Combined Partitioning and Data Padding for Scheduling Multiple Loop Nests *

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 Computer Science
University of Texas at Dallas
Richardson, TX 75083
edsha@utdallas.edu

Xiaobo (Sharon) Hu
Dept of Comp Sci & Engr
University of Notre Dame
Notre Dame, IN 46556
shu@cse.nd.edu

ABSTRACT

With the widening performance gap between processors and main memory, efficient memory accessing behavior is necessary for good program performance. Loop partition is an effective way to exploit the data locality. Traditional loop partition techniques, however, consider only a singleton nested loop. This paper presents multiple loop partition scheduling technique, which combines the loop partition and data padding to generate the detailed partition schedule. The computation and data prefetching are balanced in the partition schedule, such that the long memory latency can be hidden efficiently. Multiple loop partition scheduling explores parallelism among computations, and exploit the data locality between different loop nests as well in each loop nest. Data padding is applied in our technique to eliminate the cache interference, which overcomes the problem of cache conflict misses arisen from loop partition. Therefore, our technique can be applied in architectures with low associativity cache. The experiments show that multiple loop partition scheduling can achieve the significant improvement over the existing methods.

1. INTRODUCTION

Multi-dimensional applications such as image processing, DSP processing, etc, comprise of lots of computation in the form of multi-dimensional loops. These loops access the array data in rather regular patterns. The time cost for loading the data from the memory plays an important role in determining the overall performance. Two methods can be used to reduce the inefficiency caused by the long memory access time: hide the memory access time by prefetching, and enhance the data locality to reduce the number of memory references.

Loop partition technique combines these two aspects to take into consideration of both computation parallelism and memory access latency. It divides the iteration space into regular partitions and provides a detailed schedule which is easily adopted by a compiler. It has been shown that loop partition technique [3, 15] is an effective method to exploit the data locality in a singleton nested loop. Nevertheless, it is usual that a program includes several loop nests in the consecutive order. A lot of computational overhead is caused by the repetitive access to array data elements. Separately partitioning each loop will not take full advantage of the data reuse occurred among different loop nests, thereby incur much more memory references than multiple loop partition. This is demonstrated by the experiments in Section 6. Therefore, the traditional loop partition technique is not efficient for multiple loop nests. Multiple loop partition technique deals with multiple loop nests together and partitions them at the same time, such that the data locality can be exploited to the largest extent.

Our technique can be used in the architecture model with multiple processors or a processor with multiple function units. They share a memory hierarchy. We assume two levels memory in this paper, in which the first and second levels memory can be regarded as the on-chip and off-chip memories, respectively. The model consists of multiple ALU and memory units. The ALU units are for computation, while the memory units are used to prefetch the data into the first level memory. The partition schedule includes the memory schedule and ALU schedule, corresponding to arranging the prefetch and ALU operations in the respective units. This model can be found in DSP processors, embedded system and shared-memory multiprocessors computer systems. For example, separated ALU and memory units exist in TMS320C64x.

Much work has been done on loop fusion and loop tiling. Loop fusion is used to fuse the consecutive loops into a single loop to exploit the data locality and reduce the additional synchronization. Kennedy and McKinley [5] perform loop fusion to combine a collection of loops. Naraig and Tarek [6] suggest a "shift-and-peel" loop transformation to fuse loops. However, loop fusion can only exploit the data locality among operations in the same index iterations. In contrast, multiple loop partition can exploit the data locality exists among not only different loop nests but also different iterations. In fact, loop fusion can be thought of as a special case of multiple loop partition when the partition size is 1.

Loop tiling is a technique for grouping elemental computation points so as to increase computation granularity and

*This work is supported in part by NSF under grant number MIP-9701416.

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

thereby reduce communication time. Boulet *et al* [10] introduce a criterion for defining optimal tiling in a scalable environment. Sarkar *et al* [13] propose an analytical model for estimating the memory cost and a constant-time algorithm for finding an optimal solution to this model. However, the traditional loop tiling only considers the singleton loop and lacks the consideration of prefetching and scheduling. On the other hand, grouping computation in loop tiling will introduce more loop index variables and increase the loop depth, thereby incur more control overhead. The small tile size is not applicable, since overhead due to tiling cannot be compensated by the benefit from tiling [4].

An important problem related to grouping computation in the loop nest is that much more cache interference may be incurred. Chame and Moon [4] propose a new tile selection algorithm for eliminating self interference and simultaneously minimizing capacity and cross-interference misses. Data padding, which changes the array store sequence in memory, is an effective technique to address the cache interference problem. Rivera and Tseng [12] develop and evaluate heuristics for guiding two data-layout transformations, inter- and intra-variable padding. Data padding technique is utilized in multiple loop partition to eliminate the cache conflict when the first level memory is a direct mapped cache.

In this paper, we present multiple loop partition scheduling technique which combines loop partitioning and data padding. The detailed schedules of ALU and memory units are presented. Compiler can directly map this schedule to operations on the corresponding function units. In such case, we can avoid the overhead incurred by the partition as well as possess the benefit of the small partition size. With the small size partition, the task to determine a data padding pattern without cache conflict becomes much easier. The partition size is carefully selected such that the ALU and memory schedules are balanced and the memory latency is tolerated effectively. According to the author's knowledge, no previous research has been done to partition multiple loop nests together. Multiple loop partitioning scheduling technqiue also extends previous partitioning research by considering low-associativity cache. The experiments show that this technique improves the traditional methods by about 15% to 45% for different benchmarks.

