Loop Scheduling with Complete Memory Latency Hiding on Multi-core Architecture *

Chun Xue[1] Zili Shao[2] Meilin Liu[1] Meikang Qiu[1] Edwin H.-M. Sha[1]
[1]Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA
{ cxx016000, mxl024100, mxq012100, edsha}@utdallas.edu
[2]Department of Computing
Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
cszshao@comp.polyu.edu.hk

Abstract

The widening gap between processor and memory performance is the main bottleneck for modern computer systems to achieve high processor utilization. In this paper, we propose a new loop scheduling with memory management technique, **Iterational Retiming with Partitioning (IRP)***, that can completely hide memory latencies for applications with multi-dimensional loops on architectures like CELL processor [1]. In IRP, the iteration space is first partitioned carefully. Then a two-part schedule, consisting of processor and memory parts, is produced such that the execution time of the memory part never exceeds the execution time of the processor part. These two parts are executed simultaneously and complete memory latency hiding is reached. Experiments on DSP benchmarks show that IRP consistently produces optimal solutions as well as significant improvement over previous techniques.*

1 Introduction

With the constant reduction in feature size, multi-core processors and system-on-chip(SoC) designs have gain popularity in recent years. One of the main limitations of modern microprocessor performance is memory latency, which is increasing along with the increasing CPU speed. Maintaining cache consistency and minimize power consumption are also big challenges to multi-core processors and SoC designs. As a result, many researches have turned to software managed memory techniques for solutions. A well planned data prefetching scheme may reduce the memory latency by overlapping processor computations with memory access operations to achieve high throughput computations. In this paper, we develop a methodology called *Iterational Retiming with Partitioning (IRP)* which generates schedules with full parallelism at iteration level and hides memory latency completely with software prefetching technique.

IRP algorithm takes memory latency into account while optimizing schedules for multi-dimensional loops. It can be used in computation intensive applications (especially multi-dimensional applications) when two-level memory hierarchy exists. These two levels of memory are abstracted as local memory and remote memory. We assume each processor has multiple cores and multiple memory units. The processing cores work on computations, while the memory units are special hardwares that perform operations like prefetching data from the remote memory to the local memory. One real world example of this type of architecture is the CELL processor [1]. It has 9 processing cores, and software managed local memories that can prefetch data from the remote memory.

The partitioning (tiling) technique is incorporated into the IRP algorithm. We partition the entire iteration space and then execute one partition at a time. Iterational retiming [2] is used in conjunction with partitioning technique. Iterational retiming first partitions the iteration space into basic partitions, and then retiming is performed at iteration level so that full parallelism is reached and all the iterations in each partition can be executed in parallel. With this nice property, each core can execute an iteration in parallel and no communication overhead is needed among cores. Scheduling for the processing cores becomes an easy task. **We analyze the legal partition shape and provide mechanism to determine the right partition size to guarantee an optimal two-part schedule where memory latency is hidden completely.** Furthermore, we estimate the requirement of the local memory size for executing each partition. The estimation gives designers a good indication of the local memory requirement.

Prefetching schemes based on hardware [3, 4], software [5], or both [6, 7] have been studied extensively. In hardware prefetching schemes, the prefetching activities are controlled solely by the hardware. In contrast, software prefetching schemes rely on compiler technology to analyze a program statistically and insert explicit prefetch instructions into the program code. One advantage of soft-

* This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, and NSF CCR-0309461, NSF IIS-0513669, Microsoft, USA.

ware prefetching is that a lot of the compile-time information can be explored to effectively issue prefetching instructions.

Various techniques have been proposed to consider both scheduling and prefetching at the same time. Fei Chen et al. [8, 9] show the successful usage of the partitioning idea in improving loop schedules. They presented the first available technique that combines loop pipelining, prefetching, and partitioning to optimize the overall loop schedule. Zhong Wang et al. [10, 11] took it one step further in finding the optimal partition under memory constraints. However, their techniques only explore instruction level parallelism and are not able to take advantage of all the hardware resources available in multi-core processors. To the author's knowledge, the IRP technique proposed in this paper is the first available technique that generates optimal two-part schedules and hides memory latency completely on multi-core processors.

