Loop Scheduling and Partitions for Hiding Memory Latencies*

Fei Chen Edwin Hsing-Mean Sha
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
Email: {fchen,esha}@cse.nd.edu
Tel: (219)631-8803 Fax: (219)631-9260

Abstract

Partition Scheduling with Prefetching (PSP) is a memory latency hiding technique which combines the loop pipelining technique with data prefetching. In PSP, the iteration space is first divided into regular partitions. Then two parts of the schedule, the ALU part and the memory part, are produced and balanced to produce an overall schedule with high throughput. These two parts are executed simultaneously, and hence the remote memory latency are overlapped. We study the optimal partition shape and size so that a well balanced overall schedule can be obtained. Experiments on DSP benchmarks show that the proposed methodology consistently produces optimal or near optimal solutions.

1 Introduction

Because CPU speed has increased dramatically compared with memory speed, the slowness of memory hinders the overall system performance. A well planned data prefetching scheme may reduce the memory miss penalty by overlapping the processor computations with the memory access operations to achieve high throughput computation. Multi-dimensional (MD) problems are of particular interests. These problems, for example a large number of DSP applications, are characterized by nested loops with uniform date dependencies. *Loop pipelining* techniques are widely used to expose the instruction level parallelism so that a good schedule with high throughput can be obtained.

In this paper, we develop a methodology called *Partition Scheduling with Prefetching (PSP)* algorithm which combines the loop pipelining technique with a data prefetching approach. This technique can be used in computational intensive applications (especially multi-dimensional DSP applications) when two level memory hierarchies are existed. These two level memory are abstracted as the *local memory* and the *remote memory*. We assume it takes longer time to access the remote memory than it does for the local memory. We also assume a process contains multiple ALUs and multiple *memory units*. The ALUs are for doing the computations. The memory units are special hardwares we introduced for performing operations to prefetch data from the remote memory to the local memory.

The *partition (tiling)* technique is incorporated into the PSP algorithm. We partition the whole iteration space and execute one partition at a time. The benefit is that data locality is improved within the partition, and therefore the number of prefetching operations is reduced. We study the legal partition shape and provide formulas for determining an optimized partition size which guarantees a balanced overall schedule. Furthermore, we estimate the requirement of local memory size for executing this partition. The estimate gives designers a good indication of local memory requirement.

Traditional prefetching schemes [2, 5, 6] can be hardware or software based. They either only gave the dynamic prefetching decisions or did not give the complete static schedules. Partitioning the iteration space were not considered in those approaches either. Several multi-dimensional loop pipelining techniques have been proposed. For example, Passos and Sha proved that in multi-dimensional case (or nested loops), full-parallelism for MDFGs can always be achieved by using multi-dimensional retiming [4]. However, none of the above research efforts include the prefetching idea in their loop scheduling algorithms. Experiments are done in many DSP benchmarks and the results are compared with other scheduling algorithms, such as *list scheduling* algorithm and *PBS* algorithm [1]. Experiments show that the average length obtained by PSP is 26.7% of the one using list scheduling and 60.9% of the PBS. Since partitioning is not used in PBS, the result of our experiments also shows that partitioning the iteration space is very important for optimizing the overall schedule.

2 Algorithm framework

Modeling the ALU computation

*This work was partially supported by NSF MIP 95-01006, NSF MIP 97-04276.

A nested loop of computation can be represented by a *multi-dimensional data flow graph* (MDFG) [4]. An MDFG $G = (V, E, d, t)$ is a node- and 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 Z^n representing the multi-dimensional inter-iteration dependency (delay) between two nodes, where n is the number of dimensions, and t is a function from V to positive integers representing the computation time of each node.

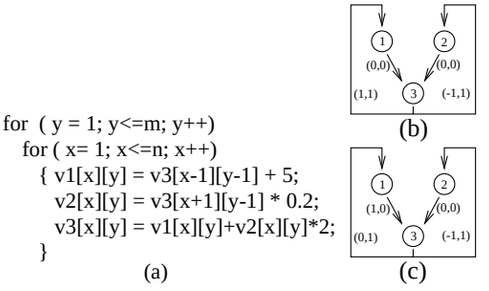

Figure 1: (a) a code of nested loops; (b) corresponding MDFG; (c) retimed MDFG.