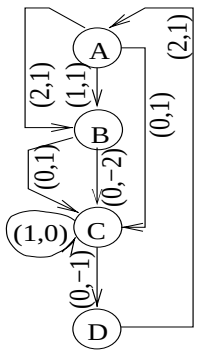
The rest of the paper is organized as follows. In Section 2, the program models are introduced. Section 3 presents the concept of partition. Section 4 describes multiple loop partition scheduling in detail. Section 5 discusses how to determine the pad size to eliminate the cache conflict. Section 6 gives the experimental results.

2. PROGRAM MODEL

Multiple loop nests are represented by *Loop Dependence Graph* (LDG), as defined below.

DEFINITION 1. *A loop dependence graph (LDG)* $G = (V, E, D_L)$ *is an edge-weighted directed multigraph, where* V *is the set of loop nests,* $E \in V \times V$ *is the set of dependence edges between the loop nests, and* D_L *is a function from* E *to* Z^n*, representing the set of loop dependency vectors between two nodes, where* n *is the number of loop dimension.*

The LDG is a directed multigraph because multiple dependences can exist from one loop nest to another arising from different variables. Figure 1(a) shows an example of LDG and Figure 1(b) shows the corresponding code segment. In this example, $V = \{A, B, C, D\}$, weights are labeled on the corresponding edges between nodes. and $E = \{e_1, e_2 : (A, B), e_3, e_4 : (B, C), e_5 : (C, C), e_6 : (C, D), e_7 : (A, C), e_8 : (D, A)\}$ where, $D_L(e_1) = (2, 1)$, $D_L(e_2) = (1, 1)$, $D_L(e_3) = (0, 1)$, $D_L(e_4) = (0, -2)$, $D_L(e_5) = (1, 0)$, $D_L(e_6) = (0, -1)$, $D_L(e_7) = (0, 1)$, $D_L(e_8) = (2, 1)$. Note that multiple edges exist between nodes A, B and B, C, due to the different array elements access.



(a) LDG

```
      DO   10   i = 0, n
A:    DO   20   j=0, m
          a[i][j] =        e[i − 2][j − 1]
20        CONTINUE
B:    DO   30   j= 0,m
          b[i][j] = a[i − 1][j − 1]
                    +a[i − 2][j − 1]
30        CONTINUE
C:    DO   40   j= 0,m
          c[i][j] = b[i][j + 2] − a[i][j − 1]
                    +b[i][j − 1]
          d[i][j] = c[i − 1][j]
40        CONTINUE
D:    DO   50   j= 0,m
          e[i][j] = c[i][j + 1]
50        CONTINUE
10    CONTINUE
```

(b) Program segment

Figure 1: The code and its LDG

This paper concentrates on the loops with uniform data dependence. A lot of applications in DSP and image processing fit in this category. Also the importance can be justified by the fact that affine index access can be uniformized [11, 14] to apply our technique. We are dealing with multiple dimensional nested loops in this paper. For convenience of presentation, the algorithms are illustrated with multiple two-dimensional nested loops. The idea can be applied to the higher dimension without difficulty. Under two dimension, the weight of edges in the corresponding LDG can be written as $D_L = (D_L[1], D_L[2])$. The two different target program models are shown in the Figures 2(a) and 2(b), respectively.

Both two models include several uniform loop nests without the intervening code between them. All the loop nests are *conformable* with each other. Conformability is a binary equivalence relation. Two loop nests are said to be conformable if their corresponding loops have the same type (serial in this paper) and identical iteration space size (loop bounds). The data dependences between loop nests are constant, as well as lexically forward (from the earlier loop to the later). Program transformations may be used to obtain a program segment that belongs to the above models. For instance, code motion [1] may be employed to obtain a sequence of loops with no intervening code. Loop permutation [2] may be used to ensures that loop nests are conformable.

The difference between two models is that multiple sequential loops have the same outer-most loop in the nested multiple loops program model, while the loops in the multiple nested loops model are realatively independent. Thus,

```
      DO   K1   i = 0, n
           DO   K2   j=0, m
           ...
K2         CONTINUE

           DO   K3   j= 0,m
           ...
K3         CONTINUE

           .
           .
           .

K1    CONTINUE
```

(a) Nested Multiple Loops

```
      DO   K1   i = 0, n
           DO   K2   j=0, m
           ...
K2         CONTINUE
K1    CONTINUE
           DO   K3   j= 0,m
                DO   K4   j=0, m
                ...
           K4   CONTINUE
           K3   CONTINUE
                .
                .
                .
```

(b) Multiple Nested Loops

Figure 2: Program Model

there will be some loop-carried data dependence introduced by the outer-most loop in the first model, which may cause a dependence cycle in the corresponding LDG. To guarantee the correctness of the execution order, there should not exist the dependence that flows backwards with respect to the iteration execution order. A set of nested multiple loops is illegal if no execution order exists to sastify all the data dependence. The following lemma ensure the legality in the first program model.