Experiments are done on DSP benchmarks for IRP algorithm and comparisons have been drawn with other scheduling algorithms, such as list scheduling and PSP algorithm [8]. In the experiments, we calculate the lower bounds of average schedule length based on the hardware resources limitation without considering memory latencies. For all of our test cases, IRP algorithm always reaches this lower bound. The average schedule length obtained by IRP is 46.1% of that derived using list scheduling and 65.5% of that from PSP. The result of our experiments shows that careful partitioning the iteration space is very important for optimizing the overall schedule. IRP algorithm always work correctly to produce optimal schedules no matter how big the speed gap between the processor and the remote memory access time is.

The remainder of this paper is organized as follows. A motivating example is shown in Section 2. Section 3 introduces basic concepts and definitions. The theorems and algorithms are proposed in Section 4. Experimental results and concluding remarks are provided in Section 5 and 6, respectively.

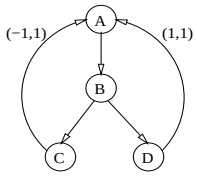
2 Motivating Examples

In this section, we provide a motivating example that shows the effectiveness of the IRP technique. In the example, we compare the schedules generated by list scheduling, rotation scheduling, PSP and IRP respectively. For simplicity, we only show the results of these four techniques without going into details on how each result are generated. The multi-dimensional loop used in this section is shown in Figure 1(a). There are two levels, i and j, in this loop. The MDFG representation of this two-level loop is shown in Figure 1(b). MDFG stands for Multi-dimensional Data Flow Graph. A node in MDFG represents a computation, and an edge in MDFG represents a dependence relation between two nodes. Each edge is associated with a delay that helps to identify which two nodes are linked by this edge.

In this example, we assume that in our system, there are

```
for i=0 to n  do
   for j=0 to m do
      A[i,j] = C[i+1,j−1] + D[i−1,j−1];
      B[i,j] = A[i,j] + 7;
      C[i,j] = B[i,j] * 5;
      D[i,j] = B[i,j] − 9;
   end for
end for
```

(a)



(b)

Figure 1. (a) Example program. (b) MDFG.

2 memory units and 2 processing cores in the processor, and there are 2 ALUs inside each core. We also assume that it takes 1 clock cycle to finish each computation node, 3 clock cycles to finish a data fetch, and 2 clock cycles to write back data.

Based on the dependency constraints in the MDFG shown in Figure 1(b), a list scheduling generated schedule is shown in Figure 2(a). For simplicity, one iteration is shown to represent a schedule. The notation A(0) in the figure represents the computation of node A in iteration 0, and the notation fA0 represents the data fetch operation for node A in iteration 0. Two fA0 are needed because node A needs two input data. As we can see, it takes 6 clock cycles for the schedule generated by list scheduling to finish one iteration, 3 clock cycles for the fetch operations and 3 clock cycles for the computations. After we apply rotation scheduling, which implicitly uses retiming, the resulting schedule is shown in Figure 2(b). The schedule length is now reduced to 5 clock cycles. The rotation technique explores instruction level parallelism, reduces processor part of schedule by redistributing instructions between different iterations. However, since no consideration is given to the memory, the long memory fetch time still dominates the total execution time.

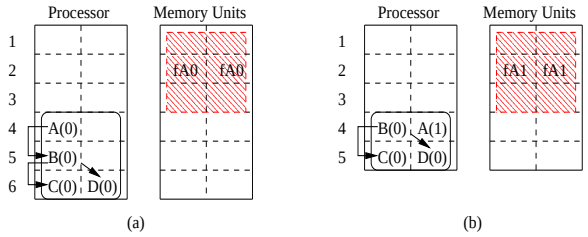


Figure 2. (a) List sch. (b) Rotation sch.

