral-area keep parallelogram. The number of points with integer coordinates, I , in the region consisting of the closed region R with the exception of its right and upper boundaries is given by the formula:*

$$I = w * h$$

Proof: To an integral-area keep parallelogram with width 1 and height m , which has the integer coordinates at the left endpoint of the lower boundary. it is obvious the number of such points as defined in the lemma is m . This parallelogram can be divided into m sub-parallelograms each with width $\frac{1}{m}$. It can be proved that the left boundary of each sub-parallelogram passes through exact one integer point. Thus, the number of integer points in each sub-parallelogram is 1. Therefore, for any integral-area keep parallelogram with width $n/m, n < m, n \in \mathbb{N}$ and height m , the number of integer points is n .

To any integral-area keep parallelogram with width $w = t/m$ and height $h = l * m$, 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 $p = \lfloor t/m \rfloor, q = t \% m$. We can first divide this parallelogram into l parts, each with the same number of integer points. Then, for each sub-parallelogram with width $w = t/m$ and height $h = m$, divide it into two parts, one is the parallelogram with width $w = p * m$, the other with the width $w = q/m$. From the knowledge of the first paragraph in this proof, the number of integer points in the second part will be q . The number of integer points in the first part is $p * m$.

In conclusion, the overall number of points is $l * (p * m + q) = w * h$.

□

For ease of notation, we introduce the following definitions.

Definition 7 $\text{frac}(\mathbf{a})$ will be used to denote the fractional part of a real number a .

Definition 8 (\mathbf{a},\mathbf{b}) will be used to denote the interval from a to b on the real axis not including a and b . $[\mathbf{a},\mathbf{b}]$ will be used to denote the interval from a to b on the real axis inclusive of the points a and b .

Definition 9 Given a horizontal interval $[a,b]$ of width $\frac{m}{n}$, with m and n integers, $m \neq n$, define $\delta_i(m,n)$ as follows:

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

Lemma 3 Let R be an integral 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 $\sum_{i=0}^{q-1} \delta_i(a,b)$, where a,b are integers that satisfy the relation $a/b = \text{distance from } (p,q) \text{ to the left boundary of } R$.

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

It is clear that for any interval K , with $\text{frac}(\text{length}(K)) \neq 0$ there will be at least $\lceil \text{length}(K) \rceil - 1$ points with integer coordinates in the interval. At the same time, there are at most $\lceil \text{length}(K) \rceil$ points of 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 of 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 definition 9 of δ_i and with the above argument it becomes clear that the formula is correct. □

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 integers such that $m = \lceil \frac{d_x - d_y \frac{P_y \cdot x}{P_x \cdot x}}{f_x \cdot P_x \cdot x} \rceil$, the region of the partition from which results need to be kept in memory for use in the $(m - 1)^{\text{th}}$ 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: The first step in the proof is to determine the first point in the current partition which will map to the top right 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 $(m - 1)^{\text{th}}$ 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 $\text{height_basic} = \lfloor \frac{d_y}{P_y \cdot y} \rfloor + 1$. Then, the basic partition to consider within the target memory partition is the $\text{height_basic}^{\text{th}}$ basic partition from the base of the partition. Now, the right, top corner point of this basic partition, p is $p = (m * P_x, \text{height_basic} * P_y)$

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 $\text{height_basic} - 1$ basic partitions will have iterations that map into the future $(m - 1)^{\text{th}}$ or m^{th} next partition. Of these basic partitions, all but the last one will have the entire NR region map into the $(m - 1)^{\text{th}}$ 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. □

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 $m = \lceil \frac{d_x - d_y \frac{P_y \cdot x}{P_x \cdot x}}{f_x \cdot P_x \cdot x} \rceil$, 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.

Proof: The proof follows directly from the proof of theorem 4. □

4 Algorithms

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

4.1 ALU part scheduling

To the ALU part scheduling, the *multi-dimensional rotation scheduling algorithm* [8] is used to get a static compact schedule for one iteration. In this technique, we first get the initial schedule by *list scheduling*, and find the down-rotatable nodes set in which no node has a zero delay vector coming from any node not in this set. We then, at each rotation step, rotate down some nodes in this set and try to push them to their earliest control step according to the precedence constraints and resource availability. This can be implemented by selecting certain 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.

