EURASIP Journal on Applied Signal Processing 2002:9, 926–935
© 2002 Hindawi Publishing Corporation

Partitioning and Scheduling DSP Applications with Maximal Memory Access Hiding

Zhong Wang
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA
Email: zwang1@cse.nd.edu

Edwin Hsing-Mean Sha
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA
Email: edsha@utdallas.edu

Yuke Wang
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA
Email: yuke@utdallas.edu

Received 2 September 2001 and in revised form 14 May 2002

This paper presents an iteration space partitioning scheme to reduce the CPU idle time due to the long memory access latency. We take into consideration both the data accesses of intermediate and initial data. An algorithm is proposed to find the largest overlap for initial data to reduce the entire memory traffic. In order to efficiently hide the memory latency, another algorithm is developed to balance the ALU and memory schedules. The experiments on DSP benchmarks show that the algorithms significantly outperform the known existing methods.

Keywords and phrases: loop pipelining, initial data, maximal overlap, balanced partition scheduling.

1. INTRODUCTION

The contemporary DSP and embedded systems always contain the memory hierarchy, which can be categorized as on-chip and off-chip memories. In general, the on-chip memory have a fast speed and restrictive size, while the off-chip memory have the much slower speed and larger size. To do the CPU's computation, the data need to be loaded from the off-chip to on-chip memories. Thus, the system performance will be 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 [1]. The on-chip and off-chip memories are abstracted as the first and second level memories, respectively, in this paper.

Prefetching [1, 2, 3, 4, 5] is a technique to fetch the data from the memory in advance of the corresponding computations. It can be used to hide the memory latency. On the other hand, software pipelining [6] and modulo scheduling [7, 8] are the scheduling techniques used to explore the parallelism in the loop. Both the prefetching and scheduling techniques can be used to accelerate the execution speed. However, these traditional techniques have some weaknesses [9] such that they cannot efficiently solve the problem mentioned in the first paragraph. This paper combines the software 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 ALU schedule is optimized by using the software pipelining technique under the resource constraints. The operations in the ALU units and memory units execute simultaneously. Therefore, the long memory access latency is tolerated by overlapping the data fetching operations with the ALU operations. Although using computation to hide the memory latency has been studied extensively before, trying to balance the computation and memory loading has never been researched thoroughly according to the authors' knowledge. This paper presents an approach to balance the ALU and memory schedules to achieve an optimal overall schedule length.

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

$$\begin{aligned} B[i+1] &= B[i]*B[i-1]+A[i], \\ C[i+1] &= B[i-1]*A[i+1]+A[i]. \end{aligned} \tag{1}$$

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

To take full use of the data locality, the entire iteration space can be divided into small blocks named *partitions*. A lot of works have been done on the partitioning technique. Loop tiling [10, 11] is a technique used to group basic computations so as to increase computation granularity and thereby reduce communication time. Generally, they have no detailed schedule of ALU and memory operations as our method. Moreover, only intermediate data are taken into consideration. Agarwal and Kranz [12] make an extensive study of data partition. They use an approximation method to find a good partition to minimize the data transfer among the different processors. Affine reference index is considered in their work. However, they mainly concentrate on the initial data and have few consideration on the intermediate data.

The approaches in [9, 13] are the few approaches to consider the detailed schedule under memory hierarchy. Nevertheless, their memory references consider only the intermediate data, and ignore the initial data, which are an important influence factor of performance. From the experimental results in Section 5, we can see that such deficiency will lead to an unbalanced schedule, which means a worse schedule.

In our approach, both the intermediate and initial data are considered. For the intermediate data, we will restrict our study to nested loops with uniform data dependencies. The study of uniform loop nests is justified by the fact that most general linear recurrence equations can be transformed into a uniform form. This transformation (uniformization [14]) greatly reduces the complexity of the problem. On the other hand, it is difficult to implement uniformization for the initial data. Therefore, affine reference index is considered. The concept *footprint* [12] is used to denote the initial data needed for the computation of ALU units in one partition. Given a partition shape, this paper presents an algorithm to find a partition size which can give rise to the maximum overlap between the adjacent overall footprints such that the number of memory operations is reduced to the largest extent.

When considering the schedule of the loop, we propose the detailed ALU and memory schedules. Each of the memory and ALU operations are assigned to an available hardware unit and time slot. Therefore, it is very convenient to apply our technique to a compiler. The memory schedule is balanced to the ALU schedule such that the overall schedule is close to the lower bound, which is determined by the ALU schedule. Our method gives the algorithm to determine the partition shape and size in order to achieve balanced ALU and memory schedules. At last, the memory requirement of our technique for applications is also presented.