In order to hide memory latency, we apply PSP algorithm [8] which takes into account the balance between computation and memory access time. A much better performance is obtained in Figure 3(a). There are 4 fetch operations in the memory part of schedule. Operation fA3 represents fetch operation for computation A in iteration 3. Both fA3 operations finish before computation A(3) is scheduled at step 5, so A(3) is ready to execute without waiting for the data. In this schedule, 4 iterations are scheduled at a time as a partition, and it takes 10 clock cycles to finish all 4 iterations because of the overlapping execution of processor and memory parts. As a result, the average

time to complete one iteration is 2.5 clock cycles. However, because of the dependency constraints presented both inside each partition and between partitions, we can not take full advantage of all the hardware resources available. Only one core of the processor is used.

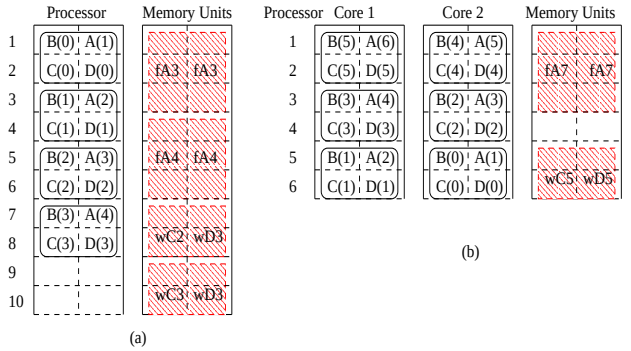


Figure 3. (a) PSP sch. (b) IRP Sch.

To take full advantage of all the hardware resources, we generate a schedule using our proposed IRP algorithm. The result is shown in Figure 3(b). It takes 6 clock cycles for 6 iterations to complete. Hence the average time to complete one iteration is 1 clock cycles. This huge performance improvement is generated by exploring iterational level parallelism so that all the iterations in a partition could be scheduled in parallel, and both processor cores can be utilized. In fact, the performance gain achieved by IRP algorithm scales linearly with the amount of hardware resources available. For example, if there are 10 cores in each processor instead of 2, we can achieve an average execution time of 0.2 clock cycle to complete an iteration, which is a huge gain compared to all the previous techniques.

3 Basic Concepts and Definitions

In this section, we introduce basic concepts which will be used in the later sections. First, we discuss the architecture model used in this paper. Second, we introduce the model and notions that we use to analyze nested loops. Third, loop partitioning technique is introduced. Fourth, iterational retiming technique is presented. In this paper, our technique is presented with two dimensional notations. It can be easily extended to multi-dimensions.

3.1 Architecture Model

We assume that there are multiple processing cores and special hardwares called memory units inside each processor. Associated with each processor is a small local memory. Accessing this local memory is fast. There is also a large multi-port remote memory. However, accessing it is slow. The goal of our technique is to prefetch the operands of all computations into the local memory before the actual computations take place. These prefetching operations are performed by the memory units. Two types of memory operations, prefetching and write_back, are supported by the memory units. The prefetching instruction prefetches data from the remote memory to the local memory; The write_back instruction writes data back to the remote memory for future accessing. Both of these instructions are issued to ensure those data will be referenced soon will appear in the local memory. Here we are not concerned with the load/store operations, which are scheduled in the processing cores to load or store data from local memory. They are not operations scheduled in the memory units.

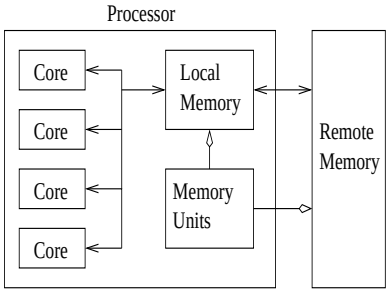


Figure 4. Architecture diagram.

Our architecture is a general model. A real implementation was done in CELL processor [1]. Inside each CELL processor, there are 9 processing cores. There are 256k local memory called local store inside each core. This local memory is not a cache, and prefetching operations on these local memories are done by special memory units. A total of 16 concurrent memory operations is possible at one time for each core. The compiler and software design for CELL processor need to generate schedule for both the processing cores and memory units. With this type of architecture, CELL processor gains more than 10 times the performance of a modern top of line CPU.