LEMMA 1. *A set of nested multiple loops is legal if each cycle in its corresponding LDG sastify* $W(c) \geq (1, -\infty)$ [1] *and* $D_L(e)[1] >= 0, \forall e \in E$, *where* $W(c)$ *is the weight summation for all edges in a cycle.*

On the contrary, there is no data dependence from a loop nest to its ancester in the second program model. Its LDG is always acyclic. The existence of cycle in LDG brings a little more complication on the in-partition execution order, which is discussed in the next section.

3. CONCEPT OF PARTITION SCHEDULING

Partitioning is an effective technique to improve the data locality. Due to the general short data dependence relative to the iteration space size, grouping several iteration executions can increase both the cache and register locality. Partitioning multiple loop nests is built upon partitioning an individual loop nest, taking the data dependence between loop nests into consideration. It is necessary to review the partitioning technique for an individual loop nest.

An individual nested loop can be described by a *multi-dimensional data flow graph (MDFG)* [8]. Each node in the MDFG corresponds to a computation. An edge between two nodes represents that a data dependence exists between these two nodes. Its weight (d_e) is the data dependence vector. A partition is a parallelogram in the iteration space. All iterations with integer index vectors which lie in this parallelogram belong to the same partition. A partition is characterized by its shape and size. Its shape is delimited by two direction vectors: P_x and P_y, where P_y is anti-clockwise to P_x. The following property [15] guarantees that all data dependence vectors lie between P_x and P_y, such that data dependences can be maintained after partitioning.

PROPERTY 1. *It is a legal partition shape if and only if the cross products* $d_e \times P_x \leq 0, \quad d_e \times P_y \geq 0, \quad \forall$ *data dependence* d_e *in MDFG.*

A partition schedule [3] is made up of the ALU schedule, which schedules the ALU operations, and memory schedule, which schedules the prefetching operations. The ALU schedule is formed by Multi-dimensional Rotation Scheduling Algorithm [9]. In fact, any loop pipelining scheme can be used to schedule the ALU operation. The reason we use this algorithm is that it is proven to be optimal [9]. The memory schedule is formed by issuing all the prefetching operations for the entire partition. A partition size is determined to make the ALU and memory schedules *balanced*, such that the large memory access latency can be tolerated by the ALU operations. A *balanced partition schedule* is defined to be a partition schedule in which the memory schedule is shorter than the ALU schedule.

Another important aspect of partitioning technique is the partition execution order. Instead of the general row-wise order along X axis or column-wise order along Y axis in the unit of iterations, the loop nest is executed in the unit of partitions in the row-wise manner: execute all the first line of partitions along X, then the second line, the third line, etc.

In the case of partitioning multiple loop nests, a partition consists of several *subpartitions*. Figure 3 plots an example of a single partition. In a partition, all the iterations which belong to the same iteration space constitute a subpartition. In this figure, L_1, L_2 and L_3 are the iteration spaces for the first, second and third loop nests, respectively. SP_1, SP_2 and SP_3 are the subpartitions in L_1, L_2 and L_3, respectively.

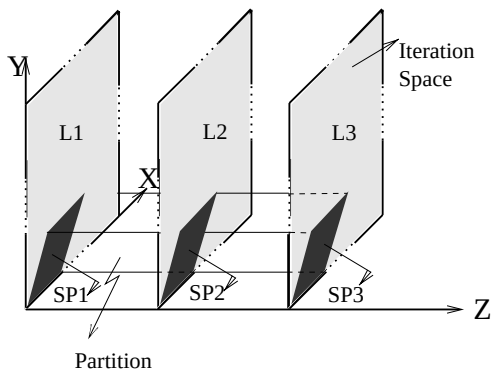


Figure 3: Partition and subpartitions

Multiple loop partition exploits not only the data locality inside each iteration space, but also the data locality among different loop nests. Taking Figure 4 as an example, $SP1$ and $SP2$ represent the subpartitions in iteration space L1 and L2, respectively. The dark circles denote the iterations. Iterations $I1$, $I2$, and $I3$ lie in $SP1$, while iterations $I4$ and $I5$ lie in $SP2$. Therefore, data dependences $d1$ and $d2$ belong to the dependences inside a single loop nest, $d3$ and $d4$ belong to the dependence between different loop nests.

[1] In this paper, all the comparsion between two vectors is based on the lexicographic order

Assuming iterations $I2$ and $I3$ are in different rows, so are iterations $I4$ and $I5$. They all need the outcome from the computation of iteration $I1$. According to the general row-wise order, the result of iteration $I1$ will be fetched from the memory for 4 times. On the contrary, this result is still in the first level memory and no fetch is needed with the multiple loop partition technique.

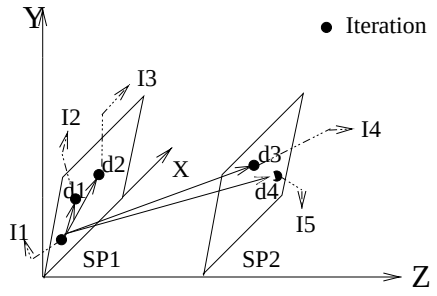
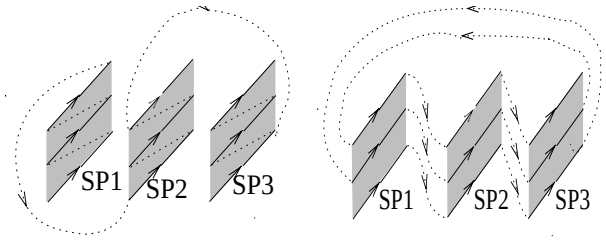


Figure 4: Different data dependences

With the row-wise partition execution order, differently grained in-partition execution order can be applied. The coarse-grained in-partition execution order is shown in Fig 5(a). The first subpartition is finished first, then the second subpartition, until the last subpartition's execution ends. On the other hand, we can execute the iterations alternatively in the loop nests. Figure 5(b) shows this fine-grained in-partition execution order. After the execution of the first line of iterations in each subpartition along loop sequence, we jump to the first subpartition and begin to execute the second line. A partition is finished until all the lines of iterations are executed in this manner.