The execution of all nodes in V one time represents an *iteration*. An iteration is identified by a vector $\hat{j}$, which is equivalent to a multi-dimensional index starting from $(0,0,\ldots,0)$. Inter-iteration dependencies are represented by vector-weighted edges. For any iteration $\hat{j}$, an edge e from u to v ($u \xrightarrow{e} v$) with delay vector $d(e)$ means that the computation of node v at iteration $\hat{j}$ depends on the execution of node u at iteration $\hat{j} - d(e)$. An edge with delay vector $(0,0,\ldots,0)$ represents a data dependency within the same iteration. Figure 1(a) is an example code of a nested loop, with the corresponding MDFG in Figure 1(b).

We call a legal MDFG $G = (V, E, d, t)$ is *realizable*, or *implementable*, if there exists a *schedule vector* s for the MDFG, such that $s \cdot d(e) \geq 0$ for any $e \in E$ [4]. This schedule vector s is regarded as the normal vector for a set of parallel equitemporal hyperplanes, of which the iterations in the same hyperplane will be executed in sequence. For example, in Figure 1(b), $s = (0,1)$.

Multi-dimensional retiming technique [4] is used in our algorithm. In our study, the legal retiming vector r is chosen as the base vector orthogonal to s. Using Figure 1(b) as an example, since $s = (0,1)$, the base retiming vector r can be $(1,0)$. The graph after MD retiming is shown in Figure 1(c).

Partitioning the iteration space

Instead of executing the whole iteration space in order by rows or columns, we first partition it and then execute the partitions one by one. The two boundaries of a partition are called *the partition vectors.* We will denote them by P_x and P_y. Due to the dependencies in the MDFG, partition vectors

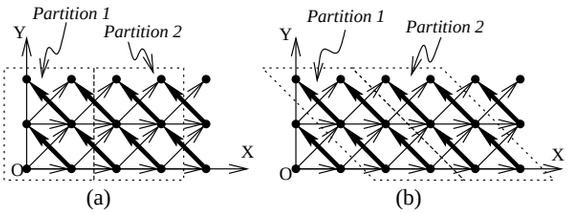

Figure 2: (a) An illegal partition of the iteration space; (b) A legal partition.

cannot be arbitrarily chosen. For example, note the iteration space in Figure 2(a), where dots represent iterations and vectors represent the inter-iteration dependencies. If we partition the iteration space to rectangular shape, as shown in Figure 2(a), this partition method is illegal, because of the forward dependencies from $\mathsf{Partition}_i$ to $\mathsf{Partition}_{i+1}$ (the thin vectors) and the backward dependencies from $\mathsf{Partition}_{i+1}$ to $\mathsf{Partition}_i$ (the bold vectors). Due to these two-way dependencies between partitions, we cannot execute either one first. This partition is therefore not implementable and is *illegal.* In contrast, consider the alternative partition method shown in Figure 2(b). Since there are no two-way dependencies, a feasible partition execution sequence exists. For example, execute $\mathsf{Partition}_1$ first, then $\mathsf{Partition}_2$, and so on. Therefore, it is a legal partition.

Architecture model

We assume a processor contains multiple ALUs and multiple special hardwares called memory units. Associated with the processor is a small local memory. Accessing the local memory is fast. There is also a large remote memory. However, accessing it is slow. The goal of our technique is to prefetch the operands of all computations into the local memory before the actual computations are taken place. These prefetching operations are performed by memory units. Two types of prefetching instructions, `prefetch` and `keep`, are supported by memory units. The `prefetch` instruction prefetches the data from the remote memory to the local memory; the `keep` instruction keeps the data in the local memory for the execution of the next partition. Both of them are issued to make sure those data about to be referenced will be appeared in the local memory.

PSP algorithm framework

The schedule generated by PSP consists of two parts: the ALU part and the memory parts. In the ALU part of the schedule, we first use the *multi-dimensional rotation scheduling algorithm* [3] to create the ALU schedule for one iteration. We then duplicate this one iteration ALU schedule and append the copies consecutively to form the n iteration ALU schedule (where n is the number of iterations in the partition). The memory part of the schedule will be executed at the same time as the ALU part. It gives the global schedule for memory operations to prefetch all the operands needed by

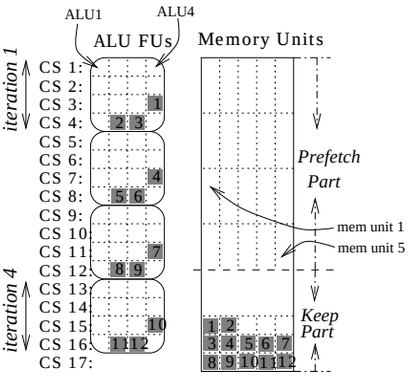

Figure 3: The overall schedule of a partition. Assuming there are four ALU functional units and five memory units.

the next partition into the local memory.