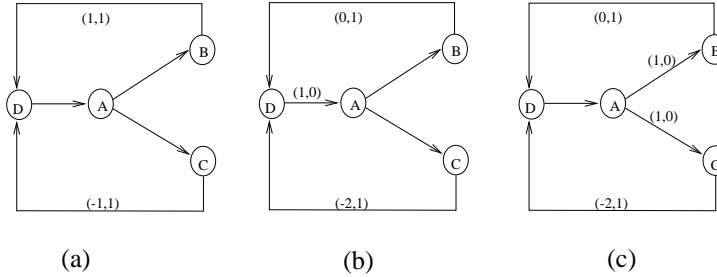


Figure 6: (a) Initial MDFG (b) After retiming $r(D) = (1,0)$ (c) After retiming $r(A) = (1,0)$

Control Step	ADD	MULTI
1	-	D
2	-	A
3	B	-
4	C	-

(a)

Control Step	ADD	MULTI
1	-	-
2	-	A
3	B	D
4	C	-

(b)

Control Step	ADD	MULTI
1	-	-
2	-	-
3	B	D
4	C	A

(c)

Figure 7: (a) Initial schedule (b) The schedule after rotated node D (c) The schedule after rotated node A

Consider the example in figure 6 (a). Let node A and D represent multiplication operations and node B and C represent addition operations. The initial schedule by 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), the node is 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), 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 length only two control steps.

After having the schedule of one iteration, we can simply duplicate this schedule for each node in the partition to get the ALU part schedule. Suppose the number of node in one partition is $\#nodes$ and the schedule length of one iteration is $len_{per-iteration}$. The ALU part schedule length will be $len_{per-iteration} \times \#nodes$, which is the least time of the program execution without the memory access interference, Therefore, this is the lower bound for the partition schedule. The goal of our algorithm is to make overall schedule as close to this lower bound as possible, while the first level memory space constraints are satisfied.

4.2 Memory part schedule

The memory unit prefetches data into the first level memory before it is needed, its operation is 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 part scheduling which is based on the scheduling per iteration, the memory part scheduling arranges the memory unit operations in one partition as a whole. Since prefetch operations do not depend on the current partition execution, we can arrange them from the beginning of the memory part scheduling. 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.

With partition defined as 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 two boundary 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 the partition size.

The memory requirement consists of three parts, the memory locations to store the intermedia 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 the later use.

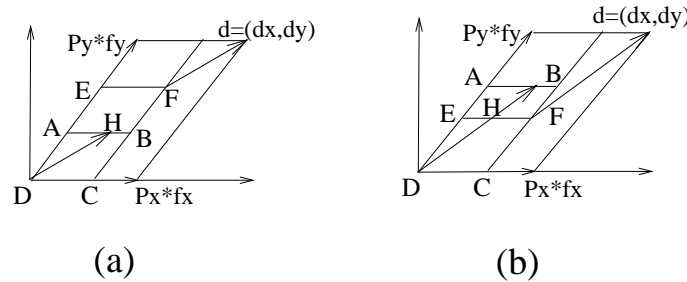


Figure 8: the parallelogram decided by the delay vector

To calculate this part of memory requirement, we can get a parallelogram ABCD shown in Figure 8(a) and EFCD shown in Figure 8(b). AB, CD and EF are parallel to the x-axis and the length is $f_x - dx + dy \frac{P_y \cdot x}{P_y \cdot y}$. AD, BC and EF are parallel to the vector P_y . In the case of $2d_y \leq P_y \cdot y * f_y$, the length of BC is $d_y \frac{P_y \cdot y}{\sqrt{P_y \cdot y^2 + P_y \cdot x^2}}$, as seen in Figure 8(a), In the case of $2d_y > P_y \cdot y * f_y$, the length of FC is $(P_y \cdot y * f_y - d_y) \frac{P_y \cdot y}{\sqrt{P_y \cdot y^2 + P_y \cdot x^2}}$, as seen in figure 8(b). The memory requirement can be decided by the parallelogram and the corresponding delay vector. When $d_y = 0$ is a special case that the height of ABCD is zero.