(a) Coarse-grained (b) Fine-grained

Figure 5: In-partition execution order

The first principle of the selection of in-partition execution order is that such order cannot violate the data dependence. For the multiple loop nests program model, both orders are feasible because of the acyclic lexically forward dependence. This is not the case for the nested multiple loops program model. Some data dependence cannot be satisfied by the coarse-grained execution order due to the cycle in LDG.

There are also some other considerations on the in-partition execution order. For example, different efficiency can be achieved for the different data access patterns in the loops. The coarse-grained order performs better when each loop nest has its own input data set, while fine-grained order is preferred when the loop nests have a common input data set. In such case, some compiler switches can be used to designate a better in-partition execution order.

4. MULTIPLE LOOP PARTITION SCHEDULING

Multi-dimensional retiming technique [8] is used in multiple loop partition to transform the iteration space, such that the iteration space can be partitioned legally. A *multi-dimensional retiming* r is a function that redistributes the nodes in the iteration space. A new LDG is created after the retiming. The retiming vector $r(u)$ of a node $u \in V$ represents the offset between the original iteration containing u, and the one after retiming. The weight of the edge changes accordingly to preserve dependence, i.e., $D_L^r(u,v) = D_L(u,v) + r(u) - r(v)$, where D_L^r and D_L is the weight after and before retiming, respectively. Also, $D_L^r(c) = D_L^r(c)$ for each cycle $c \in LDG$.

In the nested multiple loops program model, it is stated in Lemma 1 that $D_L[1]$ element of the weight in LDG should be no less than zero for any edge in LDG. Provided the fine-grained execution order is used, there are two cases for the data dependences. If $D_L[1] > 0$ or $D_L[1] = 0, D_L[2] \geq 0$, these data dependences can be preserved using the fine-grained execution order. If $D_L[1] = 0, D_L[2] < 0$, it is reverse to the flow of execution along X axis and will be violated after the partition. Therefore, we should eliminate all the data dependence that belong to the second case. This can be accomplished by the retiming technique [8] proved by the following theorem.

THEOREM 2. *Given a legal LDG, there exists a set of retiming vector $S(\vec{r})$ such that a legal retimed LDG is obtained with $d_l \geq (0,0)$ for any edge in this graph.*

PROOF. Assuming an edge $e : u \rightarrow v$, its weight after retiming is $D_L^r = D_L + r(u) - r(v)$, The inequality $r(v) - r(u) \leq D_L$ must be true to make $D_L^r \geq (0,0)$ Therefore, a feasible solution of the following integer programming problem can serve as a retiming vector set, where r_n denote the retiming vector on n^{th} node.

ILP:
$$\begin{aligned} & r_1 - r_2 \leq D_L(1,2) \\ & r_2 - r_3 \leq D_L(2,3) \\ & \vdots \\ & r_{n-1} - r_n \leq D_L(n,n-1) \end{aligned}$$

This specical IP problem can be solved using graph theory. A directed graph can be constructed with each node denoting a variable, and edges between the corresponding nodes whose weight equal to D_L. An additional source node is also added. This node has a zero weight edge to any node in the original graph. If there is no cycle with weight less than $(0,0)$, The single source shortest path is a feasible solution. Lemma 1 has shown that for all cycle in the graph, $W \geq (1, -\infty)$, which ensure the existence of a feasible solution.

□

A retiming vector set can be obtained using Bellman-Ford single source shortest path algorithm. The only modification

is to replace all the weights, real numbers in the original algorithm, by weight vectors i.e., D_L in the LDG.

In the multiple loop nests program model, there is no certainty that $D_L[1] \geq 0$. The data dependence with $D_L[1] < 0$ will also be voilated after the partition. This voilation can still be overcome by retiming. Because the LDG for this program model is acyclic, the following algorithm can be used to get rid of all of the violiations.

Algorithm 1 Find a feasible retiming vector set

Input: An LDG graph
Output: A retiming vector set of LDG
  Sort the nodes in topological order
  FOREACH Node i in LDG **do**
    $r(i) = (0, 0)$
  ENDFOR
  FOREACH Node i in LDG **do**
    Construct a vector $NV = (NV[1], NV[2])$, which sastisfy $NV[1] \leq D_L[1], NV[2] \leq D_L[2]$, $\forall$ incoming edges with weight D_L.
    IF $NV \geq 0$ **THEN** $r(i) = (0, 0)$ **ELSE** $r(i) = NV$ **ENDIF**
    Update all the outgoing edges' weigt to $D_L^r = D_L + r(i)$.
  ENDFOR

After applying the retiming vectors obtained above, all the data dependence sastify $D_L \geq (0, 0)$, thereby are automatically preserved by the partition execution order. The partition shape is only determined by the data dependence d_e in the MDFG of each loop nest. Therefore, we can find a legal subpartition shape for each loop based on Property 1. Due to the lexically forward data dependence, vector $(1, 0)$ can always serve as P_x for each subpartition, thereby the P_x of the partition. As about P_y, it is one of P_y of all the subpartitions, which has the largest angle with vector $(1, 0)$. It is not difficult to verify that all the data dependences conform to Property 1 under such partition shape.

