Optimal Partitioning and Balanced Scheduling with the Maximal Overlap of Data Footprints *

Zhong Wang
Dept. of Comp. Sci. & Engr
University of Notre Dame
Notre Dame, IN 46556
zwang1@cse.nd.edu

Edwin H.-M. Sha
Dept. of Comp. Sci.
University of Texas at Dallas
Richardson, TX 75083
edsha@utdallas.edu

Yuke Wang
Dept. of Comp. Sci.
University of Texas at Dallas
Richardson, TX 75083

ABSTRACT

The paper proposes a scheme to tolerate the slow memory access latency for loop intensive applications in the system with memory hierarchy. The scheme takes into consideration of both the intermediate data and maximal overlap of data footprints for initial data. Furthermore, a schedule is presented to balance the ALU computation and memory operations. The memory requirement under such schedule is calculated. This schedule's improvement in total execution time is approximately 20% over existing methods.

1. INTRODUCTION

The contemporary DSP and embedded systems always contain the memory hierarchy, which can be simplified as on-chip and off-chip memories. In general, the on-chip memory has the fast speed and restrictive size, while the off-chip memory has the much slower speed and larger size. During the execution, the data are loaded from the off-chip to on-chip memories for the computation. Consequently, the system performance is degraded due to this long off-chip access latency. How to tolerate the memory latency with memory hierarchy is becoming a more and more important problem [9]. In this paper, We abstract the on-chip and off-chip memories as the first and second level memories, respectively.

The prefetching techniques are proposed to attack the problem of tolerating the memory latency without considering the memory hierarchy. They can be classified into two categories – hardware-based [4, 3] and software-directed prefetching [9]. Hardware-based prefetching shows a poor performance when the program has complex access patterns, since such kind of pattern information is difficult to catch dynamicly in the execution stage. The traditional software-directed prefetching generally has little consideration on how to increase the program's parallelism. On the contrary of the prefetching techniques, multi-dimensional loop pipelining techniques [7] can be used to explore more parallelism, even to the extent of achieving full-parallelism for uniformly nested loops. These techniques, however, do not consider the memory latency in their algorithms.

*This work is partially supported by CAM Graduate Fellowship

In this paper, we combine the loop pipelining technique with the data prefetching approach. Multiple *memory units*, attached to the first level memory, will perform operations to prefetch data from the second to the first level memories. These *memory units* are in charge of preparing all data required by the computation in the first level memory in advance of computation. Multiple *ALU units* exist in the processor for doing the computation. The operations in the ALU units and memory units execute simultaneously. Therefore, the remote memory access latency is tolerated by overlapping the data fetching operations with the ALU operations. This paper presents an approach to balance the ALU and memory parts to achieve an optimal overall schedule length. Furthermore, the memory requirement of the first level memory to implement this overall schedule is calculated.

Loop tiling [8, 5] is a technique used to group basic computations so as to increase computation granularity and thereby reduce communication time. The traditional loop tiling techniques are mainly applied to the distributed computer architecture in which each processor has his own local memory, which is different from our model –multi-processors share a common memory hierarchy, which can be found in embedded computer system [6], multi-processors computer, etc. The traditional loop tiling techniques find a right tile shape to efficiently reduce the communication cost among multiple processors without considering the scheduling of the basic operations. The approaches in [2, 10] are the few approaches to consider the detailed scheduling under memory hierarchy. However, their memory references consider only the intermediate data, and ignore the initial data, which are an important influence factor for performance.

In our approach, we consider both intermediate and initial data. The *intermediate data* will be changed by value during the computation. They can appear as the left or right operands in the equation. On the other hand, the *initial data* will keep their value during the computation. They can only appear as the right operands in the equation.

To increase the data locality, the iterations are grouped as partitions. We use the concept *footprint* [1] to denote those initial data required by the computations in one partition. Given a partition shape, we present an polynomial algorithm to find a partition size which can give rise to the maximum overlap between the adjacent footprints such that the number of memory operations is reduced to the largest extent. The new algorithm in this paper exceeds the per-

formance of existing algorithms [2, 4] due to the fact it optimizes both ALU and memory schedules and considers the influence of initial data.