We call the partition which is being executed the ***current partition,*** and call the one that will be executed next the ***next partition***. For all other partitions which have not been executed except the next one, we call them ***other partitions*** (see Figure 4(c)). In the memory part scheduling, if a non-zero delay edge passes from the current partition into other partitions, a `prefetch` operation is needed. Each directed edge from the current partition to the next partition corresponds to a `keep` operation. The framework of our algorithm is illustrated in **Algorithm PSP**.

Figure 3 gives an example of our *overall schedule*—the ALU part as well as the memory part. There are four iterations in one partition. In the ALU part, each iteration takes 4 control steps (CS) to finish, and hence all four iterations take $len(ALU) = 16$ control steps. In the memory part, all `prefetch` operations are scheduled from the top, and then the `keep` operations. The length of the memory part of the schedule is $len(mem) = 17$. Since these parts are executed simultaneously, the overall schedule length is the maximum of them, which is $len(overall) = \max\{len(ALU), len(mem)\} = 17$. If we divide the overall schedule length by the number of iterations in the partition, we get the average schedule length $ave\_len(overall) = \frac{17}{4} = 4.25$.

3 PSP scheduling

We use multi-dimensional rotation scheduling algorithm to schedule the ALU part for each iteration. *Multi-dimensional rotation scheduling* is a loop pipelining technique which implicitly uses the multi-dimensional retiming heuristic for scheduling cyclic graphs. The rotation scheduling is described in detail in [3]. Given an initial schedule, the rotation technique repeatedly transforms the ALU part of the schedule to a more compact one under the resource constraint. Consider an example of MD rotation scheduling performed on the

Algorithm PSP Partition Scheduling with Prefetching

Input: Initial MDFG; ALU and memory constraint; execution time for ALU and memory operations.

Output: An optimal partition and the optimized ALU and memory schedules for executing the partition.

```
/* using list scheduling to obtain ALU schedule */
S ← initial ALU part schedule for one iteration
repeat
    /* reducing the length of one iteration ALU part of the schedule by using MD rotation scheduling */
    S ← rotate the current schedule S
    G_r ← retimed MDFG
    /* decide the optimal partition shape and size */
    Obtain the legal partition directions P_x0, P_y0 according to G_r
    Obtain the partition size so that the balancing property (Theorem 3) is satisfied
    /* produce the overall schedule */
    Number the iterations
    Entire ALU part scheduling
    Memory part scheduling
    /* evaluation */
    Calculate the average length of the overall schedule
    Calculate the local memory requirement
until the average schedule length cannot be reduced
```

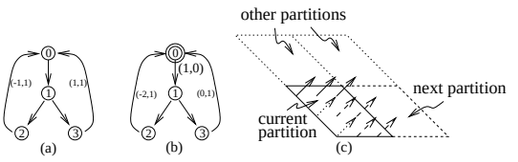

Figure 4: (a) The original MDFG of Wave Digital Filter; (b) The retimed MDFG after rotating node 0; (c) Solid edges represent `prefetch` operations; dashed edges represent `keep` operations; dot edges are the data dependencies inside the partition, hence no memory operation is needed.

Wave Digital Filter shown in Table 1, with the corresponding MDFG in Figure 4. Notation $n^{(i)}$ in the table conveys the computation node n in the original i-th iteration in the partition. According to the data dependencies in the original MDFG shown in Figure 4(a), we have an initial schedule with length three, shown in the left part of Table 1. During the rotation, computation node 0 in control step (CS) 1 is rotated, and the corresponding node 0 in the MDFG is retimed by the base retiming vector $r = (1,0)$. The schedule length is reduced to 2 after the rotation. In PSP scheduling algorithm, the ALU part then applies the same schedule pattern for each iteration in the partition. Iterations are executed one after the other in the ALU part of the schedule.

Scheduling of the memory part consists of several steps. First, given the retimed MDFG as a result from the MD rotation, we need to decide the directions of the legal partition vectors. Second, the iterations in the partition should be numbered so that they can be scheduled in that order. Third and

CS	Initial schedule		Schedule after rotation	
	ALU1	*ALU2*	*ALU1*	*ALU2*
1	$0^{(0)}$		$1^{(0)}$	$0^{(1)}$
2	$1^{(0)}$		$2^{(0)}$	$3^{(0)}$
3	$2^{(0)}$	$3^{(0)}$		

Table 1: The ALU part of the schedule.