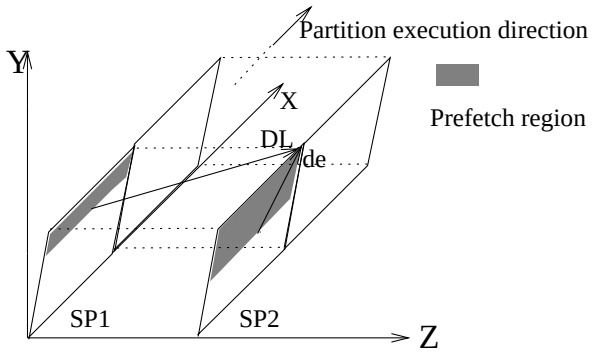


Figure 6: Prefetch regions

The determination of the partition size depends on the relation between the memory and ALU schedules. As indicated before, a partition size which can balance the ALU and memory schedules is a preferred size.

To derive the memory scheudle, all the prefetching operations need to be identified. The number of prefetching operations depends on the data dependence distance and the partition size. A certain data dependence decide a perfetching region, in which all data need to be prefetched in advance. Figure 6 shows two adjacent partitions along the partittion execution direction and two different data dependences. DL is a data dependence between two loop nests, and de is a dependence inside a loop. Each of them deicde a prefetch region in the partition, as shown in the figure.

Algorithm 2 Multiple loop partition scheduling

Input: The data dependences d (including both d_e in MDFG and D_L in LDG) in multiple loop nests
Output: The partition size
  /*find the partition shape*/
  For each loop nest, find the subpartition shape P_{x_s}, P_{y_s}
  Set $P_x = (1, 0)$. Choose P_y as the most anti-clockwise vector relative to P_x in all P_{y_s}.
  /*find the minimum f_x and f_y*/
  $f_x = 0, f_y = 0$.
  FOREACH $d = (d[1], d[2])$ /* vector d can be regarded as d_e or D_L */ **do**
    if $fx < (d[1] - d[2] * p_y[1]/p_y[2])$ **then**
      $fx = \lceil d[1] - d[2] * p_y[1]/p_y[2] \rceil$
    end if
    if $fy * P_y[2] < d[2]$ **then**
      $f_y = \lceil d[2]/P_y[2] \rceil$
    end if
  ENDFOR
  /*determine the partition size*/
  Calculate L_{mem} and L_{ALU} under the partition size f_x, f_y.
  icount = 0
  WHILE $L_{ALU} \geq L_{MEM}$ **do**
    f_y++;
    icount ++;
    Calculate L_{mem} and L_{ALU} under the partition size f_x, f_y.
    if icount $\geq$ MI **then**
      fx++;
    end if
  ENDWHILE
  Schedule the ALU and memory operations in ALU and memory units, respectively.

In the algorithm 2 to decide the partition size, a partition size is denoted by f_x and f_y. Then the actual partition size is f_x along X axis and $\|f_y \times P_y\|$ along the P_y direction. The lengths of memory and ALU schedules are denoted by L_{mem} and L_{ALU} respectively. The first "for" loop in the algorithm calculates the minimum partition size requirement, which means that no data dependence will span two or more partitions. The constraint is used to reduce the computation and simplify the analysis. The "while" loop in the algorithm calculates the partition size which can achieve a balanced partition schedule. These steps are based on the theory: increasing the partition size along P_y direction can maintain the number of prefetching operation. If the memory schedule is longer than ALU schedule at the beginning, enlarging the partition along P_y direction only increases the number of iterations in the partition, which implies the ALU schedule will increase faster than the memory schedule if the partition size along P_x direction is large enough, thereby guarantees the existence of the balanced partition schedule and the termination of the algorithm. In the algorithm, we use the variable "icount" and a predefined constant "MI" to ensure the f_x is large enough to find a balanced partition schedule.

Another property of a balance partition schedule can be

seen from the algorithm. From the partition size found by the algorithm, decreasing the partition size will lead to the longer memory schedule than the ALU schedule. On the other hand, increasing the partition size furthermore can make the difference between the memory and ALU schedules larger. Therefore, the partition size determined by the algorithm is a best balance point between the memory and ALU schedules. We prefer a small partition size for the convenience of data padding, which is discussed in the next section.

5. DATA PADDING

In the above discussion, the first level memory is always assumed full-associative. In the case of low-associativity cache, some data with different memory addresses may map to the same cache location. The references to these data will lead to conflict misses. Data padding is an effective technique to eliminate such cache interference. It can be classified as inter- and intra- variable padding. Inter-variable padding is applied to several variables. Some gaps are inserted between different variables to eliminate the cross interference among these variables. On the other hand, intra-variable padding is applied to a single variable so as to eliminate the self-interference.