2. BACKGROUND

A loop body can be represented by a *multi-dimensional data flow graph* (MDFG) [7]. Each node in the MDFG is a computation. Each edge denotes the data dependence between two computations, with its weight as the distance vector. The execution of all nodes in a *MDFG* for one time is called an *iteration*. It corresponds to executing the loop body once. Iterations are identified by a vector $\vec{i}$, equivalent to the multi-dimensional index.

In this paper, we will always illustrate our ideas under two dimensional loops. Using the same idea presented in this paper. it is not difficult to extend to loops with a higher dimension.

2.1 Architecture model

The technique in our paper is designed for use in a system with one or more processors. These processors share a memory hierarchy, as shown in Fig 1. Multiple ALU and memory units exist. Accessing the first level is significantly slower than the second level memory. 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 second level memory, which will cost much more time. Therefore, prefetching data into the first level memory before its explicit use can minimize the overall execution time. Two types of memory operations, *prefetch* and *keep* are supported by the memory units. Both of them are issued to get those data ready in the first level memory for the near future references. It is important to note that the first level memory in this model cannot be regarded as a pure cache, because we do not consider the cache associativity. In other words, it can be thought of as a full-associative cache.

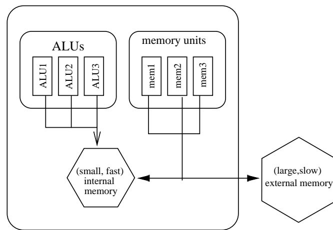


Figure 1: The overall schedule

2.2 Partitioning the iteration space

The loop execution corresponds to executing all the iterations in the iteration space. Regular nested loop execution proceeds in either a row-wise or column-wise manner. However, this mode of execution does not take full advantage of either the locality of reference or the available parallelism. The execution of such structures can be more efficient by dividing the entire iteration space into regions called partitions that better exploit spatial locality.

Provided that the total iteration space is divided into partitions of iterations, the execution sequence will be determined by each partition. Assume that the partition in which the loop is executing is the *current partition*. Then the *next partition* is the partition adjacent on the right side of the current partition along the X-axis. The *other partitions* are all partitions except the above two partitions. Based on this classification, different memory operations will be assigned to different data in a partition. For a delay dependency that goes into the next partition, a keep memory operation is used to keep this data in the first level memory for one partition because of its immediate reuse in the next partition. Delay dependencies that go into other partitions result in using prefetch memory operations to fetch data in advance.

A partition is determined by its shape and size. We use two *basic vectors* (In a basic vector, each element is an integer and all elements have no common factor except 1) – P_x and P_y to identify a parallelogram as the partition shape. These two basic vectors will be called *partition vectors* in the paper. Assume without loss of generality, the angle between P_x and P_y is less than 180°, and P_x is clockwise of P_y. Then the partition shape and size can be denoted by the direction and the multiple of two partition vectors.

How to find the optimal partition size will be discussed in the fourth section. Here we present the property of a partition shape.

PROPERTY 2.1. *A pair of partition vectors that satisfy the following constraints is legal. For each delay vector d_e, the following cross product* [1] *relations hold.* $d_e \times P_x \leq 0$ *and* $d_e \times P_y \geq 0$

Because nested loops should follow the lexicographical order, we can choose $(1,0)$ as our P_x vector and use the normalized leftmost vector of all delay dependencies as our P_y.

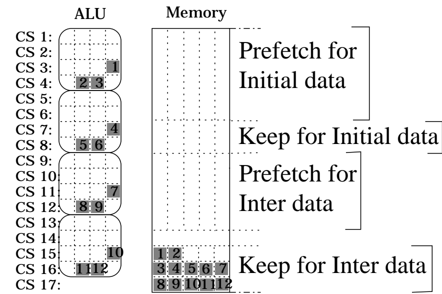


Figure 2: The overall schedule

An *overall schedule* for a partition consists of two parts: the ALU and memory parts, as seen in Figure 2. The ALU part schedules the ALU computations. Since the computation in a loop can be represented by an MDFG. The ALU part is a schedule of these MDFG nodes. The memory part schedules the memory operations – prefetch and keep operations for both initial and intermediate data.