3.2 Modeling Nested Loops

Multi-dimensional Data Flow Graph is used to model nested loops and is defined as follows. A *Multi-dimensional Data Flow Graph (MDFG)* $G = \langle V, E, d, t \rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \subseteq V * V$ is the set of dependence edges, d is a function and d(e) is the multi-dimensional delays for each edge $e \in E$ which is also known as dependence vector, and t is the computation time of each node. We use d(e) = (d_i, d_j) as a general formulation of any delay shown in a two-dimensional DFG (2DFG). An example is shown in Figure 5. In Figure 5(a), there is a two-dimensional loop program. In this loop program, there are two computations, that computes A[i,j] and B[i,j] respectively. Figure 5(b) shows the corresponding MDFG. In this MDFG, there are two nodes, A and B, representing the two computations in the original loop program. There are two edges from node B to node A, which means the computation of A depends on two instances of data from node B.

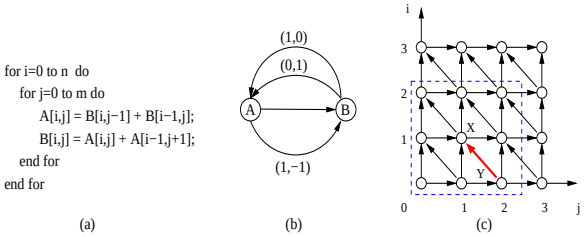


Figure 5. (a) A nested loop. (b) Its MDFG. (c) Its iteration space.

An **iteration** is the execution of each node in V exactly once. The computation time of the longest path without delay is called the *iteration period*. For example, the iteration period of the MDFG in Figure 5(b) is 2 from the longest zero-delay path, which is from node A to B. If a node v at iteration j, depends on a node u at iteration i, then there is an edge e from u to v, such that d(e) = j - i. An edge with delay (0,0) represents a data dependence within the same iteration. A legal MDFG must not have zero-delay cycles. Iterations are represented as integral points in a Cartesian space, called **iteration space** , where the coordinates are defined by the loop control indexes. Figure 5(c) shows a representation of the iteration space for the MDFG presented in figure 5(b). For simplicity, we will always show a small section of the iteration space with respect to our examples. A **schedule vector s** is the normal vector for a set of parallel equitemporal hyperplanes that define a sequence of execution of an iteration space. By default, a given nested loop is executed in a row-wise fashion, where the schedule vector $s = (1, 0)$.

Retiming [12] can be used to optimize the cycle period of a MDFG by evenly distributing the delays in it. Given a MDFG $G = \langle V, E, d, t \rangle$, retiming r of G is a function from V to integers. For a node $u \in V$, the value of $r(u)$ is the number of delays drawn from each of its incoming edges of node u and pushed to all of its outgoing edges. Let $G_r = \langle V, E_r, d_r, t \rangle$ denote the retimed graph of G with retiming r, then $d_r(e) = d(e) + r(u) - r(v)$ for every edge $e(u \rightarrow v) \in E_r$ in G_r.

Rotation Scheduling presented in [13] is a loop scheduling technique used to optimize loop scheduling with resource constraints. It transforms a schedule to a more compact schedule iteratively. In most cases, the node level minimum schedule length can be obtained in polynomial time by rotation scheduling. In each step of rotation, nodes in the first row of the schedule are rotated down. By doing so, the nodes in the first row are rescheduled to the earliest possible available locations. From retiming point of view, each node gets retimed once by drawing one delay from each of incoming edges of the node and adding one delay to each of its outgoing edges in the DFG. The detail of rotation scheduling can be found in [13].