The new algorithm in this paper significantly exceeds the performance of existing algorithms [9, 13] due to the fact that it optimizes both ALU and memory schedules and considers the influence of initial data. Taking the wave digital filter as an example, in a standard system with 4 ALU units and 4 memory units, assuming 3 initial data references exist in each iteration, our algorithm can obtain an average schedule length of 4.018 CPU clock cycles, which is very close to the theoretic lower bound of 4 clock cycles. The traditional list scheduling needs 22 clock cycles. The hardware prefetching costs 10 clock cycles. While the PSP algorithm in [13] can achieve some improvement, it still needs 8 clock cycles. Without the memory constraint, the algorithm in [9] has the same performance, 8 clock cycles. Our algorithm improves all the previous approaches.

It is worthwhile to mention that some works have been done on data layout technique [15, 16], which is used to maintain the cache coherency and reduce the conflict traffic. Our work should be regarded as another different layer which can be built upon the layer of data layout to get a better performance.

The remainder of this paper is organized as follows. Section 2 introduces the terms and basic concepts used in the paper. Section 3 presents the theory on initial data. Section 4 describes the algorithm to find the detailed schedule. Section 5 contains the experimental result of comparison of this technique with a number of existing approaches. We conclude in Section 6.

2. BACKGROUND

We can represent the operations in a loop by a *multidimensional data flow graph* (MDFG) [6]. Each node in the MDFG represents a computation. Each edge denotes the data dependence between two computations, with its weight as the distance vector. The benefit of using MDFG instead of the general data dependence graph (DDG) or statement dependence graph (SDG) is that MDFG is the finer-grained description of data dependences. Each node of MDFG corresponds to one ALU computation. On the contrary, a node always corresponds to a statement in DDG or SDG, which will consume uncertain ALU computation time depending on the complexity of the statement. It is more convenient to schedule the ALU operations with MDFG. Moreover, lots of DSP applications, such as DSP filters, and so forth, can be directly mapped into MDFG [17].

The execution of all nodes in an *MDFG* one time is an *iteration*. It corresponds to executing the loop body for one time under a certain loop index. Iterations are identified by a vector $\vec{i}$, equivalent to a multidimensional index.

In this paper, we will always illustrate our ideas under two-dimensional loops. It is not difficult to extend to loops with more than two dimensions by using the same idea presented in this paper.

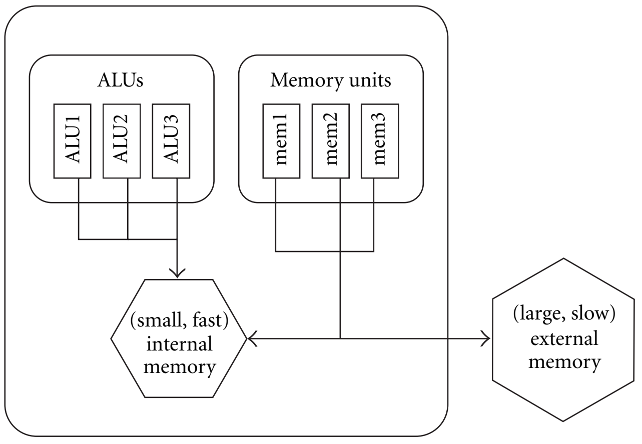


Figure 1: Architecture model with multiple function units and a memory hierarchy.

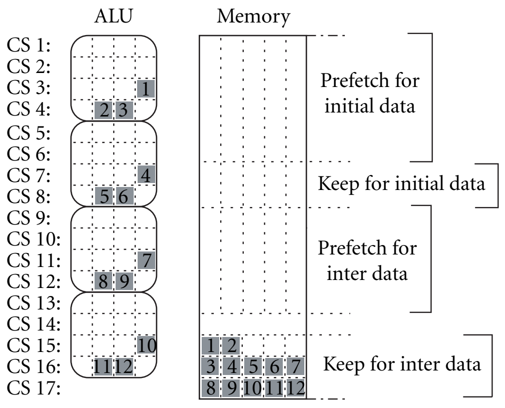


Figure 2: The overall schedule.

2.1. Architecture model

The technique in our paper is designed for use in a system which has one or more processors. These processors share a common memory hierarchy, as shown in Figure 1. There are multiple ALU and memory units in the system. The access time for the first level memory is significantly less than for the second level memory, as in current systems. 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. Thus, 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. The *prefetch* operation prefetches the data from the second level to the first level memories; the *keep* operation keeps the data in the first level memory for the execution of one partition. Both of them are issued to guarantee that those data being referenced in the near future appear in the first level memory before their 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.