3. THE THEORY ABOUT INITIAL DATA

The footprint for a partition consists of all the initial data required by the computations in this partition. It can be determined by the partition itself and the *displacement vector* of the initial data. The displacement vector is defined as the difference between the loop and initial data indexes. For instance, given the expression of the initial data $s[n+1,m+2]$ and loop index $[n,m]$, the displacement vector is $(1,2)$. The corresponding part in footprint for this initial data is the set of all the integer points lying in a parallelogram

[1]The 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 $p_1 \times p_2 = p_1.xp_2.y - p_1.yp_2.x$.

formed by moving the left lower corner of the partition from (x,y) to $(x+3,y+2)$.

The major concern on initial data is to maximize the overlap between the footprints for two adjacent partitions, and comply with the memory size restriction. More overlap will lead to less number of prefetch operations, because the corresponding data can be kept in the memory for the next partition. A larger partition may lead to a larger overlap. Nevertheless, the partition cannot be too large on account of the restrictive memory size. We first give some definitions regarding the footprint for a partition. In these definition, the set R includes all the displacement vectors for the initial data references.

DEFINITION 3.1.

1. *Given a certain partition shape $P_x \times P_y$, $S(\vec{r},\vec{s})$ is a set of integer points in a parallelogram which has the shape $P_x \times P_y$ and size $f_x \times f_y$, where $\vec{r} = (a,b)$ is a displace vector for a initial data reference, and $\vec{s} = (f_x, f_y)$ is the partition size.*
2. *Given a set of integer vectors $R = \{\vec{r}_1, \vec{r}_2, \dots, \vec{r}_n\}$, the footprint of R for a partition $F(R,\vec{s})$, is the union of all $S(\vec{r}_i,\vec{s})$ for each $\vec{r}_i$ in R.*
3. *The footprint for the next partition $F(R^s,s)$ is built upon set R^s, in which each element $\vec{r_i}^s = \vec{r}_i + f_x P_x$.*

In one dimension case, all the vectors become into integers and the second dimension elements are zero. Therefore, One dimensional case can be regarded as a simplification to the two dimensional problem. It provides the theoretic foundation to two dimensional problem. The following example demonstrates the problem in one dimensional case. In the example, the set R is $\{1,2,7\}$. The solid line represents the corresponding $S(r,s)$ for each r in the set R. The dotted line denotes the corresponding $S(r^s,s)$ for each r^s in set R^s. The union of all solid lines is set $F(R,s)$, while the union of all dotted lines is the set $F(R^s,s)$. The figure shows the case when $s = 5$, which is the minimum s value to obtain the maximum intersection between $F(R,s)$ and $F(R^s,s)$.

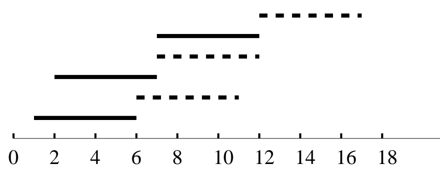


Figure 3: The one dimensional line segments

The following two lemmas show the properties of the intersection between two adjacent footprints.

LEMMA 3.1. *The minimum s is $r_2 - r_1$ which makes the maximum intersection between segments (r_1+s, r_1+2s-1) and (r_2, r_2+s-1), where $r_2 \geq r_1$.*

LEMMA 3.2. *For the intersection between segments (r_1+s, r_1+2s-1) and (r_2, r_2+s-1), where $r_2 \geq r_1$, it will keep constant, irrelevant to the value of s, as long as $s \geq r_2 - r_1$.*

Both $F(R,s)$ and $F(R^s,s)$ are the union of some line segments. Their overlap is the union of all intersections of two line segments from $F(R,s)$ and $F(R^s,s)$, respectively.

$$F(R,s) = (r_1,r_1+s-1)\cup(r_2,r_2+s-1)\cup(r_3,r_3+s-1)\cdots\cup(r_n,r_n+s-1)$$
$$F(R^s,s) = (r_1+s,r_1+2s-1)\cup(r_2+s,r_2+2s-1)\cdots\cup(r_n+s,r_n+2s-1)$$

At following, we reveal the relation between the overlap of the adjacent footprints and the line segment size s.