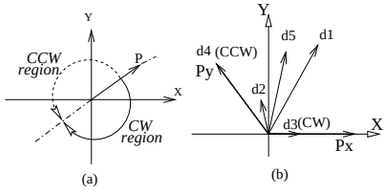

Figure 5: (a) The CW and CCW regions relative to vector p; (b) The extreme CW and CCW vectors of vectors $d_1, d_2, \ldots,$ and d_k and the partition vector P_x and P_y.

the most important step, calculate the optimal partition size to ensure a balanced schedule. Fourth and the last, actually create the memory part of the schedule. We will explain these steps below in great detail.

Among all the delay vectors in an MDFG, two extreme vectors, clockwise (CW) and counterclockwise (CCW), are the most important for deciding the directions of the legal partition vectors. They are given by the following definition.

Definition 1 *The extreme (outermost) clockwise vector CW of a vector set $D = \{d_1, d_2, \ldots, d_k\}$ satisfies these two conditions:* (1) *$CW \in D$;* (2) *all the vectors in $D - \{CW\}$ are in counterclockwise region of CW. The definition of CCW vector is similar.*

Figure 5(a) illustrates the *clockwise and counterclockwise regions* relative to a vector p. The magnitude of the *cross product* of two vectors p_1 and p_2, denoted by $p_1 \otimes p_2$, is used to determine the relative position of p_1 and p_2. If $p_1 = (p_1.x, p_1.y)$ and $p_2 = (p_2.x, p_2.y)$, then $p_1 \otimes p_2 = p_1.x \times p_2.y - p_2.x \times p_1.y$. If $p_1 \otimes p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin $(0,0)$; if this cross product is negative, then p_1 is counterclockwise from p_2.

Legal partition vectors can only be outside of CW and CCW or aligned with them. For example in Figure 5(b), we choose P_y to be aligned with CCW, and P_x to the aligned with x-axis, which is outside of CW. This is a legal choice of partition vectors. In PSP algorithm, we assume the y elements of the delay vectors of the input MDFG are always ≥ 0, which is often the case in real applications with nested loops. Therefore, vector $s = (0,1)$ is always the legal scheduling vector. After choosing the base retiming vector r as $(1,0)$, the positive x-axis is always a legal direction for the partition vector. In our algorithm, the direction of the counterclockwise partition vector, P_y, is chosen to be aligned with the vector CCW;

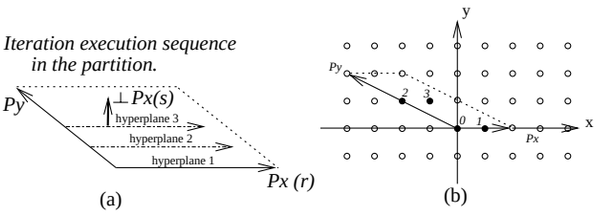

Figure 6: (a)Iterations will be executed from left to right in the P_x direction and then precede to the next hyperplane along the direction perpendicular to P_x; (b) Iteration orders in the partition.

while the direction of the clockwise extreme partition vector, P_x, is aligned with the positive x-axis. For convenience, we use P_{x0} and P_{y0} to denote the *base partition vectors* showing these two directions (The elements in the base partition vector have no common divisors). The actual partition vectors are then denoted by $P_x = f_x P_{x0}$ and $P_y = f_y P_{y0}$, where f_x and f_y are called *partition factors*, which is related to the size of the partition.

The next step is to number the iterations within the partition so that they can be scheduled in that order. The iterations are numbered from left to right in the P_x direction, as illustrated in Figure 6(a), and then to the next hyperplane along with the direction of the vector perpendicular to P_x. Figure 6(b) shows an example of the iteration order. The black dots represent the iterations in the partition, while the numbers give the order. The numbering can be easily done by sorting the iteration indices of different iterations—whoever has the smaller y element or has the same y element but the smaller x element will get the smaller number. We will discuss how to obtain the optimal partition size in Section 4.