2.2. Partitioning the iteration space

Regular execution of nested loops proceeds in either a row-wise or column-wise manner until the boundary of iteration space is reached. 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 made to 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, since this data will be reused immediately in the next partition. Delay dependencies that go into other partitions result in the use of prefetch memory operations to fetch data in advance.

A partition is determined by its partition shape and partition 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*. Assume, without loss of generality, that the angle between P_x and P_y is less than 180°, and P_x is clockwise of P_y. The partition size is determined by the vector $S = (f_x, f_y)$, where f_x and f_y are the multiples of the partition size over partition vectors P_x and P_y, respectively. Thus, the partition can be delimited by two vectors f_xP_x and f_yP_y.

How to find the optimal partition size will be discussed in Section 4. Due to the dependencies between the iterations, the P_x and P_y cannot be chosen arbitrarily. The following property gives the condition of a legal partition shape [9].

Property 1. A pair of partition vectors that satisfy the following constraints is legal. For each delay vector d_e, the following *cross products*[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. The partition shape is decided by these two vectors.

An *overall schedule* consists of two parts: an ALU part and a memory part, as seen in Figure 2. The ALU part sched-

[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 $p_1 \times p_2 = p_1 \cdot xp_2 \cdot y - p_1 \cdot yp_2 \cdot x$.

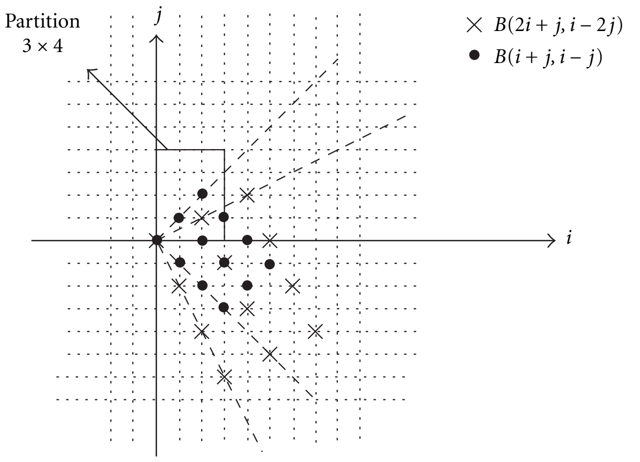


FIGURE 3: The footprint.

ules the ALU computation. We know that 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, so that the data for the computation can always be found in the first level memory.

3. THE THEORY ABOUT INITIAL DATA

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

As mentioned in Section 1, we consider affine reference for the initial data. Given a loop index vector $\vec{i}$, an affine reference index can be expressed as $\vec{g}(\vec{i}) = \vec{i}G + \vec{a}$, where $G = [\vec{G_1}\vec{G_2}]$ is a 2×2 matrix and $\vec{a}$ is the *offset vector*. The *footprint with respect to a reference* $A[\vec{g_1}(\vec{i})]$ is the set of all data elements $A[\vec{g_1}(\vec{i})]$ of A, for $\vec{i}$ an element of the partition. The overall footprint is the union of the footprints with respect to all different references. For example, in Figure 3, the partition is a rectangle with size 3×4. The initial data references are $B(i + j, i - j)$ and $B(2i + j, i - 2j)$. Their corresponding footprints are denoted by those integer points marked by × and •, respectively. The overall footprint is the union of these two footprints.

In [12], Anant presents the concept *uniformly generated references. Two references* $A[\vec{g_1}(\vec{i})]$ *and* $A[\vec{g_2}(\vec{i})]$ *are said to be uniformly generated if*

$$\vec{g_1}(\vec{i}) = \vec{i}G + \vec{a_1}, \qquad \vec{g_2}(\vec{i}) = \vec{i}G + \vec{a_2}. \tag{2}$$

If two references B_1 and B_2 are not uniformly generated, the overlap between footprint with respect to B_1 of the current partition and that with respect to B_2 of the next partition can be ignored because the overlap, if exists, diminishes rapidly. Therefore, we need only consider the overlap between footprints with respect to uniformly generated references of two consecutive partitions. Moreover, the offset vector $\vec{a}$ should satisfy that $\vec{a} = m\vec{G_1} + n\vec{G_2}$, where m and n are integer constants. Otherwise, no overlap between the footprints of consecutive partitions will exist even for the uniformly generated references.

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

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

(2) Given a set of uniformly generated references, the set $R = \{\vec{a_1}, \vec{a_2}, \dots, \vec{a_n}\}$ is set of offset vectors.[2] Assuming the partition size is $\vec{S}$, $F(R, \vec{S})$ is the overall footprint of the current partition and $F(R', \vec{S})$ is the overall footprint of the next partition.