LEMMA 3.3. *Let C_m be the intersection $(r_m, r_m+s-1) \cap (r_{m-1}+s, r_{m-1}+2s-1)$. Then the intersection of $F(R,s)$ and $F(R^s,s)$ is $\bigcup_2^n C_m$, where the number of integers in R is n.*

THEOREM 3.4. *Given the set $R = (r_1, r_2, r_3, \dots, r_n)$, the maximum intersection between $F(R,s)$ and $F(R^s,s)$ can be achieved when $s = max_{m=2}^{n}(r_m - r_{m-1})$.*

THEOREM 3.5. *For two sets $F(R,s)$ and $F(R^s,s)$, the intersection of these two sets will keep constant if the value of s continues to increase from the s value obtained by the Theorem 3.4.*

When considering to maximize the overlap between two adjacent footprints in the 2-dimensional case, we can notice that f_y element of the partition size is not so important as f_x element, since the intersection always increases when f_y is increased. As indicated later, the value of f_y can be determined based on other conditions. Therefore, the key is what is the minimum value of f_x to make the intersection maximal under a certain f_y.

Given a certain partition size of $\vec{s}$ and the set R, an augment set R' can be obtained with the following method : $r'_i = r_i$, $r'_{i+n} = r_i + f_y P_y.y$, where n is the size of set R and $P_y = (P_y.x, P_y.y)$. Arranging all the points in set R' with the increasing order along Y axis, the overall footprint for one partition can be divided into a series of stripes. Each stripe is determined by two horizontal lines which pass the adjacent two points in sorted R'. For instance, in the Figure 4, the R set is $\{(0,0),(6,1),(3,2),(1,3)\}$. Assume the value of $f_y P_y.y$ is 5, R' is $\{(0,0),(0,5),(6,1),(6,6),(3,2),(3,7),(1,3),(1,8)\}$. After sorting, it will become $\{(0,0),(6,1),(3,2),(1,3),(0,5),(6,6),(3,7),(1,8)\}$. The overall footprint consists of 7 stripes as indicated in the figure.

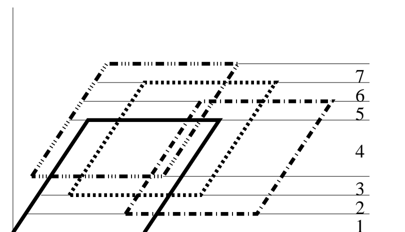
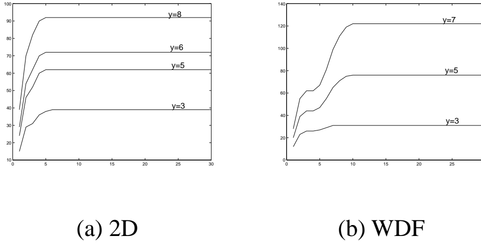


Figure 4: The stripe division of a footprint

In each stripe, a horizontal line will intersect with left bounds of some sets $S(\vec{r},\vec{s})$. Thus, the two dimensional footprint overlap problem of this stripe can be reduced to one dimensional problem, which can be solved using Theorem 3.4. Applying this idea to each stripe,

Algorithm 1 Calculating the minimum x to make the overlap maximum

Input: The set R and the shape of the partition
Output: The f_x to make the overlap maximum under a certain f_y
1. Set f_x to 0.
2. Based on the set R and partition shape, choose a f_y such that the product $f_y * P_y.y$ is larger than the difference between the largest and least b element of all vectors in set R.
3. Using the f_y above, generate the augment set R'
4. Sort all the value in the R' in increasing order according to the b element and kept them in a event list.
5. Use a horizontal line to sweep the whole iteration space. When a lower bound event point is met, insert the corresponding set $S(\vec{r},\vec{s})$ in a visiting list. Otherwise delete the corresponding $S(\vec{r},\vec{s})$ from the list.
6. Calculate the intersection point of this line with the leftbound and righbound of each set in the visiting list, respectively. Use Theorem 3.4 to derive a f'_x value to make the intersection in current stripe maximal.
7. Replace f_x with f'_x if $f'_x > f_x$.