Figure 7 is the example of inter- and intra- variable padding. In Figure 7(a), unit-stride references to $A(i)$ and $B(i)$ provide spatial locality, leading to cache reuse. However, if A and B are separated by a multiple of the cache size and the cache is direct mapped, every access to $B(i)$ will map to the same cache line as the previous $A(i)$ access and vice versa. As a result, every reference will produce a conflict miss. To regain the benefits of spatial reuse, inter-variable padding may be applied as shown in the figure. In Figure 7(b), the computation exhibits much spatial and temporal reuse between the references to A. However, if the column size of A is a multiple of the cache size, columns of A will conflict, eliminating reuse between references to A. To regain spatial reuse, intra-variable padding applied to the columns of A can change its internal layout so that columns no longer map to conflicting cache locations.

real S, $A(N)$, $B(N)$ $->$ real $A(N)$, $DUM(PAD)$, $B(N)$
do i = 1, N
  S = S + A(i) * B(i)

(a) Inter-variable Padding (Change base addr)

real S, $A(N,N)$, $B(N,N)$ $->$ real $A(N+PAD,N)$, B(N,N)
do i = 1, N
do j = 2, N-1
  B(j,i) = (A(j-1,i) + A(j, i-1) + A(j+1, i) + A(j, i+1))/4

(b) Intra-variable Padding (Change Dim. Size)

Figure 7: Data padding illustration

To apply data padding, the overall data set which is kept in the cache during the time of a partition's execution should be determined. It comprises of the prefetched data, which are used in the current partition and prepared during the time of last partition's execution, and the prefetching data, which are prepared in the cache for the next partition's execution. Because Multiple data dependences may exist on a single array, the data set for this array is the union of several parallelograms which are decided by all the data dependences. The overall data set is the aggregation of the data sets for all arrays. When data are loaded into cache, each row in data set occupies a block of consecutive cache addresses. Due to the uniform data dependence, the data set for the next partition is just a constant shift in cache address of that for the current partition. Thus, we can consider any partition and apply data padding to prevent conflict during this partition execution time.

With direct mapped cache, conflict miss may occur due to the fact that different elements may map to the same cache location. Our objective is to eliminate all the cache conflict misses. Inter- and intra- variable data padding can be combined to accomplish this goal. Intra-variable padding is applied first to distribute the data set of an array in the cache, such that the space between any two cache blocks of data is larger than a certain value. We call this certain value as the safe distance and the requirement that the space must be larger than the safe distance as safe distance constraint. The safe distance is large enough to accommodate all array data from the rows in the overall data set with the same row numbers. The data layout in the cache after data padding is shown in Figure 8. Suppose that a data array has the data set DS, whose height (HL)[2] is the largest. Then the cache is partitioned into HL parts. Each row from DS (light shaded rectangle in the figure) lies at the head of each part. For other data sets, i^{th} rows lie in the i^{th} part. As long as the safe distance constraint is satisfied, the i^{th} part is large enough to contain all such data. For example, if we have three data arrays E, F and G, which all have 4×8 rectangle data sets. Assume the cache size is 512, cache line size is 4. then the safe distance is 36 according to Algorithm 3. If the array row size is 256. their cache data layout of one partition is shown in Figure 9. In the figure, each rectangle is labeled by an array name, followed by a row number. It represents the cache positions occupied by this row. The number beneath the axis denotes the corresponding cache position.

It is not necessary to calculate the pad size for each array to satisfy the safe distance constraint. The workload can be reduced due to the uniform data dependence. If a pad size can satisfy the safe distance constraint for the array with the largest height, it is obvious that this pad size can be applied on other arrays without violating the safe distance constraint.

In Algorithm 3, the safe distance is calculated in the first three steps. It is the maximal possible cache lines occupied by the data from all rows in the overall data setwith the same row numbers. Note that the cache distance in this paper is a circular cache distance. There are two cache distance value between any two blocks. We adopt the one with the smaller absolute value as the cache distance. For instance, if the cache size is 512, the cache distance between 1 and 510 is 3, instead of 509. In the data set, the start position of a row is the column number of its leading element. Each row

[2]The height of a date set is the number of rows in this set
[3]For a row of β elements, the function $MaxC(\beta)$ is defined to be the maximal possible number of cache lines to include this row of data.