CS	ALU part		memory part	
	ALU1	*ALU2*	*MEM1*	*MEM2*
1	$1^{(0)}$	$0^{(1)}$	$P2^{(2)}(-2,1)$	$P2^{(3)}(-2,1)$
2	$2^{(0)}$	$3^{(0)}$	$\downarrow$	$\downarrow$
3	$1^{(1)}$	$0^{(2)}$	$P3^{(2)}(0,1)$	$P3^{(3)}(0,1)$
4	$2^{(1)}$	$3^{(1)}$	$\downarrow$	$\downarrow$
5	$1^{(2)}$	$0^{(3)}$	$K0^{(2)}(1,0)$	$K3^{(0)}(0,1)$
6	$2^{(2)}$	$3^{(2)}$	$K3^{(1)}(0,1)$	
7	$1^{(3)}$	$0^{(4)}$		
8	$2^{(3)}$	$3^{(3)}$	$K0^{(4)}(1,0)$	

Table 2: The overall schedule with respect to the MDFG of Wave Digital Filter in Figure 4.

After obtaining the partition directions and size, we can start to schedule the memory part. `Prefetch` operations are scheduled as early as possible, because they do not have any data dependencies. `Keep` operations have the data dependencies from the ALU part. Therefore, a `keep` operation must be scheduled after the corresponding computation, whichever provides the result of that data instance, is finished. For each `keep`, we define the earliest starting time (`ES`) as the con-

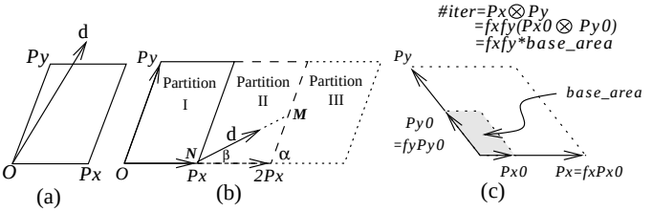

Figure 7: (a) f_y restriction; (b) f_x restriction; (c) The number of iterations (#iter) inside one partition.

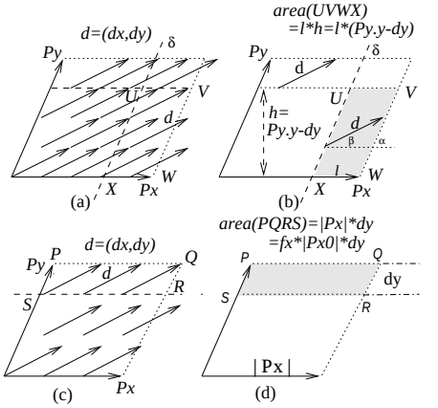

Figure 8: (a)(b) Calculating the number of the delay edges crossing the boundary of the current partition and entering the next partition; (c)(d) Calculating the number of the delay edges crossing the boundary of the current partition and entering other partitions.

trol step when the corresponding value is finished computing. Then, starting from ES, we schedule the keep operation at the earliest available place in the memory part.

Table 2 is the overall schedule of Wave Digital Filter shown in Figure 4. Here we assume two ALUs and two memory units. The ALU part of the schedule is a duplication of the four iterations of the schedule shown in Table 1. In the memory part, the notation "$Pn^{(i)}(x,y)$" conveys "prefetch the data instance which corresponds to the delay vector (x,y) from node n in the i-th iteration". For example, "$P2^{(3)}(-2,1)$" means "prefetch the data corresponding to the delay vector $(-2,1)$ from node $2^{(3)}$". We assume in Table 2 each prefetch operation takes two time units, that is, $T_{pre}=2$. The down arrows (↓) in the table represent the continuation of the prefetch operation. Similarly, "$Kn^{(i)}(x,y)$" denotes the keep operation. We assume each keep operation takes one time unit, i.e., $T_{keep}=1$. In this example, the length of the overall schedule "L" is 8. Since there are 4 iterations in the partition, the average length of the overall length, denoted by L_{ave}, is $\frac{L}{4}=2$, which is equal to the lower bound.

4 Partition size and memory size

In the previous section, we have decided the partition directions, denoted by P_{x0} and P_{y0}. Here we will determine the two partition factors f_x and f_y, so that a balanced schedule can be achieved.

First, we impose the restriction to f_y that it should be large enough so that no delays can pass through the entire partition along the direction of P_y. For example, the partition vector P_y in Figure 7(a) is not large enough, because the delay vector d crosses both the bottom and the top boundaries of the partition. Denoting the set of all the non-zero delay vectors in the MDFG as D, the above restriction can be represented by inequality: $f_y \times P_{y0}.y \geq dy, \quad \forall d=(dx,dy) \in D$.