(a) 2D (b) WDF

Figure 5: The tendency of intersection with x and y

we can solve the whole two dimensional problem, as demonstrated in the following algorithm.

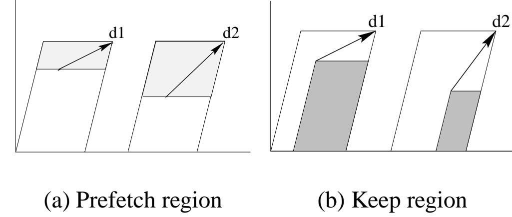
From the Lemma 3.2, the intersection will keep unchanged if f_x is greater than the value chosen by this algorithm, and reduce under the less f_x. We can demonstrate this phenomenon by two examples. The set R for the first example is $\{(0,1),(5,3),(-3,1),(4,-1),(-2,-2)\}$ and the partition shape is $(1,0)\times(0,1)$. It is the partition shape for *Wave Digital filter*. The set R is $\{(0,2),(3,5),(1,3)(-1,-1)\}$ in the second example and the partition shape is $(1,0)\times(-3,1)$. It is the partition shape for *Two Dimensional filter*. The Figure 5(a) and 5(b) show the varying trends of footprint overlap with the value of f_x and f_y for two examples, respectively.

4. THE OVERALL SCHEDULE

The overall schedule can be divided into two parts – ALU and memory schedules. For the ALU schedule, the *multi-dimensional rotation scheduling algorithm* [7] is used to generate a static schedule for one iteration. Then the entire ALU schedule can be formed by simply replicating this schedule for each iteration in the partition. The schedule obtained by this way is the most compact schedule since it only considers the ALU hardware resource constraints. The overall schedule length must be larger than it. Therefore, this ALU schedule provides a lower bound for the overall schedule. This lower bound can be calculated by $\#len_{iteration} \times \#nodes$, where $len_{iteration}$ represents the schedule length obtained by multi-dimensional rotation scheduling algorithm for one iteration, and $\#nodes$ denotes the number of nodes in one partition. Our objective is to find a partition whose overall schedule length can be very close to this lower bound.

4.1 Balanced overall schedule

Different from the ALU schedule, which is the replication of a single iteration schedule, The memory schedule is considered based on the entire partition. It schedules the memory operations for initial and intermediate data, as shown in Figure 2. Both operations consist of the prefetch and keep operations for the corresponding data. Since all the prefetch operations have no relation to the current computation, they can be arranged at the start of the memory schedule. On the contrary, the keep operation for intermediate data can only be issued after the corresponding computation has finished. The keep operations for initial data can be issued as soon as they are prefetched from the second level memory. The memory schedule length is the summation of all these operations, execution time.



(a) Prefetch region (b) Keep region

Figure 6: The tendency of intersection with x and y

For the intermediate data, a prefetch operation is necessary if there is a delay dependency going to the other partition from the current partition. Thereby each delay vector will delimit a parallelogram in which the corresponding data are prefetched from the second level memory. The number of prefetch operations for this delay dependency is the area of this parallelogram. The overall number of prefetch operations for intermediate data in one partition is all such parallelogram for all delay dependencies. It is the summation of all the parallelograms' area under the condition that there is no common intersection among the parallelograms. If such intersection does exist, we have the definition of *equivalent prefetch class*. An equivalent prefetch class comprises of all those prefetch operations with the delay dependencies that start from the same data node and go into the same other partition. The prefetch operation will be counted only once for each equivalent prefetch class. The Figure 6(a) shows the example with two delay dependencies exist in the partition. Those prefetching parallelograms are the shadow regions decided by two delay dependency d_1 and d_2, respectively. The overall number of prefetch operations for these two delays is the summation of the two areas, except those equivalent prefetch classes.