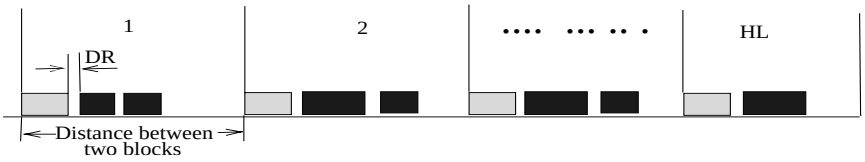


Figure 8: Data layout in the cache

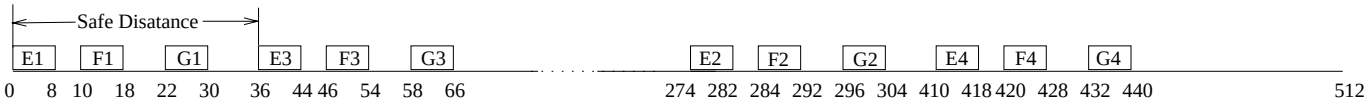


Figure 9: An example of data layout in cache

Algorithm 3 Calculate the intra-pad size p

Input: The partition size and shape, data dependences
Output: The intra-pad size

1, For each data array, determine the data set according to the partition and data dependences. Find the array with the largest height data set. Calculate its data set size and start position of each row of the data set. /*SP_k is used to denote the start position of i^{th} row and $DSize$ is used to represent the size of a data set*/
2, In all the data sets, find all the rows which have the same row number. Suppose the number of such rows is n, and each one has size s_i, calculate the summation S= $\sum_{i=1}^{n} MaxC(s_i)^3$.
3, Scan through all possible rows, choose the maximum S as the safe distance DIS_{safe}.
if $n \times DIS_{safe} + DSize > CS$ **then**
  quit /*no padding pattern can be found*/
end if
/*In the following, A, B are used to record the minimum and maximum difference of start positions.*/
FOREACH $m = 1, n-1$ **do**
  $A[m] = Min(SP_k - SP_{k-m})\ \ \forall k \in [2, n]$;
  $B[m] = Max(SP_k - SP_{k-m})\ \ \forall k \in [2, n]$;
  Calculate $d[m]$ as the distance between cache locations of the first and m^{th} rows of data .
ENDFOR
FOREACH $i \in [1, n]$ **do**
  if $d[i]+A[i] < DIS_{safe}$ *or* $d[i]+B[i] < DIS_{safe}$ **then**
    p= $MAX((DIS_{safe}-A[i]-d[i])/i, (DIS_{safe}-B[i]-d[i])/i)$.
  end if
  /* The following loop eliminates the possible new violation of safe distance constraint introduced by the above pad*/
  FOREACH $m \in [1, i]$ **do**
    if $d[m] + A[m] + m \times p < DIS_{safe}$, then $p = (DIS_{safe} - A[m] - d[m])/m$.
    if $d[m] + B[m] + m \times p < DIS_{safe}$, then $p = (DIS_{safe} - B[m] - d[m])/m$.
  ENDFOR
ENDFOR

may have a different start position. Suppose all the start positions are same, i.e., all rows start at the same column, the following observation is easily deduced.

PROPERTY 2. *Assume the cache distance between n^{th} and m^{th} rows of data set is α and the height of the data set is H. Then for any two rows l^{th} and k^{th}, where $l, k \in [1, H]$, their distance equal to α as long as $|l-k| = |n-m|$.*

Thus, we can simply ensure that the first row of data set has the enough cache distance with all other rows to maintain the safe distance constraints. If the start positions are different, the safe distance constraint can still be satisfied after the difference of start positions are considered. In the algorithm, without loss of generality, the first row of data set is put at the beginning of the cache. The cache size is CS. The time complexity of the algorithm is $O(H^2)$.

Inter-variable padding can be used to eliminate the cross-interference between different arrays. The pad size should be selected such that no two arrays conflict in the cache. We use the concept *adjacent data arrays* to derive the pad size. Two data arrays are said to be adjacent if their corresponding data set satisfy: two rows, with the same row number, lie side by side, as shown in Figure 8. DR means the space between the two cache blocks, which come from the different data sets but have the same row number. When $DR \leq 2 \times C$, where C is cache line size, they are said to lie side by side. Owing to the safe distance constraint, the space between two blocks from the same arrays is large enough to accommodate all the blocks from other arrays. Therefore, The inter-pad size can be determined as long as these two arrays are adjacent.

6. EXPERIMENTAL RESULTS

In this section, the effectiveness of multiple loop partition scheduling technique is evaluated by running a set of simulations on four benchmarks. *Example* is the example code shown in Figure 1(b). *Jacobi* is a PDE solver. *LL18* is the eighteenth kernel from the Livermore Loops benchmark. *Tomcatv* is extracted from the 101.tomcatv in SPEC95 benchmark. We compared the *AET/Iter* by applying three different schemes on these benchmarks. *AET/Iter* (Average execution time per iteration) is the summation of all the average execution times for one iteration in each loop nest. Because of the comfortability of the loop nests, AET/Iter reflects the overall effectiveness of each scheme.

	#L	Original	Fusion	MLP	size	L_{bound}
Example	4	8.5	5.41	**2.31**	8×8	2
Jacobi	2	5.25	4.0	**2.14**	4×8	2
LL18	3	20.85	13.44	**11.5**	4×4	11
Tomcatv	5	25.25	20.38	**18.21**	4×4	18

Table 1: Experiment result of average execution time

In the simulation environment, we have 4 ALU and 4 memory units. The first level memory is 32KB in size and full associative. Each data is float type which consumes 8 bytes. The computation time for each operation is simplified as one unit time. The data are prefetched in block with size 64 bytes. This operation takes 10 units. The assumption is reasonable considering the big gap between the CPU and memory speeds.