The one-dimensional case can be regarded as a simplification to the two-dimensional problem, in which the f_y is always set to zero. It provides the theoretic foundation for the two-dimensional problem. In the case of one dimension, a partition is reduced to a line segment and all vectors reduce to integer numbers. The partition size can be thought of as the length of the line segment. We use an example to demonstrate the problem we are tackling. In Figure 4, there are three different offset vectors: 1, 2, 7. The solid lines represent the overall footprint of the current partition, and dotted lines denote that of the next partition. Then, we need to find the condition of the partition size, that is, the length of the line segment, to achieve a maximal overlap. The figure shows the case when the length equal 5, which is the minimum length to obtain the maximum overlap between overall footprints.

In order to derive the theorem on the minimum value S which can generate the maximum overlap, we first have

[2]Note that the elements in the set R are in lexicographically increasing order.

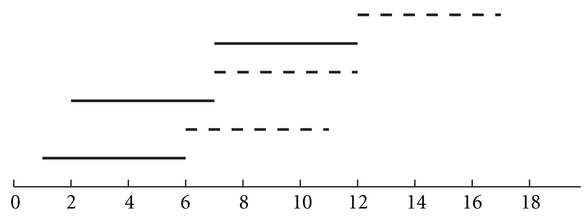


FIGURE 4: One-dimensional line segments.

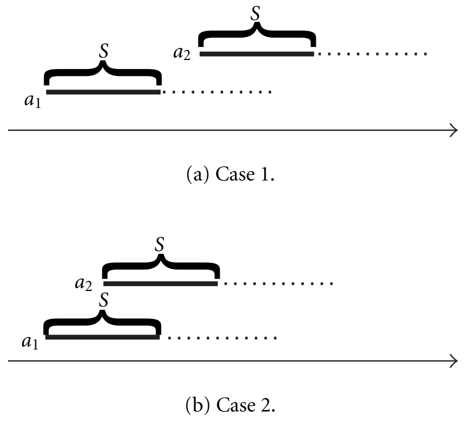


FIGURE 5: Two different relations between a_1 and a_2.

the following lemmas. They are used to consider the overlap of two footprints of the consecutive partitions, as show in Figure 5. The solid line is the footprint of the current partition and the dotted line is the footprint of the next partition.

Lemma 1. *The minimum S is $a_2 - a_1$ which makes the maximum intersection between $f(a_1', S)$ and $f(a_2, S)$, where $a_2 \geq a_1$.*

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

Case 1. As shown in Figure 5a, $a_1 + S \leq a_2$, that is, $S \leq a_2 - a_1$. The intersection is $(a_2, a_1 + 2S - 1)$. It can reach the maximum value $a_2 - a_1$ when $S = a_2 - a_1$.

Case 2. As shown in Figure 5b, $a_1 + S > a_2$, that is, $S > a_2 - a_1$. The intersection of two segments is $(a_1 + S, a_2 + S - 1)$. It has no relation to S. This means the size of intersection will not increase in spite of the increment of S. □

Lemma 2. *For the intersection between $f(a_1', S)$ and $f(a_2, S)$, where $a_2 \geq a_1$, it will keep constant, irrelevant to the value of S, as long as $S \geq a_2 - a_1$.*

According to Definition 1, $F(R, S)$ and $F(R', S)$ can be expressed as

$$F(R, S) = f(a_1, S) \cup f(a_2, S) \cup \cdots \cup f(a_n, S),$$
$$F(R', S) = f(a_1', S) \cup f(r_2', S) \cup \cdots \cup f(r_n', S). \tag{3}$$

The following lemma gives the expression of their intersection.

Lemma 3. *Let C_m be the intersection $f(a_m, S) \cap f(a_{m-1}', S)$. Then the intersection of $F(R, S)$ and $F(R', S)$ is $\bigcup_2^n C_m$, where the number of integers in R is n.*

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

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

Induction hypothesis. Assume that, for some $n \geq 2$, $F(R, S) \cap F(R', S) = \bigcup_2^n C_n$.

Induction step. For $n + 1$, the added intersection is $A_{n+1} \cap (B_1 \cup B_2 \cup \cdots \cup B_n)$. There are two different cases.

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

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

$$\begin{aligned} & A_{n+1} \cap (B_1 \cup B_2 \cup \cdots \cup B_n) \\ &\quad = A' \cap (B_1 \cup B_2 \cup \cdots \cup B_n) \\ &\qquad \cup A'' \cap (B_1 \cup B_2 \cup \cdots \cup B_n) \\ &\quad \subseteq \bigcup_2^n C_n \cup (A_{n+1} \cap B_n) \\ &\quad = C_{n+1}. \end{aligned} \tag{4}$$

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

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

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

From Theorem 1 and Lemma 2, we can directly derive the following theorem.