For the initial data, they can be prefetched from the second level memory in blocks. This kind of operation can fetch several data at one time and costs only a little longer time than general prefetch operation. To calculate the number of such operations, we first have the following observation.

PROPERTY 4.1. *As long as $f_y * P_y.y$, the projection of partition size along Y coordinate, is larger than the maximum difference of the Y coordinates in all displacement vectors for initial data, the footprint will increase at a constant rate with the increment of f_y, so does the number of prefetch operations for initial data.*

Therefore, given a certain f_x and f_y, which satisfying the condition

in the above property, the number of prefetch operations for the initial data is $Pre_{Base_ini} + (f_y - f_{y_0}) \times Pre_{incr_ini}$, where $f_{y_0} = \lceil \frac{y_0}{P_y.y} \rceil$, y_0 is the maximal distance of the Y coordinates in all displacement vectors, Pre_{Base_ini} denotes the number of prefetch operations for a partition with size $f_x \times f_{y_0}$, and Pre_{incr_ini} represents the increment of the number of prefetch operations when f_y is increased by one.

As the prefetch operation, there are keep operations for the intermediate and initial data. The number of keep operations for the intermediate data can be calculated by the same principle for prefetch operations. The definition of an *equivalent keep class* is similar to the *equivalent prefetch class*. An equivalent keep class comprises of all those keep operations with the delay dependencies that start from the same data node. The Figure 6(b) shows the corresponding keep regions for those two delay dependencies.

To the initial data, the intersection of the current and next footprints will be needed by both partitions. They can be kept in the first level memory, thereby spare some high-cost prefetch operations. This is the reason to use minimal value of f_x to achieve the maximal intersection. Different from keep operations for the intermediate data, these operations have no relation to the current partition's computation. They can be arranged after they have been prefetched from the memory. The number of such operations is $Keep_{Base_ini} + (f_y - f_{y_0}) \times Keep_{incr_ini}$, where y_0 and f_{y_0} have the same meaning as above. $Keep_{Base_ini}$ denotes the number of keep operations for a partition with size $f_x \times f_{y_0}$, and $Keep_{incr_ini}$ represents the increment of keep operations when f_y is increased by one.

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

DEFINITION 4.1. *A* **balanced overall schedule** *is a schedule in which the memory schedule is at most one unit time of keep operation longer than the ALU schedule.*

To obtain a balanced overall schedule, some conditions need to be satisfied. In theorem 4.1, $d = (d_x, d_y)$ denotes a delay dependency. $\#pre$ and $\#keep$ are the number of the prefetch and keep operations for intermediate data. T_{pre} is the time consumption of a prefetch operation, T_{block_pre} is the time consumption of a block prefetch operation for initial data, and T_{keep} is the time needed for a keep operation. L_{ALU} represents the ALU schedule length for one iteration and $\#iter$ is the number of iterations in one partition. N_{mem} is the number of memory units. Other symbols have the same meanings as we discussed above.

THEOREM 4.1. *The size of the partition should satisfy the following three constraints to make the overall schedule balanced.*

1. *There is no delay dependency which can span more than two partitions along the Y coordinate direction, that is* $f_y * P_y.y \geq d_y, \forall d = (d_x, d_y) \in D$.
2. *There is no delay dependency which can span more than two partitions along the X coordinate direction, that is* $f_x > max\{d_x - d_y \frac{P_y.y}{P_y.x}\}$.
3. $\left\lceil \frac{\#pre}{N_{mem}} \right\rceil * T_{pre} + \lceil \frac{Pre_{Base_ini} + (f_y - f_{y_0}) \times Pre_{incr_ini}}{N_{mem}} \rceil * T_{block_pre} + \left\lceil \frac{\#keep + Keep_{Base_ini} + (f_y - f_{y_0}) \times Keep_{incr_ini}}{N_{mem}} \right\rceil * T_{keep} \leq L_{ALU} * \#\mathtt{iter} + T_{keep}$.

The theorem not only gives the balanced overall schedule's constraints. but also provides the method to find a partition size which can generate a balanced overall schedule.