Partition vector P_x is restricted so that no delays starting from the current partition can reach two partitions later. In other words, in Figure 7(b), delay edges starting from Partition I cannot reach Partition III. Therefore, we have $|d| < |\overline{NM}| = |P_x|\frac{\sin\alpha}{\sin(\alpha-\beta)} = f_x|P_{x0}|\frac{\sin\alpha}{\sin(\alpha-\beta)}$. This gives us: $f_x > \frac{|d|\sin(\alpha-\beta)}{|P_{x0}|\sin\alpha}, \quad \forall d=(dx,dy) \in D$. Since f_x is an integer, this inequality is equivalent to $fx \geq \lfloor \frac{|d|\sin(\alpha-\beta)}{|P_{x0}|\sin\alpha} \rfloor + 1, \quad \forall d=(dx,dy) \in D$.

Below we derive the conditions for a balanced schedule. Lemma 1 shows how to calculate the length of the ALU part of the schedule, referring to Figure 7(c).

Lemma 1 *The length of the ALU part of the schedule is $L_{ALU} \times \#iter = L_{ALU} f_x f_y (P_{x0} \otimes P_{y0})$, where L_{ALU} denotes the length of the one-iteration ALU part of the schedule, and #iter is the number of iterations in the partition.*

Then we estimate how many memory operations are needed by calculating the areas of two shaded regions in Figure 8. Given a delay vector $d=(dx,dy)$, region $UVWX$ in the current partition, shown in Figure 8, is the region where d will enter the next partition. Similarly, region $PQRS$ is where d will enter other partitions. We denote the areas of the above two regions as $A_{goto\_next}(d)$ and $A_{goto\_others}(d)$, respectively, with respect to a given delay vector $d=(dx,dy)$.

Lemma 2 *Given a delay vector $d=(dx,dy)$, $A_{goto\_next}(d) = (f_y P_{y0}.y - dy)\frac{\sin(\alpha-\beta)}{\sin\alpha}|d|$, and $A_{goto\_others}(d) = f_x dy |P_{x0}|$.*

Note that the number of delay edges entering the next partition, i.e. keep operations, is very close to the area of $UVWX$. Summing up all these areas for every distinct d, we get the total number of keep operations, $\#\text{keep} = \sum_d A_{goto\_next}(d) = \sum_d (f_y P_{y0}.y - dy)\frac{\sin(\alpha-\beta)}{\sin\alpha}|d|$, for all $d=(dx,dy)$. Similarly, the total number of prefetch operations is $\#\text{prefetch} = \sum_d A_{goto\_others}(d) = \sum_d area(PQRS) = \sum_d |P_x|dy = f_x \sum_d dy |P_{x0}|$.

Theorem 3 gives the conditions of what we call as a *balanced* schedule. The idea here is to schedule prefetch

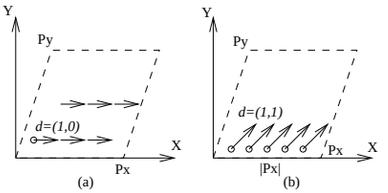

Figure 9: (a) One memory location is needed for delay $d = (1,0)$; (b) $|P_x|d_y$ memory locations are needed for delay $d = (d_x, d_y)$, when $d_y \neq 0$.

operations from the top of the memory part of the schedule, and schedule the `keep` from the bottom. The left-hand side of Inequality 1 is the estimated length of the memory part schedule, and we only allow it to be at most T_{keep} control steps longer than the ALU part, as shown in the right-hand side. The reason of leaving out T_{keep} steps is to make rooms for those potential `keep` operations corresponding to the computational nodes at the last control step in the ALU part. Corollary 4 concerns about the average overall schedule length.

Theorem 3 *Assume that $N_{ALU} \leq N_{mem}$, $T_{ALU} \geq T_{keep}$, and Inequality 1 is satisfied.*

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

The length of the memory part of the schedule is at most T_{keep} control steps longer than that of the ALU part.

Corollary 4 *If the partition satisfies the conditions presented in Theorem 3, the average length of the overall schedule is at most $\frac{T_{keep}}{\#iter}$ plus the average length of the ALU part of the schedule.*

Experiments show that rotation scheduling in most cases can generates the ALU part of the schedule which achieves the lower bound, i.e., $L_{ALU} = bound(ALU)$. Therefore, the overall schedule either reaches its lower bound or is very close to it; the difference is at most $\frac{T_{keep}}{P_x \otimes P_y}$.