Theorem 2. *For the overall footprints $F(R, S)$ and $F(R', S)$, their overlap will keep constant if the value of S continues to increase from the S value obtained by Theorem 1.*

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

Next, we discuss the situation with G a two-dimensional identity matrix. If G is not an identity matrix, the same idea

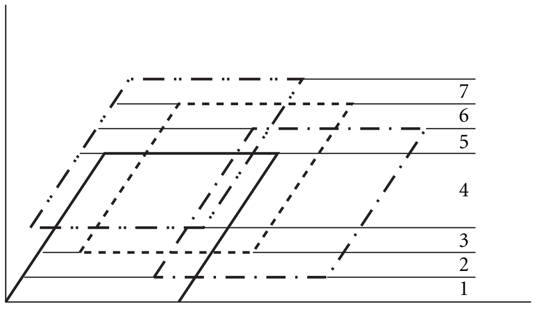


FIGURE 6: The stripe division of a footprint.

can be applied as long as $\vec{a} = m\vec{G_1} + n\vec{G_2}$. The only difference is that the original XY-space will be transformed to the new space by the G matrix. An augment set R^* can be obtained based on a certain partition size of $\vec{S}$ and the set R with the following method: $a_i^* = a_i$, $a_{i+n}^* = a_i + f_y P_y \cdot y$, where n is the size of the set R and $P_y = (P_y \cdot x, P_y \cdot y)$. Arranging all the points in the set R^* with the increasing order of the Y element, the overall footprint of one partition can be divided into a series of stripes. Each stripe is determined by two horizontal lines which pass the two adjacent points sorted in R^*. For instance, in Figure 6, the R set is $\{(0,0),(6,1),(3,2),(1,3)\}$. Assume the value of $f_y P_y \cdot y$ is 5, then the augment set R^* is $\{(0,0),(0,5),(6,1),(6,6),(3,2),(3,7),(1,3),(1,8)\}$. After sorting, it will become $\{(0,0),(6,1),(3,2),(1,3),(0,5),(6,6),(3,7),(1,8)\}$. The overall footprint consists of 7 stripes as indicated in Figure 6.

In each stripe, a horizontal line will intersect with left bounds of some footprints $f(\vec{a},\vec{S})$. Thus, the two-dimensional intersection problem of this stripe in the footprint can be reduced to the one-dimensional problem, which can be solved using Theorem 1. Applying this idea to each stripe, we can solve the two-dimensional overlap problem, as demonstrated in Algorithm 1. The algorithm is obviously a polynomial-time algorithm, whose time complexity is $O(n^2)$.

From Lemma 2, the intersection will keep constant if f_x is greater than the value chosen by this algorithm, and will reduce with less f_x. We can demonstrate this phenomenon by two examples. The set R for the first example is $\{(0,1),(5,3),(-3,1),(4,-1),(-2,-2)\}$ and the partition shape is $(1,0) \times (0,1)$. It is the partition shape for *wave digital filter*. The set R for the second example is $\{(0,2),(3,5),(1,3),(-1,-1)\}$ and the partition shape is $(1,0) \times (-3,1)$. It is the partition shape for *two-dimensional filter*. Figures 7a and 7b show the varying trends of footprint intersection 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 *multidimensional rotation scheduling algorithm* [6] is used to generate a

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

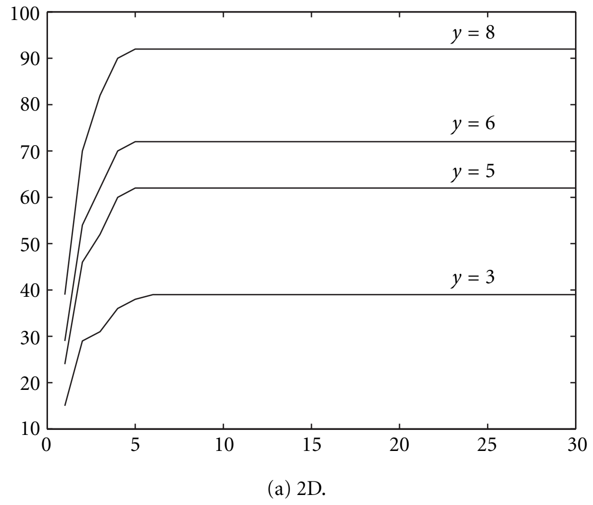
ALGORITHM 1: Calculating the minimum x to make the overlap maximum.