Lemma 6 Given a delay vector and a partition size $f_x P_x \times f_y P_y$, with the delay vector satisfying $d_x - d_y \frac{P_y \cdot x}{P_y \cdot y} \leq f_x$

and $2d_y \leq P_y \cdot y * f_y$, then the memory requirement to store intermedia internal-partition data is equal to the number of integer points in the parallelogram ABCD (see in Figure 8(a)) plus the number of integer points on the line AH.

Proof: From Figure 8.a, we can easily see that all the nodes except those in the parallelogram EFCD will need prefetch and keeping operations to satisfy this delay dependency. Their memory requirement will be considered in the memory requirement of prefetch and keep operation. Only those nodes in parallelogram EFCD have this delay vectors go to the 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 life time will not overlap with those nodes on and below the line AH. In conclusion, the memory requirement is the sum of the number of node in the parallelogram ABCD and the number of node on the line AH. \square

Similarly, when $2d_y > P_y \cdot y * f_y$, as seen in Figure 8(b), we can get 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 line DB and line EF).

If all the delays start from different nodes, the overall internal-partition memory requirement will be simple addition of all the memory requirement for each delay dependencies. Under the condition of multiple delay vectors starting from the same node, more consideration is needed. In this situation, the memory requirement still can be decided based on the parallelogram. However, the length of the two sides become the maximum value of all these parallelogram, and the point H becomes the point with the largest y coordinate (breaking ties by selecting the rightmost point) of all such points. Thus, the memory requirement can be calculated by the above lemma 6.

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 pre-loaded data for the current partition, the other is allocated for the new 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. By the same rule, the memory requirement for keep_3, keep_4, etc, obey this lemma. \square

Knowing the memory consumption for each kind of memory operation, we also need to know the number of each kind of memory operations 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 *In any partition of size $f_x P_x \times f_y P_y$, the number of prefetch operation is f_x times larger than the number of prefetch operation when partition size equal to $P_x \times P_y$.*

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 further in other partitions, which will be treated with prefetch operation. Therefore, the number of prefetch operations increases proportionally with the increasing of f_x . \square

Now the number of keep operations and the memory requirement of different kind of keep operations have been given in section 3 and by the above lemma 8. The next lemma investigates the change in memory requirement when f_x is decreased.

Lemma 9 *For any delay dependency, when $f_x > d_x - d_y \frac{P_y \cdot x}{P_y \cdot y}$, the memory requirement of the keep operation for this delay dependency will not change upon decreasing f_x . When $f_x \leq d_x$, the memory requirement of the keep operation will decrease.*

Proof:

If $f_x > d_x - d_y \frac{P_y \cdot x}{P_y \cdot y}$, this situation belongs to 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 $f_x \leq d_x - d_y \frac{P_{y,x}}{P_{y,y}}$, let $m = \lceil \frac{d_x - d_y \frac{P_{y,x}}{P_{y,y}}}{f_x \times P_{x,x}} \rceil$, the number of `keep_(m-1)` operations and the number of `keep_m` can be decided by the theorem 4 and theorem 5, respectively. Each of these two parts can be divided into two areas, R1 and R2. Assume the two areas of `keep_(m-1)` part are R1 and R2, while the two areas of `keep_m` part are R1' and R2'.

We can obtain the memory requirement for the all `keep` operations through the following steps.

- The number of `keep_(m-1)` operations in R1: $n_1 = \#BP * P_{y,y} * (m * f_x - d_x + \frac{d_y}{P_{y,y}} P_{y,x})$. $\#BP$ denotes how many basic partitions in the R1 area, it is $f_y - \lceil \frac{d_y}{P_{y,y}} \rceil$.
- The number of `keep_(m-1)` operations in R2: $n_2 = \sum_{i=0}^{\lceil \frac{d_y}{P_{y,y}} \rceil - d_y - 1} \delta_i(a, b)$, $a/b = m f_x - d_x + \frac{d_y}{P_{y,y}} P_{y,x}$
- The number of `keep_m` operations in the area R1': $n_3 = \#BP * P_{y,y} \times P_{y,x} - n_1$.
- The number of `keep_m` operations in the area R2': $n_4 = (\lceil \frac{d_y}{P_{y,y}} \rceil - d_y - 1) * f_x - n_2$
- According to the lemma 7, the overall memory requirement is $mem_{f_x} = (n_1 + n_2) * m + (n_3 + n_4) * (m + 1)$,

If we decrease the length of partition's lower boundary by one, the difference of memory requirement is $mem_{f_x} - mem_{f_x-1} = \#BP * f_y.y + P_{y,y} - d_y$ \square

From the above two lemma and lemma 6, we can know the change in memory requirement when f_x is decreasing.