Now we estimate the local memory size for executing the partition. We classify the memory usage into two categories: basic memory for the working set and reserved memory for `prefetch` and `keep` operations.

The former corresponds to all the internal delay edges in the partition. The delay edge $d = (1,0)$ in Figure 9(a) indicates a data instance produced in Iteration I_0 and consumed in the next Iteration I_1. Only one memory location is needed to hold this data because we can reuse the same location for later iterations. In general, we need d_x memory locations for each $d = (d_x, 0)$. However, when $d = (d_x, 1)$, as shown in Figure 9(b), a whole row of intermediate values need to be kept. Thus a total of $|P_x| \times 1$ memory locations are needed. In general, for each $d = (d_x, d_y)$ where $dy \neq 0$, $|P_x|d_y$ memory locations are needed. Summarizing the above, the size of the basic memory for the working set is equal to

$$Size_{ws} = \sum_{\forall d=(d_x,d_y)} \begin{cases} d_x & \text{,when} \quad d_y = 0 \\ |P_x|d_y & \text{,when} \quad d_y \neq 0 \end{cases}$$

Now let us consider the second category: reserved memory for `prefetch` and `keep` operations. These operations represent the data instances *pre-loaded* or *pre-occupied* in the local memory before we execute this partition. Each one of them needs a reserved memory location. The total number of these pre-occupied data is two times the total number of memory operations (one for the pre-loaded data for the current partition; the other for the new generated data for the next partition). Therefore, the size of this part of the memory is $Size_{reserved} = 2(\#pre + \#keep)$. Finally, the local memory needed to execute this partition is $Local\_size = Size_{ws} + Size_{reserved}$.

5 Experimental Results

In this section, the effectiveness of the PSP algorithm is evaluated by running a set of simulations on DSP benchmarks. Table 3 and Table 4 show our scheduling results. The first column presents the benchmarks' names. The second to fourth columns are the parameters of the input MDFG, with the second column showing the number of nodes and the third and fourth columns showing the ALU and memory unit resource constraints. The partition generated by the algorithm is shown in the fifth to seventh columns. The final schedule is shown in the next three columns. Column "L" gives the length of the overall schedule and Column "L_{ave}" is the average ($\frac{L}{\#iter}$). In order to compare our results with the lower bound, as well as the results from other algorithms, we calculated the lower bounds of the schedule length , $\lceil \frac{N}{N_{alu}} \rceil$, and put them in Column "*LB*". We also ran the same set of benchmarks using *list scheduling* and *Prefetch Balanced rotation Scheduling (PBS)*. The results are shown in Columns "List' and "PBS", respectively, where the sub-column "len" is the schedule length and the sub-column "ratio" is the ratio comparing the PSP schedule length with that of list scheduling and PBS scheduling, i.e. ratio= $\frac{L_{ave}}{len}$.

The abbreviations for our benchmarks "WDF", "IIR", "DPCM", "2D" and "Floyd" stand for *Wave Digital filter*, *Infinite Impulse Response filter*, *Differential Pulse-Code Modulation device*, *Two Dimensional filter*, and *Floyd-Steinberg algorithm*, respectively. In Table 3, we assume that each ALU operation takes 1 time unit, each `keep` operation also takes 1 time unit, and each `prefetch` takes 2 time units, while in Table 4, we assume each `prefetch` takes 10 time units. In the PBS experiments in Table 4, the graphs are first

Benchmark	Parameters			Partition			PSP Schedule			List		PBS	
	N	N_{alu}	N_{mem}	P_x	P_y	*#iter*	L	L_{ave}	LB	len	ratio	len	ratio
WDF(1)	4	2	2	(3,0)	(-4,2)	6	12	**2**	2	*3*	66.7%	*3*	66.7%
WDF(2)	12	3	3	(4,0)	(-3,1)	4	17	**4.25**	4	*6*	70.8%	*4*	106.3%
IIR	16	3	3	(6,0)	(-4,2)	12	73	**6.08**	6	*8*	76%	*6*	101.3%
DPCM	16	4	4	(6,0)	(-4,2)	12	49	**4.08**	4	*7*	58.3%	*7*	58.3%
2D(1)	34	3	3	(3,0)	(0,1)	3	36	**12**	12	*16*	75%	*12*	100%
2D(2)	4	2	2	(2,0)	(-4,2)	4	9	**2.25**	2	*4*	56.3%	*3*	75%
MDFG1	8	2	2	(4,0)	(-3,1)	4	17	**4.25**	4	*7*	60.7%	*4*	106.3%
MDFG2	8	2	2	(4,0)	(-6,6)	24	97	**4.04**	4	*8*	50.5%	*8*	50.5%
Floyd	16	3	3	(4,0)	(-6,2)	8	48	**6**	6	*11*	54.5%	*6*	100%

Table 3: Experimental results on DSP filter benchmarks assuming $T_{prefetch} = 2$.