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 in this way is the most compact schedule since it only considers the ALU hardware resource constraints. The overall schedule length must be longer than it. Thus, this ALU schedule provides a lower bound for the overall schedule. This lower bound can be calculated by $\#\text{len}_{\text{iteration}} \times \#\text{nodes}$, where $\text{len}_{\text{iteration}}$ represents the schedule length obtained by multidimensional rotation scheduling algorithm for one iteration, and #nodes denotes the number of iteration nodes in one partition. Our objective is to find a partition whose overall schedule length can be very close to this lower bound.

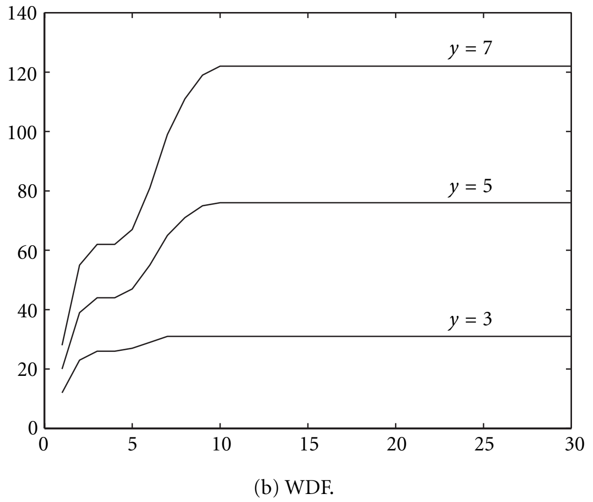
4.1. *Balanced overall schedule*

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

For the intermediate data, the calculation of the number



(a) 2D.



(b) WDF.

Figure 7: The tendency of intersection with f_x and f_y.

of prefetch and keep operations can refer to [13]. For the initial data, they can be prefetched in blocks. This kind of operation can fetch several data at one time and costs only a little longer time than general prefetch operation. To calculate the number of such operations, we first have the following observation.

Property 2. As long as $f_y P_y G_2$, the projection of footprint size along the direction G_2, is larger than the maximum difference of $\vec{a}G_2$, for all $\vec{a}$ belongs to a uniformly generated offset vector set, the overall footprint will increase at a constant rate with the increment of f_y, so does the number of prefetch operations for initial data.

Note the requirement in the above property guarantees that the partition is large enough, such that the footprint with respect to an offset vector can intersect with the footprint with respect to all other offset vectors belonging to the same uniformly generated set.

Suppose that a two-dimensional vector can be written as $\vec{a} = (a \cdot x, a \cdot y)$. Given a certain f_x, the number of prefetch operations for initial data for any f_y, which satisfy the condition in the above property, is $\text{Pre}_{\text{Base\_ini}} + (f_y - f_{y_0}) \times \text{Pre}_{\text{incr\_ini}}$, where $f_{y_0} = \lceil y_0/((P_y G) \cdot y) \rceil$, y_0 is the maximum difference of $(\vec{a}G) \cdot y$ for all offset vectors, $\text{Pre}_{\text{Base\_ini}}$ denotes the number of such operations for a partition with size $f_x \times f_{y_0}$, and $\text{Pre}_{\text{incr\_ini}}$ represents the increment of number of prefetch operations when f_y is increased by one.

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

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

Definition 2. A balanced overall schedule is a schedule for which the memory schedule is at most one unit time of keep operation longer than the ALU schedule.

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

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

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

As long as these constraints on minimal partition size are satisfied, the length of prefetch and keep parts for intermediate data in memory schedule increases slower than the ALU schedule length when partition size is enlarged. At this time, if a partition size cannot be found to meet the balanced overall schedule requirement, it means that the length of the block prefetch part for initial data increases too fast. Due to the property of block prefetch, increasing f_x will increase the number of block prefetch only by a small number, while increase the ALU part by a relative large length. Therefore, a partition size which satisfy the balanced overall schedule requirement can be found. Algorithm 2 determines the partition size to obtain the balanced overall schedule.

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

The memory requirement for a partition consists of four parts, the memory requirement for the calculation of in-partition data, the memory for prefetch operations of intermediate data, the memory for keep operations of intermediate data, and the memory for those operations of initial data. The memory consumption calculation for in-partition data can refer to [9]. For the other part memory requirements,

Table 1: Experimental results with only one initial data.

