Scheduling and Partitioning for Multiple Loop Nests[*]

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

Qingfeng Zhuge
Department of Comp Sci
University of Texas at Dallas
Richardson, TX 75083
qfzhuge@utdallas.edu

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

ABSTRACT

This paper presents the multiple loop partition scheduling technique, which combines the loop partition and prefetching. It can exploit the data locality better than the traditional loop partition, which only focus on a singleton nested loop, and loop fusion. Moreover, multiple loop partition scheduling balances the computation and memory loading, such that the long memory latency can be hidden effectively. The experiments shows that multiple loop partition scheduling can achieve the significant improvement over the existed methods.

1. INTRODUCTION

Multi-dimensional applications on 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. To reduce the inefficiency caused by the long memory access time, two methods can be applied: use prefetching to hide the memory access time by computation, and enhance the data locality to reduce the number of memory references. In this paper, we propose a new technique base on loop partition. This multiple loop partition scheduling technique combines the above two approaches. The experiments show that it can achieve the improvement over the traditional methods by about 15% to 45% for different benchmarks.

The traditional loop partition technique [1, 10] is an effective method to exploit the data locality in a singleton nested loop. However, 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 between different loop nests, thereby incur much more memory reference than multiple loop partition scheduling. This is demonstrated by the experiments in Section 5. Therefore, the traditional loop partition technique may be achieve a good result for a singleton nested loop, it cannot perform well to multiple loop nests. Multiple loop partition scheduling technique deal with multiple loop nests together and partition 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 with multiple processors or a processor with multiple function units. They shared 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 execute the memory operations to prepare the data in the first level memory. Two kinds of memory operation are supported by the memory units. *Prefetch* operation is used to load the data in advance, and *keep* operation is used to keep the data in the first level memory for the near future use. The objective of keep operation is to prevent unnecessary long time data swapping. It is needed when the first level memory is really restrictive. When the first level memory is large enough, the keep operation can be neglected because such data still remain in the first level memory. The overall schedule includes the memory schedule and ALU schedule, corresponding to arranging the memory 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.

In the past, a lot of work has been done on loop fusion [4] and loop tiling [8, 2]. Loop fusion is used to fuse the consecutive loops into a single loop to exploit the data locality and reduce the additional synchronization. However, the memory reference maybe too much to be hidden efficiently even after the loops are fused. Therefore, we use multiple loop partition scheduling in order to exploit more data locality. Moreover, the partition size is carefully selected such that the computation time and memory loading time are balanced and memory latency is tolerated effectively. In fact, loop fusion can be thought of as a special case of our technique when the partition size is 1. Loop tiling is a technique for grouping elemental computation points so as to increase computation granularity and thereby reduce communication time. The traditional loop tiling only considers the singleton loop and lack of the consideration of prefetching and scheduling.

[*]This work is partially supported by Graduate Fellowship of the Center for Applied Mathematics

Another technique related to the memory access latency is data layout optimizations [3]. They modify the memory storage order of multi-dimensional arrays so as to reduce the cache miss rate. Loop partition is in the higher level above the data layout transformation. We assume full associativity in the first level memory. For the lower associativity, data layout transformation can be applied first to prevent the cache conflict.

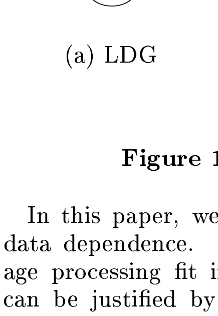
The rest of the paper is organized as follows. In Section 2, the program models and their representation are introduced. Section 3 presents the concept of partition. Section 4 describe multiple loop partition scheduling in detail and Section 5 is the experimental results.

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, $V = \{A, B, C, D\}$ 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.



```
      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
```