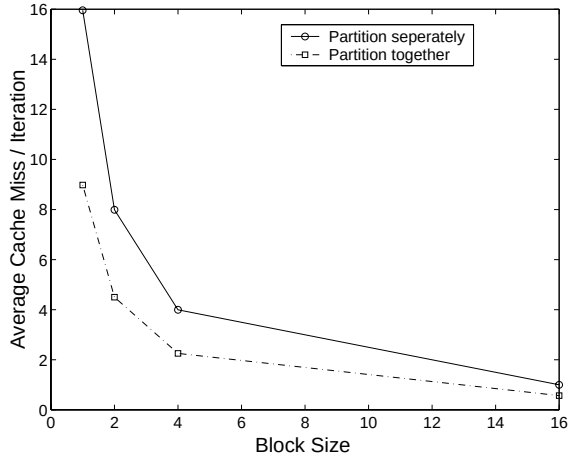


Figure 10: LL18 Kernel

We first compare multiple loop partition scheduling and the traditional partitioning technique, i.e., partition each loop nest separately. The former is superior to the latter, due to the fact that the better data locality is exploited and the repetitive array element accesses are reduced. Figures 6 shows how *average cache miss/ Iteration* changes with the cache block size (denoted by the number of float data in a block) for LL18 kernel under the above simulation environment. We can see that the number of average cache miss for the traditional partitioning technique is always two times the number of our technique. This observation can be explained by the number of array references per iteration. Partitioning separately has two times memory reference as many as multiple loop partition.

The simulation results are listed in Table 1. Three methods compared are *Original*, which is the original loop without any transformation, *Fusion*, in which we apply the loop fusion on the benchmarks, and *MLP*, which is multiple loop partition. Because it has been shown in [15] that software pipelining and loop tiling cannot outperform the traditional partitioning technique, we didn't compare their results. The column *#L* denotes the number of loop nests in each benchmark. The *size* column is the partition size. We also list the theoretic lower bound in the L_{bound} column. As mentioned in Section 3, the ALU schedule is obtained by Multi-dimensional Rotation Scheduling Algorithm [9]without the consideration of the memory load latency. The ALU schedule length per iteration provides the lower bound for computation under the resource constraints. Comparing the AET/Iter, we can find that partitioning technique can achieve the best performance. Accredit to increasing the data locality appeared between different loop nests, the loop fusion perform better than the original loop. The multiple loop partitioning not only takes advantage of the benefit of loop fusion, but also explores more data reuses which exist in the different iterations. Moreover, it takes the balance of computation and memory reference into consideration, thereby prevents the wasted time caused by the dominant memory reference time. Therefore, the multiple loop partition scheduling technique can efficiently exploit the data locality.

The performance of multiple loop partition scheduling is very close to the lower bound, which demonstrates that the memory latency is hidden very efficiently. Their difference is due to the iterations lie on the boundary of iteration space. These iterations can be regarded as the preface, which is the preparation stage of the normal partition execution.

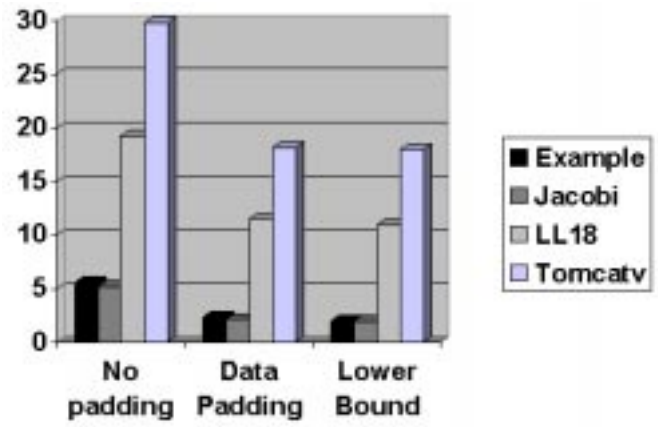


Figure 11: Experiment result of data padding

If the first level memory is direct mapped cache, the average execution time will become larger due to the conflict misses. However, the same result as Table 1 can be achieved after data padding is applied, which is illustrated in Figure 11. In this figure, *No padding* denotes the result without data padding, while *Data Padding* shows the result after data padding. *Lower bound* has the same meaning as above.

In tomcatv benchmark, there are five loop nests with the same iteration space size. However, the first three and the last two loop nests should be thought of as in two indepen-

dent groups, since the step direction is contrary in different groups. The first group is in the form as $do\ 10\ i = 1, N, 1$, while the second group is in the form of $do\ 10\ i = N, 1, -1$. Because loop fusion can be regarded as a special case of multiple loop partition when the partition size is 1, The problem to group loop nests is essentially the same as the fusible loop problem [7].

7. CONCLUSION

In this paper, a new scheme which combines loop partitioning and data padding was proposed. The new algorithms consider the case with multiple consecutive loop nests and partition them at the same time, such that the data locality can be exploited to the largest extent. Moreover, a good balance detailed partition schedule is found so as to efficiently hide the memory latency. In order to deal with low-associativity cache, data padding is utilized to extend the application of our technique. Experiments show a significant improvement over the existed approaches.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical report, Computer Science Division, University of California, Berkeley, 1993.

[3] 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.

[4] 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.

[5] K. Kennedy and K.S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing-Sixth International Workshop*, pages 301–320, Springer-Verlag, Berlin, 1994.

[6] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2), Feb 1997.

[7] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of 9th ACM symposium on Parallel algorithms and architectures*, pages 282–291, Newport, RI, June 1997.

[8] N.Passos and E.H.-M.Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11), November 1996.

[9] 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.

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

[11] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989.

[12] G. Rivera and C. W. Tseng. Data transformation for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[13] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *IEEE Inter Symposium on Performance Analysis of Systems and Software*, pages 146–153, April 2000.

[14] W. Shang, E. Hodzic, and Z. Chen. On uniformization of affine dependence algorithms. *IEEE Transactions on Computers*, 45(7), 1996.

[15] 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.