4.2 Algorithm for finding balanced overall schedule

As long as the first two conditions in Theorem 4.1 are satisfied, we can guarantee that, when partition size is enlarged, the overall time consumption of prefetch and keep operations for intermediate data in memory schedule increases slower than the ALU schedule length. At this time, if a partition size cannot be found to meet the third constraints, it means the time consumption of the block prefetch operations for initial data increased too fast. Due to the property of block prefetch, increasing f_x will increase the number of block prefetch operations only by a small number, while increase the ALU schedule by a relative large length. Consequently, a partition size which can satisfy all of the three conditions can be found.

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

Algorithm 2 Find a balanced overall schedule

Input: The ALU schedule for one iteration, the partition shape $P_x \times P_y$ and the initial data displacement vector set R
Output: A partition size which can generate a balanced overall schedule

1. Based on the information of the initial data, use algorithm 1 to calculate the minimum partition size f'_x and f'_y
2. Using the first two condition in Theorem 4.1 to calculate another pair of minimal f''_x and f''_y
3. Get a new pair $f_x = max(f'_x, f''_x)$ and $f_y = max(f'_y, f''_y)$.
4. Using this pair (f_x, f_y), calculate the number of prefetch operations, block prefetch operations and keep operations.
5. Calculate the ALU schedule length to see if the third condition in Theorem 4.1 is satisfied.
6. If it is satisfied, this pair (f_x, f_y) is the partition size. Otherwise, increase f_x by one, use the third condition to find the minimal f_y which make the inequation true. If such f_y does not exist, continue increasing f_x until the feasible f_y is found. Use them as the partition size.
7. Based on the partition size, Output the corresponding ALU part and memory schedules.

4.3 The memory requirement for the overall schedule

The memory requirement consists of four parts, the memory space for the calculation of the in-partition data, for prefetching intermediate data, for keeping intermediate data and for those operations on initial data. We will discuss these four part memory requirements respectively.

For those in-partition data, they will be computed and reused in the same partition. Their memory requirement computation is a little complicated due to the consideration of memory reuse, which can be refer to /citeDAC.

For the other part memory requirements, they can be computed simply by multiply the number of operations with the memory requirement of each operation. The memory requirement for a

Benchmark	Par Vector		New algo			PSP algo				list		hardware	
	Px	Py	size	m_r	len	size	m_r	len	ratio	len	ratio	len	ratio
WDF	(1,0)	(-3, 1)	4×7	221	**4.107**	4×4	143	5.312	22.68%	18	77.18%	10	58.93%
IIR	(1,0)	(-2, 1)	4×9	407	**6.028**	4×7	350	6.893	12.55%	36	83.26%	37	83.71%
DPCM	(1,0)	(-2, 1)	8×10	736	**4.01**	8×8	628	4.891	18.01%	25	83.96%	21	80.9%
2D	(1,0)	(0,1)	3×5	233	**12**	3×4	207	12	0.0%	55	78.18%	51	76.47%
Floyd	(1,0)	(-3,1)	7×5	301	**6.057**	4×4	174	6.312	4.04%	32	81.72%	30	79.81%

Table 1: Experimental results with only one initial data

Benchmark	par Vector		New algo			PSP algo				List		hardware	
	Vx	Vy	size	m_r	len	size	m_r	len	ratio	len	ratio	len	ratio
WDF	(1,0)	(-3, 1)	8×7	474	**4.018**	4×4	206	8	49.78%	22	81.74%	10	58.92%
IIR	(1,0)	(-2, 1)	5×13	772	**6.015**	4×7	472	7.857	23.44%	40	84.96%	37	83.74 %
DPCM	(1,0)	(-2, 1)	8×14	1207	**4.001**	8×8	811	5.266	24.02%	29	86.2%	21	80.95%
2D	(1,0)	(0,1)	4×5	346	**12**	3×4	253	13.833	13.25%	59	79.66%	51	76.47%
Floyd	(1,0)	(-3,1)	8×6	526	**6**	4×4	223	8.812	31.91%	36	83.33%	30	80%

Table 2: Experimental results with three initial data

prefetch operation is 2. One is used to store the data prefetched by the previous partition and consumed in the current partition, the other stores the data prefetched by the current partition and consumed in the next partition. As the same rule, the keep operation will take 2 memory locations. The block prefetch operations will take $2\times$ *block_size* memory locations.