Benchmark	Parameters			Partition			PSP Schedule			List		$PBS_{\text{unfold by } 2\times 2}$	
	N	N_{alu}	N_{mem}	P_x	P_y	*#iter*	L	L_{ave}	LB	len	ratio	len	ratio
WDF(1)	4	2	2	(3,0)	(-14,7)	21	42	**2**	2	*10*	20%	*5.25*	38.1%
WDF(2)	12	3	3	(4,0)	(-12,4)	16	65	**4.06**	4	*10*	40.6%	*5*	81.2%
IIR	16	3	3	(6,0)	(-14,7)	42	253	**6.02**	6	*21*	28.7%	*6*	100.3 %
DPCM	16	4	4	(6,0)	(-14,7)	42	169	**4.02**	4	*20*	20.1%	*5.5*	73.1%
2D(1)	34	3	3	(3,0)	(0,4)	12	144	**12**	12	*40*	30%	*20*	60%
2D(2)	4	2	2	(2,0)	(-16,8)	16	33	**2.06**	2	*10*	20.6%	*5*	41.2%
MDFG1	8	2	2	(4,0)	(-12,4)	16	65	**4.06**	4	*10*	40.6%	*5.25*	77.3 %
MDFG2	8	2	2	(4,0)	(-35,35)	140	561	**4.01**	4	*40*	10.0%	*23.75*	16.9%
Floyd	16	3	3	(4,0)	(-12,4)	16	96	**6**	6	*20*	30%	*10*	60%

Table 4: Experimental results on DSP filter benchmarks assuming $T_{prefetch} = 10$.

unfolded by a factor of 2×2 before performing PBS scheduling.

As we can see, list scheduling rarely achieves the optimal schedule length; the schedules are often dominated by a long memory part. In order words, the list schedules are not well balanced. Although PBS is better than list scheduling, it too becomes less effective to generate a balanced schedule especially when $T_{prefetch}$ is large. Moreover, PBS needs to explicitly unfold by large factors in order to generate good schedules. This may cause a lot of computations (For example, after unfolded by a factor of 2×2, the total number of nodes is 4 times that of the original).

The PSP algorithm consistently produces optimal or near optimal schedules, as shown by the bold figures in the tables. Even in case of long memory latency, when $T_{prefetch}$ is large, the algorithm still gives good overall schedules without doing any unfolding. Almost all of the resulting schedules are very close to the optimal. In Table 3, the average ratio of the schedule length from the PSP algorithm to that from list scheduling and PBS are 63.2% and 84.9%, respectively; and in Table 4, 26.7% and 60.9% respectively. Moreover, since we do not unfold the graph, the computation time of this algorithm is very little. Almost all the experiments are finished in less than two to three seconds. Comparing Tables 3 and 4, we also see that when the memory latency is increased, the PSP algorithm tends to create a larger partition in order to compensate for this long latency. It shows that the larger the partition, the closer the average schedule length is to the lower bound, because the overhead (T_{keep}) control steps are amortized over more iterations.

References

[1] F. Chen, S. Tongsima, and E. H.-M. Sha. Loop scheduling optimization with data prefetching based on multi-dimensional retiming. In *Proc. ISCA 11th International Conference on Parallel and Distributed Computing Systems*, pages 129–134, 1998.

[2] F. Dahlgren and M. Dubois. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 7*, pages 733–746, Jul. 1995.

[3] N. L. Passos and Edwin H.-M. Sha. Scheduling of uniform multi-dimensional systems under resource constraints. To appear in the IEEE Transactions on VLSI systems.

[4] N. L. Passos and Edwin H.-M. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 11,*, pages 1150–1163, Nov. 1996.

[5] J. Skeppstedt and M. Dubois. Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In *the Proceedings of the International Conference on Parallel Processing*, pages 298–305, 1997.

[6] M. K. Tcheun, H. Yoon, and S. R. Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *the Proceedings of the International Conference on Parallel Processing*, pages 306–313, 1997.