Theorem 10 *Let $Size_{inter}$ represent the memory requirement for intermedia data in partition, $Size_{prefetch}$ represent the memory requirement for prefetch operation and $Size_{keep}$ denote the memory requirement for keep operation. When f_x is reduced,*

1. If $f_x > \max\{d_x - d_y \frac{P_{y,x}}{P_{y,y}}\}$, $Size_{inter}$ and $Size_{prefetch}$ will decrease, $Size_{keep}$ will not change.
2. If $f_x \leq \max\{d_x - d_y \frac{P_{y,x}}{P_{y,y}}\}$, all these three sizes will decrease.

Proof: It is obvious from the above lemmas. \square

Once the relation between memory requirement and the change of 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 large average schedule length compared with 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 reducing 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 than reduce f_y , because reducing f_x only interfere with the balanced schedule by small multiple of the number of `keep` operations.

4.3 Balanced schedule

The partition schedule consists of two parts, ALU schedule and memory schedule. 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 memory part schedule almost equal to the length of ALU part schedule. If we do not consider the first level memory constraint, we can always achieve this goal.

Definition 10 *A balanced schedule is a schedule for which the memory part schedule is at most one unit time of keep operation longer than the ALU part schedule.*

In the following theorem, we let be $\#pre$ the number of prefetch operations, $\#keep$ be the number of keep operations, and $\#iter$ be the number of iterations in a partition. N_{mem} is the number of memory units, N_{ALU} the number of ALU units. T_{keep} and T_{pre} 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 $N_{\text{ALU}} \leq N_{\text{mem}}$, $T_{\text{ALU}} \geq T_{\text{keep}}$. Then*

$$\left\lceil \frac{\#\text{pre}}{N_{\text{mem}}} \right\rceil * T_{\text{pre}} + \left\lceil \frac{\#\text{keep}}{N_{\text{mem}}} \right\rceil * T_{\text{keep}} \leq L_{\text{ALU}} * \#\text{iter} + T_{\text{keep}} \quad (1)$$

Proof: In the memory part of the schedule, the length of the prefetch part is $\left\lceil \frac{\#\text{pre}}{N_{\text{mem}}} \right\rceil * T_{\text{pre}}$, and the length of the keep part is $\left\lceil \frac{\#\text{keep}}{N_{\text{mem}}} \right\rceil * T_{\text{keep}}$. 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 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 $N_{\text{ALU}} \leq N_{\text{mem}}$ and $T_{\text{ALU}} \geq T_{\text{keep}}$, a legal memory part of the 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. \square

Once a balanced schedule is known, the following theorem proves that we can always reach this schedule by tentatively selecting the partition size, it is also the method of how to deciding the partition size to obtain a balanced schedule.

Theorem 12 *If the partition size satisfy the following conditions, there must exist some f_x and f_y that make the partition schedule a balanced schedule.*

1. $f_y * P_y \cdot y \geq d_y, \forall d = (d_x, d_y) \in D$
2. $f_x > \max\{d_x - d_y \frac{P_y \cdot y}{P_y \cdot x}\}$

Proof: These two conditions guarantee that there is not any delay vector spanning over more than two partitions. If a partition size satisfies the above two conditions, there are two cases: 1). The length difference between the memory and the ALU part is less than T_{keep} , which is a balanced schedule. 2). The memory part is more than T_{keep} longer than the ALU part. In this case, we can always enlarge f_y . Since condition 1 guarantees that the number of prefetch operations will not change with the increasing of f_y , and condition 2 guarantees that the number of keep operations is increasing at a slower rate than that of the number of iterations in partition. Combining 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 illustrate how to find a balance schedule. The memory requirement of this kind of balanced schedule may exceed the memory constraint. In this case, from theorem 10, we can satisfy the memory constraint by reducing the partition size.