5. EXPERIMENT

In this section, we use several DSP benchmarks to illustrate the effectiveness of our new algorithm. They are WDF, IIR, DPCM, 2D and Floyd, as indicated in the tables, which stand for *Wave Digital filter, Infinite Impulse Response filter, Differential Pulse-Code Modulation device, Two Dimensional filter and Folyd-Steinberg algorithm*, respectively. We applied four different algorithms on these benchmarks: list scheduling, hardware prefetching scheme, PSP partition algorithm [2] and our new partition algorithm. In list scheduling, the same architecture model is used. However, the ALU part uses the traditional list scheduling algorithm, and the iteration space is not partitioned. In hardware prefetching scheduling, we use the model presented in [4]. In this model, whenever a block is accessed, the next block is also loaded.

In the experiment, we assume a prefetch time of 10 CPU clock cycles and a block prefetch time of 16 CPU clock cycles, which is reasonable when the big performance gap between CPU and the main memory is considered. The first table presents results with only one initial data with the displacement vector $(1,1)$, and the second table is results with three initial data with the displacement vector set $\{(1,1),(2,-2),(0,3)\}$. In the tables, the *par vector* column determines the partition shape. The *list* column includes the schedule length for list scheduling and the improvement ratio our algorithm can get compared to list scheduling. The *hardware* column includes the schedule length for hardware prefetching and our algorithm's relative improvement ratio. In the *PSP algo* and *new algo* columns, the *size* column is the size of partition presented with the multiple of the partition vectors. The *m_r* column represents the corresponding memory requirement and the *len* column is the average scheduling length for corresponding algorithms. The *ratio* column in *PSP algo* is the improvement our new algorithm can get relative to the PSP algorithm.

All the experiments are done on SUN Ultra-SPARC2 platform. Given a *MDFG* for DSP filter and the initial data *displacement vector* set, a program is run to determine the balanced partition size and the corresponding ALU, memory schedules. The computation for each DSP filter can be finished in no more than 2 seconds.

As we can see from these tables, list scheduling and hardware prefetching scheduling have much worse performance than other two algorithms, due to the reason that long memory part schedule dominates the overall schedule and is far from the balanced schedule. The PSP partition algorithm gets the worse results because of the lacking of consideration on the initial data. The time cost for the initial data will lead to an unbalanced schedule. Our new algorithm considers both data locality and the initial data. Therefore, the much better performance can be achieved through balancing the ALU part and memory part schedule.

6. REFERENCES

[1] Anant Agarwal, David A. Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans on Parallel and Distributed Systems*, 6(9), September 1995.

[2] F.Chen and E.H.-M.Sha. Loop scheduling and partitions for hiding memory latencies. In *Proc. IEEE 12th International Symposium on System Synthesis*, pages 64–70, San Jose, November 1999.

[3] F.Dahlgren and M.Dubois. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7), July 1995.

[4] 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.

[5] J.Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *Proc. of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[6] J. Madsen and P.Bjorn-horgensen. Embedded system synthesis under memory constraints. In *Proc. of 7th International Workshop on Hardware/Software Codesign*, pages 188–193, Rome, May 1999.

[7] N.Passos and E.H.-M.Sha. Scheduling of uniform multi-dimensioanl systems under resource constraints. *Journal of IEEE Transactions on VLSI Systems*, 6(4), December 1998.

[8] P.Bouilet, A.Darte, T.Risset, and Y.Robert. (pen)-ultimate tiling. In *Scalable High-Performance Computing Conference*, pages 568–576, May 1994.

[9] T.Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.

[10] Z. Wang, M. Kirkpatrick, and E.H.-M.Sha. Optimal two level partitioning and loop scheduling for hiding memory latency for dsp applications. In *Proc. ACM 37th Design Automation Conference*, pages 540–545, Los Angeles, California, June 2000.