Benchmark	Par vector		New algo			Partition algo				List		Hardware	
	P_x	P_y	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%

```
Input: The ALU schedule for one iteration, the partition
shape $P_x \times P_y$ and the initial data offset vector set $R$.
Output: A partition size which can generate a balanced
overall schedule.
    (1) Based on the information of initial data, use
        Algorithm 1 to calculate the minimum partition size
        $f'_x$ and $f'_y$.
    (2) Using the two above conditions on partition size,
        calculate another pair of minimum $f''_x$ and $f''_y$.
    (3) Get a new pair $f_x = \max(f'_x, f''_x)$ and
        $f_y = \max(f'_y, f''_y)$.
    (4) Using this pair $(f_x, f_y)$, calculate the number of
        prefetch operations, block prefetch operations, and
        keep operations.
    (5) Calculate the ALU schedule length to see if the
        balanced overall schedule requirement is satisfied.
    (6) If it is satisfied, this pair $(f_x, f_y)$ is the partition size.
        Otherwise, increase $f_x$ by one, use the balanced
        overall schedule requirement to find the minimum
        $f_y$. If such $f_y$ does not exist, continue increasing $f_x$
        until the feasible $f_y$ is found. Use them as partition
        size.
    (7) Based on the partition size, output the
        corresponding ALU part schedule and memory part
        schedule.
```

Algorithm 2: Find a balanced overall schedule.

they can be computed simply by multiplying the number of operations with the memory requirement of each operation. The memory requirement for a prefetch operation is 2. One is used to store the data prefetched by the previous partition and consumed in the current partition, the other stores the data prefetched by the current partition and consumed in the next partition. As the same rule, the keep operation will take 2 memory locations, too. The block prefetch operations will take $2 \times \text{block\_size}$ memory locations.

5. 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 Tables 1 and 2, which stand for *wave digital filter, infinite impulse response filter, differential pulse-code modulation device, two-dimensional filter and Folyd-Steinberg algorithm*, respectively. These are DSP filters in common usage in real DSP applications. We applied five different algorithms on these benchmarks: list scheduling, hardware prefetching scheme, partitioning algorithms in [9, 13] and our new partition algorithm (since it has been shown in [9] that loop tiling technique cannot outperform partitioning algorithms, we do not compare the result of loop tiling in this section). In list scheduling, the same architecture model is used. However, the ALU part uses the traditional list scheduling algorithm, and the iteration space is not partitioned. In hardware prefetching scheduling, we use the model presented in [18]. In this model, whenever a block is accessed, the next block is also loaded. The partitioning algorithms in [9, 13] assume the same architecture model as ours. They partition the iteration space and execute the entire loop along the partition sequence. However, they do not take into account the influence of the initial data.

In the experiment, we assume an ALU computation, a keep operation of one clock cycle, a prefetch time of 10 CPU clock cycles, and a block prefetch time of 16 CPU clock cycles, which is reasonable when the big performance gap between CPU and the main memory is considered. Table 1 presents results with only one initial data with the offset vector $(1,1)$, and Table 2 presents results with three initial data with the offset vector set $\{(1,1),(2,-2),(0,3)\}$. Note all these three initial data references are uniformly generated. From the discussion in Section 4, the overall footprint is only the simple summation of the footprint with respect to different uniformly generated reference sets. In Tables 1 and 2, the *par vector* column determines the partition shape. The *list* column lists the schedule length for list scheduling and the improvement ratio our algorithm can get compared to list scheduling. The *hardware* column lists the schedule length for hardware prefetching and our algorithm's relative improvement ratio. Since the algorithm in [13] will get the same result as the algorithm in [9] when there is no memory size constraint, we merge their results into one column *partition algo*. In the *partition algo* and *new algo* columns, the *size* column is the size of partition presented with the multiple of partition vectors. The *m_r* column represents the corresponding memory requirement and the *len* column is the average scheduling length for corresponding algorithms. The *ratio* column is the improvement our new algorithm can get relative to the corresponding algorithms.

The list scheduling and hardware prefetching schedule the operations based on the iteration, which will result in the

Table 2: Experimental results with three initial data.

Benchmark	Par vector		New algo			Partition algo				List		Hardware	
	V_x	V_y	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%

much longer memory schedule. It is this dominant memory schedule that leads to an overall schedule which is far away from the balanced schedule. Thus, lots of ALU resources are wasted waiting for the data. Their much worse performance compared with the partitioning technique can be seen from the tables.

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

6. CONCLUSION

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

REFERENCES

[1] T. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Trans. Computer Systems*, vol. 16, no. 1, pp. 55–92, 1998.

[2] T.-F. Chen, *Data prefetching for high-performance processors*, Ph.D. thesis, Dept. of Comp. Sci. and Engr., University of Washington, Wash, USA.

[3] F. Dahlgren and M. Dubois, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733–746, 1995.

[4] N. Manjikian, "Combining loop fusion with prefetching on shared-memory multiprocessors," in *Proc. International Conference on Parallel Processing*, pp. 78–82, Bloomingdale, Ill, USA, August 1997.