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

To a partition with size $P_x \times f_y P_y$, we can easily calculate its memory requirement by using the knowledge of section 2.3 and section 4.2. Let $\text{mem}_{\text{keepbase}}$ and $\text{mem}_{\text{prebase}}$ the memory requirement for keep operations and prefetch operations under this size, respectively.

Given all the delay vectors after finishing the rotation for the ALU part, the projection of any delay vector on x-axis along the direction of P_y can be calculated. Sorting all these projections in increasing order. Then 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 *For the m^{th} interval, let the coordinate of its left endpoint and right endpoint be PL_m and PR_m , respectively, so $PR_m = PL_{(m+1)}$, 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:*

$$\begin{aligned} \text{Size}_{\text{mem-require}} &= \text{Size}_{\text{inter}} + f_x * \text{mem}_{\text{prebase}} + \text{mem}_{\text{keep}} \\ \text{mem}_{\text{keep}} &= \text{mem}_{\text{keepbase}} + \sum_{n=1}^m \text{PL}_n * (f_y \text{P}_y.y - dy) + (\#\text{interval} - m - 1) * f_x * (f_y \text{p}_y.y - dy), \end{aligned}$$

represent the overall number of intervals.

Proof: It can be gotten 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 can not 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 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 the main memory in the contemporary computer systems. We apply four different algorithms on these benchmarks: list scheduling, hardware prefetching scheme, a simple partition based algorithm and our algorithm. In the simple partition based algorithm, partitions are also used and the partition shape is the same as that in our algorithm, but the partition size is decided through a very simple method – each time f_x and f_y are increased by one in turn, until the memory constraints is reached. Then this size is the partition size used in the simple partition based algorithm schedule. In list scheduling, we use the same architecture model as that in our algorithm, but the ALU part uses the traditional list algorithm, and the memory is not partitioned. In hardware prefetching scheduling, we use the model presented in [9], in this model, whenever a block is accessed, the next block is also loaded.

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

In the first table, the first column list 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 list the two boundary partition vectors which decide the optimal partition size, $V_x = P_x \times f_x$ and $V_y = P_y \times f_y$. In the two algorithms that use partitioning, m_{req} represent 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 compared with other algorithms.

Benchmark	Partition		our		list		simple			hardware	
	Vx	Vy	m_req	len	len	ratio	len	ratio	m_req	len	ratio
WDF	(4,0)	(-12, 4)	116	4.06	16	74.62%	4.06	0%	116	10	59.4%
IIR	(6,0)	(-14, 7)	245	6.02	34	82.29%	6.03	0.2%	245	37	83.73%
DPCM	(12,0)	(-16, 8)	564	4.01	23	83%	4.01	0%	545	37	89.16%
2D	(3,0)	(0,4)	227	12	53	77.36%	12	0%	260	51	76.47%
MDFG	(3,0)	(0,23)	463	4.01	40	89.98%	5.51	27.23%	465	32	87.47%
Floyd	(4,0)	(-12,4)	149	6	30	80%	6	0%	149	30	80%

Table 1: Experimental results without memory constraints assuming $T_{prefetch} = 10$

Benchmark	Partition size		our		partition size		simple		
	Vx	Vy	m_req	len	Vx	Vy	len	m_req	ratio
WDF	(2,0)	(-12, 4)	82	4.5	(3,0)	(-9,3)	5.2	81	13.46%
IIR	(3,0)	(-18, 9)	181	6.04	(5,0)	(-10,5)	7.08	183	14.69%
DPCM	(10,0)	(-16, 8)	423	4.01	(9,0)	(-18,9)	4.01	427	0%
2D	(2,0)	(0,4)	179	12	(3,0)	(0,2)	19.33	176	37.92%
MDFG	(2,0)	(0,19)	347	5.26	(8,0)	(0,7)	7.75	354	32.13%
Floyd	(2,0)	(-12,4)	111	6	(3,0)	(-9,3)	6.56	111	8.54%