3.3 Partitioning the iteration space

Instead of executing the entire iteration space in the order of rows and columns, we can first partition it and then execute the partitions one by one. The two boundaries of a partition are called the *partition vectors*. We will denote them by P_i and P_j. Due to the dependencies in the MDFG, partition vectors cannot be arbitrarily chosen. For example, consider the iteration spaces in Figure 6. Iterations are now represented by dots and inter-iteration dependencies. Partitioning the iteration space into rectangles, with $P_j = (0, 1)$ and $P_i = (2, 0)$, as shown in Figure 6(a), is illegal because of the forward dependencies from $Partition_{(0,0)}$ to $Partition_{(0,1)}$ and the backward dependencies from $Partition_{(0,1)}$ to $Partition_{(0,0)}$. Due to these two-way dependencies between partitions, we cannot execute either one first. This partition is therefore not implementable and is thus illegal. In contrast, consider the alternative partition shown in Figure 6(b), with $P_j = (0, 1)$ and $P_i = (2, -2)$. Since there is no two-way dependency, a feasible partition execution sequence exists. For example, $Partition_{(0,0)}$ is executed first, then $Partition_{(0,1)}$, and so on. Therefore, it is a legal partition.

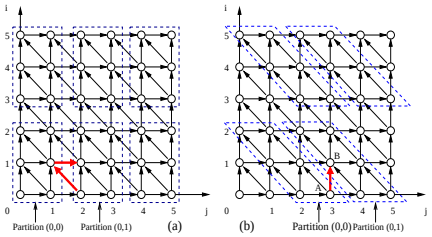


Figure 6. (a) An illegal partition of the iteration space. (b) A legal partition.

Iteration Flow Graph is used to model nested loop partitions and is defined as follows. An *Iteration Flow Graph (IFG)* $G_i = \langle V_i, E_i, d_i, t_i \rangle$ is a node-weighted and edge-weighted directed graph, where V_i is the set of iterations in a partition. The number of nodes $|V_i|$ in an IFG G_i is equal to the number of nodes in a partition. $E_i \subseteq V_i * V_i$ is the set of iteration dependence edges. d_i is a function and $d_i(e)$ is the multi-dimensional delay for each edge $e \in E_i$. t_i is the computation time for each iteration. An iteration flow graph $G_i = \langle V_i, E_i, d_i, t_i \rangle$ is *realizable* if the represented partition is legal. An example of IFG of the legal partition in Figure 6(b) is shown in Figure 7. Edges with (0,0) delay are shown in thicker line, which represents data dependencies within the same partition. For the edge between iteration A and iteration B in the Figure 7, shown in dashed line, it has delay d(e) of (0,1), which represents iteration B in partition (0,1) depends on iteration A in partition (0,0).

3.4 Iterational Retiming

Iterational retiming [2] achieves full parallelism for iterations in a partition. It is carried out in the following steps. Given a MDFG as input, first a basic partition is obtained

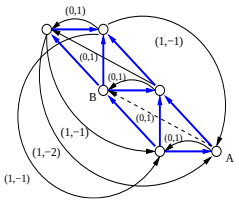


Figure 7. IFG for the partition in Figure 6 (b).

by identifying the directions of legal partition vectors and determining the partition size. then, iterational retiming is applied to create the retimed partition.

First, we will find a basic partition. Among all the delay vectors in a MDFG, two extreme vectors, the clockwise (CW) and the counterclockwise (CCW), are the most important vectors for deciding the directions of the legal partition vectors. Legal partition vector cannot lie between CW and CCW. In other words, they can only be outside of CW and CCW or be aligned with CW or CCW. For example, for the nested loop in Figure 5, there are three non-zero delay vectors, (0,1), (1,0), and (1,-1). (0,1) is the CW vector and (1,-1) is the CCW vector. If partition vectors are chosen to be $P_j = (0,1)$ and $P_i = (2,0)$, as shown in Figure 6(a), it is an illegal partition because partition vector (2,0) lies between CW and CCW and causes cycles in the partition dependency graph as shown in Figure 6(b). we choose P_j to be aligned with j-axis, and P_i to be aligned with CCW for the basic partition. This is a legal choice of partition vectors because the i elements of the delay vectors of the input MDFG are always positive or zero, which allows the default row-wise execution of nested loops.