(a) LDG (b) Program segment

Figure 1: The code and its LDG

In this paper, we only consider 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 first [9]. The idea of multiple loop partition scheduling is illustrated with multiple two-dimensional nested loops. 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.

```
      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

Both two models comprise of 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 may be employed to obtain a sequence of loops with no intervening code. Loop permutation 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, 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.

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

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. The partitioning of multiple loop nests is built upon the partitioning of an individual loop nest. The former is the extension of the latter, 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)* [6]. Each node in the MDFG corresponds to a computation. An edge between two nodes represents that these two nodes have a data dependence. Its weight (d_e) is the data dependence vector between these two nodes. 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 guarantees all the data dependence vectors lie between P_x and P_y, such that the data dependence for a singleton nested loop can be maintained after partition.

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

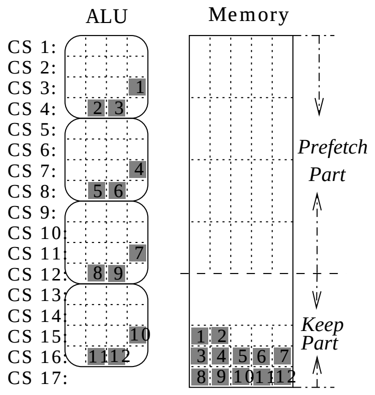


Figure 3: Partition schedule

A partition schedule, as shown in Figure 3, is made up of the ALU schedule, which schedules the ALU operations, and memory schedule, which schedules the prefetch and keep operations. The ALU schedule is formed by Multi-dimensional Rotation Scheduling Algorithm [7]. 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 [7]. The entire ALU schedule is the simple duplication of a single iteration schedule achieved by Rotation Scheduling Algorithm for each iteration in the partition. To arrange the memory operation, the data dependence is analyzed to collect the prefetch and keep operations for a partition. These operations are scheduled in the sequence of prefetch, keep operations, as well as maintain the time relation between the ALU execution and memory operation i.e., the keep operations must be issued after the corresponding execution. 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 at most one keep operation time longer than the ALU schedule length.

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 4 plots an example of a single partitoin. 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.

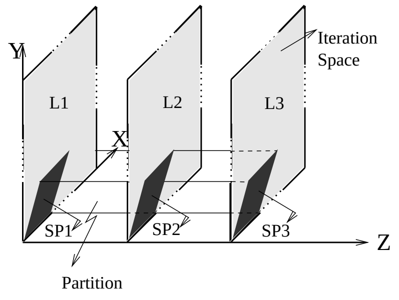


Figure 4: Partition and subpartitions

Multiple loop partition exploits not only the data locality inside each iteration space, but also the data locality among different loop nests. Take Figure 5 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. Assuming iterations $I2$ and $I3$ are in different rows, so do 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 scheduling technique. With the row-wise partition execution order, different grained in-partition execution order can be applied. The coarse-grained in-partition execution order is shown in Fig 6(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 6(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

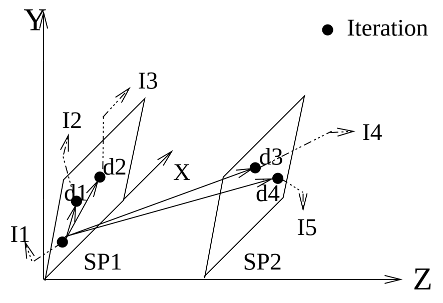
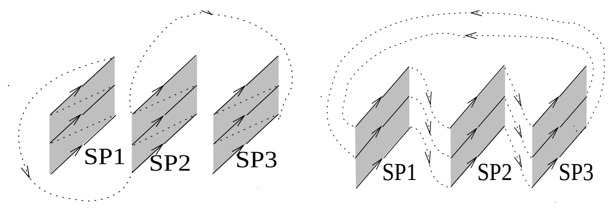


Figure 5: Different data dependences

until all the lines of iterations are executed in this manner.



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

Figure 6: 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 [6] 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 [6]proved by the following theorem.

THEOREM 2. *Given a legal LDG, there exists a set of retiming vector 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 $e : u \to v$, 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.

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

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, Algorithm 1 can be used to get rid of all of the violiations.

After applying the retiming vectors obtained above, all the data dependences 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.

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
```

As the same principle of partitioning an individual nested loop, the determination of the partition size depends on the relation between the memory and ALU schedules. A partition size which can balance the ALU and memory schedules is a prefered size. To derive the memory scheudle, all the memory operations need to be identified.

The number of these memory operations depends on the data dependence distance and the partition size. A certain data dependence decide a perfetch region, in which all data need to be prefetched in advance, and a keep region, in which all data will be kept in the first level memory. All the data in the keep region will be reused soon (in the next partition along the execution sequence), while the data in the prefetch region will be reused in a relatively long time. The Figure 7 shows two adjacent partitions along the partittion execution direction and two different data dependences. $d1$ is a data dependence between two loop nests, and $d2$ is a dependence inside a loop. Each of them deicde a prefetch region and a keep region in the partition, as shown in the figure. For the keep region determined by a data dependence, we can see that this dependence occurs between two adjacent partition. while it is not true for the prefetch region.