[5] M. K. Tcheun, H. Yoon, and S. R. Maeng, "An adaptive sequential prefetching scheme in shared-memory multiprocessors," in *Proc. International Conference on Parallel Processing*, pp. 306–313, Bloomington, Ill, USA, August 1997.

[6] N. Passos and E. H.-M. Sha, "Scheduling of uniform multidimensional systems under resource constraints," *IEEE Trans. on VLSI Systems*, vol. 6, no. 4, pp. 719–730, 1998.

[7] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson, "Register requirements of pipelined processors," in *Proc. International Conference on Supercomputing*, pp. 260–271, Washington, DC, USA, July 1992.

[8] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proc. 27th Annual International Symposium on Microarchitecture*, pp. 63–74, San Jose, Calif, USA, November 1994.

[9] Z. Wang, T. W. O'Neil, and E. H.-M. Sha, "Minimizing average schedule length under memory constraints by optimal partitioning and prefetching," *Journal of VLSI Signal Processing*, vol. 27, no. 3, pp. 215–233, 2001.

[10] P. Bouilet, A. Darte, T. Risset, and Y. Robert, "(pen)-ultimate tiling," in *Scalable High-Performance Computing Conference*, pp. 568–576, Knoxville, Tenn, USA, May 1994.

[11] J. Chame and S. Moon, "A tile selection algorithm for data locality and cache interference," in *Proc. 13th ACM International Conference on Supercomputing*, pp. 492–499, Rhodes, Greece, June 1999.

[12] A. Agarwal, D. A. Kranz, and V. Natarajan, "Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 9, pp. 943–962, 1995.

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

[14] V. Van Dongen and P. Quinton, "Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic array," in *International Conference on Systolic Arrays*, pp. 473–482, San Diego, Calif, USA, May 1988.

[15] R. Bixby, K. Kennedy, and U. Kremer, "Automatic data layout using 0-1 integer programming," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, pp. 111–122, Montreal, Canada, August 1994.

[16] G. Rivera and C. W. Tseng, "Eliminating conflict misses for high performance architectures," in *Proc. 1998 AACM International Conference on Supercomputing*, pp. 353–360, Melbourne, Australia, July 1998.

[17] N. L. Passos, E. H.-M. Sha, and S. C. Bass, "Schedule-based multi-dimensional retiming on data flow graphs," *IEEE Trans. Signal Processing*, vol. 44, no. 1, pp. 150–156, 1996.

[18] J. L. Baer and T. F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. Supercomputing '91*, pp. 176–186, Albuquerque, NM, USA, November 1991.

Zhong Wang received a Bachelor's degree in electric engineering in 1994 from Xi'an Jiaotong University, China and a Master's degree in information and signal processing in 1998 from Institute of Acoustics, Academia Sinica, China. Currently, he is pursuing his Ph.D. in computer science and engineering at University of Notre Dame in Indiana. His current research focuses on the loop scheduling and high-level synthesis.

Edwin Hsing-Mean Sha received his B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986; he received the M.S. and Ph.D. degrees from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, Notre Dame, IN. He served as Associate Chairman for Graduate Studies since 1995. He is now a tenured full professor in the Department of Computer Science at the University of Texas at Dallas. He has published more than 140 research papers in refereed conferences and journals. He has been serving as an editor for several journals such as IEEE Transactions on Signal Processing and Journal of VLSI Signal Processing. He also served as program committee member in numerous conferences. He received Oak Ridge Association Junior Faculty Enhancement Award in 1994, and NSF CAREER Award. He was a guest editor for the special issue on Low Power Design of IEEE Transactions on VLSI Systems in 1997. He also served as the program chairs for the International Conference on Parallel and Distributed Computing Systems (PDCS), 2000 and PDCS 2001. He received Teaching award in 1998.

Yuke Wang received his B.S. degree from the University of Science and Technology of China, Hefei, China, in 1989, the M.S. and Ph.D. degrees from the University of Saskatchewan, Canada, in 1992 and 1996, respectively. He has held faculty positions at Concordia University, Canada, and Florida Atlantic University, Florida, USA. Currently he is an Assistant Professor at the Computer Science Department, University of Texas at Dallas. He has also held visiting assistant professor positions in the University of Minnesota, the University of Maryland, and the University of California at Berkeley. Dr. Yuke Wang is currently an Editor of IEEE Transactions on Circuits and Systems, Part II, an Editor of IEEE Transactions on VLSI Systems, an Editor of Applied Signal Processing, and a few other journals. Dr. Wang's research interests include VLSI design of circuits and systems for DSP and communication, computer aided design, and computer architectures. During 1996–2001, he has published about 60 papers among which about 20 papers are in IEEE/ACM Transactions.