After basic partition is identified via P_i and P_j, an IFG $G_i = \langle V_i, E_i, d_i, t_i \rangle$ can be constructed. An *iterational retiming* r is a function from V_i to Z^n that redistributes the iterations in partitions. A new IFG $G_{i,r}$ is created, such that the number of iterations included in the partition is still the same. The retiming vector r(u) of an iteration $u \in G_i$ represents the offset between the original partition containing u, and the one after iterational retiming. When all the edges $e \in E_i$ have non-zero delays, all the nodes $v \in V_i$ can be executed in parallel, which means that all the iterations in a partition can be executed in parallel. We call such a partition a *retimed partition*.

After the iterational retiming transformation, the new program can still keep row-wise execution, which is an advantage over the loop transformation techniques that need to do wavefront execution and need to have extra instructions to calculate loop bounds and loop indexes. Algorithms and theorems for iterational retiming is presented in detail in [2].

3.5 IRP algorithm framework

IRP generates a schedule consisting of two parts: the processor part and the memory part. The original MDFG usually contains inter-iteration dependencies and intra-iteration dependencies. This causes difficulty to generate efficient code for the processor part because little parallelism exists among iterations. For example, CELL processor [1] have 9 processing cores. Most of the cores will be sitting idle if using the original MDFG to schedule. In order to increase parallelism, iterational retiming [2] is applied to achieve full parallelism to take advantage of all the available processing cores.

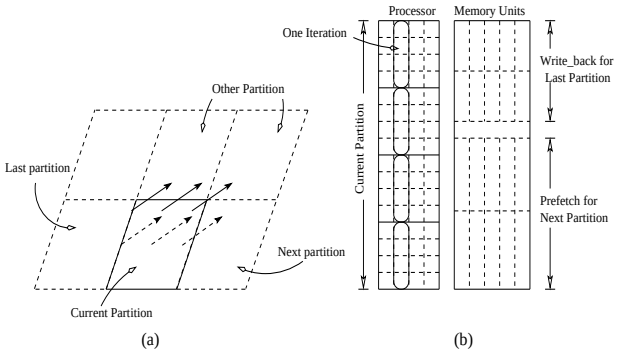


Figure 8. (a) A representation of last partition, current partition, next partition, and other partitions. (b) An IRP Schedule.

We call the partition just finished execution the **last partition**, the partition that is currently being executed the **current partition**, and the partition that will be executed next the **next partition**. Any other partition which has not been executed are called **other partition** (see Figure 8(a)). When scheduling the memory part, if a non-zero delay edge passes from the current partition into an other partition, a prefetch and a write_back operation are needed, shown by a solid vector in Figure 8(a). Each directed edge from the current partition to the next partition corresponds to a store operation, shown by a dashed vector in the figure. When scheduling the memory part, we wish to prefetch all the operands needed by the next partition into the local memory at the same time the processor computations are being performed for the current partition.

Figure 8(b) gives an example of our overall schedule — the processor part as well as the memory part. There are 16 iterations in one partition. In the processor part, the IRP scheduling generates a 4-control-step (CS) processor schedule for one iteration and four iterations are executed at the same time by four processing cores. This schedule is then duplicated 4 times for all the iterations inside the partition. In contrast, the partition boundaries are not preserved in the memory part of the schedule. All the write_back operations for the last partition are scheduled first, then all the prefetch operations for the next partition are scheduled next. As we can see, in the overall schedule, 16 iterations are finished in 16 control steps. That is, on average, ave_len(overall) = 16/16 = 1 control step per iteration. The framework of our algorithm is illustrated in Algorithm 4.1.

4 IRP Scheduling

IRP scheduling consists of two parts, the processor part and the memory part. The processor part of the schedule for one iteration is generated by using the rotation scheduling algorithm. Rotation scheduling is a loop pipelining technique which implicitly uses retiming heuristic for scheduling cyclic graphs. Rotation scheduling is described in detail in [13]. In IRP scheduling, one-dimensional rotation is applied instead of multi-dimensional(MD) rotation [14]. This is because multi-dimensional(MD) rotation will implicitly change the schedule vector and the execution sequence of the iterations. Since we wish to maintain row-wise execution, only one-dimensional rotation is used.