Table 2: Experimental results when reducing the available memory to $\frac{2}{3}$ of the optimal

Benchmark	Partition size		our		partition size		simple		
	Vx	Vy	m_req	len	Vx	Vy	len	m_req	ratio
WDF	(1,0)	(-12, 4)	50	5.25	(2,2)	(-6,2)	7.75	48	32.26%
IIR	(2,0)	(-12, 6)	117	6.83	(3,0)	(-8,4)	8.67	111	21.22%
DPCM	(4,0)	(-16, 8)	267	4.5	(6,0)	(-12,6)	5.23	265	13.96%
2D	(1,0)	(0,4)	113	12	(2,0)	(0,2)	20	128	40%
MDFG	(1,0)	(0,17)	232	6.24	(5,0)	(0,5)	10.92	232	42.86%
Floyd	(1,0)	(-9,3)	64	7.67	(2,0)	(-6,2)	10	59	23.3%

Table 3: Experimental results when memory requirement is $\frac{1}{2}$ of the original

In the two Table 2 and Table 3, we compared the effectiveness of the algorithms under memory constraints. To list schedule and hardware prefetching schedule, they have little memory requirement, so their length keep constant to benchmarks when the memory size reducing. But this constraint have a large influence on our algorithm and simple algorithm. So we compared these two algorithms' performance with the reduction of memory size. All items have the same meaning as the Table 1.

As we can see from these tables. list scheduling and hardware prefetching scheduling has much worse performance than other two algorithm. The reason is that in list scheduling, the schedule is dominated by a long memory part 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 same performance as list scheduling.

The simple algorithm can sometimes be competitive in performance with our algorithm in the case without the 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. Seen from the ratio item in the table, the performance gain can be significant in comparison with the other algorithms. To decide the partition shape and size, the computation is not complex. In our experiment, all the partition size can be decided in less than three seconds on a UltraSparc-30 platform. We have two part schedule in the architecture model, ALU part executes the computation, while the memory part prepares all the data for next ALU partition computation, which means the memory access and processor computation have been overlapped well, and add little overhead to ALU part. Comparing data in these tables, we can see the memory latency has been successfully hidden.

6 Conclusion

In this paper, an algorithm to get the minimal schedule length under memory constraints was proposed. This algorithm explores the ILP among instructions by using retiming techniques, while combined with data prefetch

to produce high throughput schedules. It works as follows, an ALU part and a memory part, are produced for the partition. Then, through the studies of the properties of different partition size under different memory constraints, the algorithm gives a partition size and shape so that an overall optimal average schedule length can be obtained. Experiment on DSP benchmarks showed that this algorithm can always produce optimal solutions as we had expected.

References

- [1] A.C.Klaiber and H.M.Levy. An architecture for software-controlled data prefetching. In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pages 43–53, 1991.
- [2] E.Gornish, E.Granston, and A.Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proc. 1990 Intl. Conf. on Supercomputing*, pages 354–368, 1990.
- [3] F.Chen, S.Tongsima, and E.H.-M.Sha. Loop scheduling optimization with data prefetching based on multi-dimensional retiming. In *Proc. ICSA 11th Intl. Conference on Parallel and Distributed Computing Systems*, pages 129–134, 1998.
- [4] F.Dahlgren and M.Dubois. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7), July 1995.
- [5] J.-L.Baer and T.-F.Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing'91*, pages 176–186, 1991.
- [6] J.W.C.Fu and J.H.Patel. Stride directed prefetching in scalar processors. In *Proc. of the 25th Intl. Symp. on Microarchitecture*, pages 102–110, December 1992.
- [7] N.Passos and E.H.-M.Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11), November 1996.
- [8] N.Passos and E.H.-M.Sha. Scheduling of uniform multi-dimensional systems under resource constraints. *Journal of IEEE Transactions on VLSI Systems*, 6(4), December 1998.
- [9] T.-F.Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, Dept. of Comp. Sci. and Engr, Univ. of Washington, 1993.
- [10] T.Mowry and A.Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2), June 1991.
- [11] T.Mowry, M.S.Lam, and A.Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.
- [12] W.Y.Chen, S.A.Mahlke, P.P.Chang, and W.M.Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, 1991.