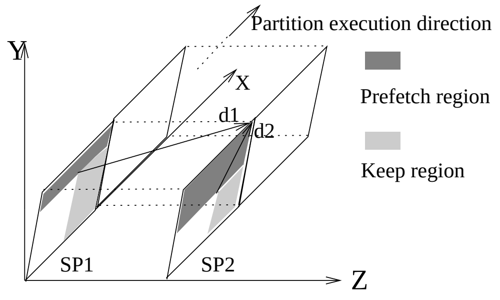


Figure 7: Prefetch and keep regions

In the algorithm 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. T_{keep} represents the time for a keep operation. The first "for" loop in the algorithm calculates the minimum partition size requirement. This requirement means that no data dependence will span two or more partitions. This 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. The theory behind these steps is: increase the partition size along P_y direction can maintain the number of prefetch operation. If the memory schedule is longer than ALU schedule at the start step, enlarge the partition along P_y direction only increases the number of keep operations and the number of iterations in the partition, which imply the ALU schedule will increase faster than the memory schedule if the partition size along P_x direction is large enough. It is this theory that guarantee 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.

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 $d = (d[1], d[2])$ 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 $L_{mem} \geq L_{ALU} + T_{keep}$ 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.
```

5. 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 applied 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.

In the simulation, we assume 4 ALU units 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 take 10 units time while keep operation take one unit time. The assumption is reasonable considering the big gap between the CPU and memory speeds.

We first compare multiple loop partition scheduling and partitioning 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 elements access are re-

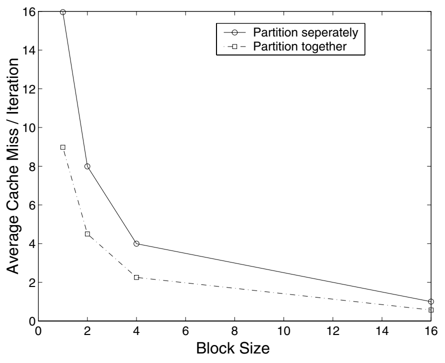


Figure 8: LL18 Kernel

duced. Figures 5 shows how *average cache miss/ Iteration* change 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 partitioning separately is always two times the number of our technique. This observation can be explained by the number of array reference per iteration. Partitioning separately has two times memory reference as many as multiple loop partition. Since the overall execution time is mainly decided by the memory access time, much more number of memory references mean the longer execution time. Therefore, multiple loop partition scheduling can get better performance than the traditional partitioning technique.

The simulation results are listed in Table 1. Three methods compared are *Original*, which is the original loop without any transformation, *Fusion*, which applied the loop fusion on the benchmarks, and *MLP*, which is multiple loop partition. Because it has been shown in [10] that software pipelining and loop tiling cannot perform better than the traditional separately partitioning, we didn't compare their results. The column *#loop nests* denotes the number of loop nests in each benchmarks. The *size* column is the partition size. We also list the theoretic lower bound in the L_b column. The ALU schedule is obtained without the consideration of the memory load latency. Thus, the ALU schedule length per iteration provides the lower bound for computation under the resource constraints. Compare 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 prevent the wasted time caused by the dominated memory reference time. Therefore, the multiple loop partition scheduling technique can efficiently exploit the data locality and hide the memory access.

The performance of multiple loop partition is very close to the lower bound, which demonstrates that the memory latency is hidden very efficiently. Their difference is mainly 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. Also,

	#loop nests	Original	Fusion	MLP	size	L_b
Example	4	8.5	5.41	**2.34**	8×8	2
Jacobi	2	5.25	4.0	**2.16**	4×8	2
LL18	3	20.85	13.44	**11.5**	4×4	11
Tomcatv	5	25.25	20.38	**18.26**	4×4	18

Table 1: Experiment result of average execution time

the extra keep operations by which the memory schedule is longer than the ALU schedule play a minor role in this difference.

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 independent 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 the same as the fusible loop problem [5] in essence.

6. REFERENCES

[1] 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.

[2] 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.

[3] M. Kandemir, A. Choudhary, and N. Shenoy. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), Feb 1999.

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

[5] 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.

[6] 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.

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

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

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

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