Algorithm 4.1 Iterational Retiming with Partitioning (IRP)

Input: MDFG $G = \langle V, E, d, t \rangle$, number of cores C
Output: A retimed partition that hides memory latency
/* based on the input MDFG, find the initial schedule for an iteration */
$s \leftarrow$ rotation_scheduling(MDFG) ;
$L_s \leftarrow$ get_schedule_length(s) ;
/* find basic partition size and shape */
$P_{j0} \leftarrow (0, 1)$;
$P_{i0} \leftarrow$ CCW vector of the all delays;
$f_j = \{ k \mid (0, k)$ is smallest $(0, x)$ delays $\}$;
Obtain f_i based on inequalities in Theorem 4.2;
$P_i = f_i \cdot P_{i0}$;
$P_j = f_j \cdot P_{j0}$;
Obtain basic partition with P_i, P_j;
/* transform the basic partition into a retimed partition */
Call Step 2 of the iterational retiming algorithm;
/* produce the overall schedule */
Number the iterations;
Processor part scheduling;
Memory part scheduling;

Scheduling of the memory part consists of several steps. First, we need to decide a legal partition. Second, partition size is calculated to ensure an optimal schedule. Third, iteration retiming is applied to transform the basic partition into a retimed partition so that all the iterations can be scheduled in parallel. Fourth, iterations are numbered and both the processor part and memory part of schedule are generated. We will explain these steps blow in greater detail.

A partition is identified by two partition vectors, P_i and P_j, where $P_i = P_{i0} \times f_i$ and $P_j = P_{j0} \times f_j$. While P_{i0} and P_{j0} determine the direction and shape of a partition, f_i and f_j determine the size of a partition. How to choose the vectors P_{i0} and P_{j0} to identify the shape of the legal basic partition is discussed in detail in section 3.4. How to choose f_j is shown in Algorithm 4.1. We will pay special attention to how f_i is chosen to achieving the goal of complete memory latency hiding, which means the schedule length of the memory part will always be equal or smaller than the schedule length of the processor part. How and why f_i is chosen is discussed in detail in Section 4.2.

After obtaining the basic partition directions and size, we apply the step 2 of the iteration retiming algorithm [2] to transform the basic partition into a retimed partition, so that all the iterations inside the retimed partition can be executed in parallel. After a retimed partition is identified, we can begin to construct both the processor and the memory part of the schedule.

4.1 Schedule Generation

For each partition, the processor part of schedule is generated in the default order. The memory part of the schedule is generated in a pipelining fashion. Write_back operations for the last partition are scheduled at the beginning as the data has already been generated and no data dependencies are pending. Then, all the prefetch operations for the next partition are scheduled. Because the way that the partition size is chosen, we can always get a memory schedule that is not longer than the processor schedule, hence memory latency is hidden and optimal schedule is reached. An example is shown in Figure 9. In this example, current partition includes iterations numbered from 0 to 7. While iterations 0 to 7 are executed in the processor, prefetching for iterations 8 and 9 is done in the memory units. Write_back operations of iteration 6 and 7 are done in the memory units during the execution of next partition, when iterations 8 and 9 are processed in the processor. As we can see, at the same time that processor is executing iterations inside current partition, memory units are executing write_back for the last partition and prefetching operations for the next partition. Partitions are executed in a pipelining fashion.

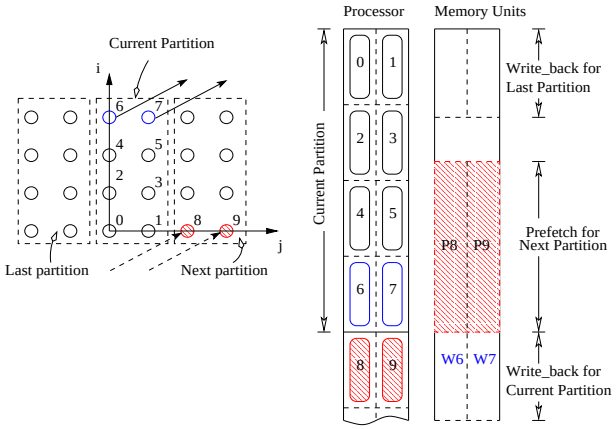


Figure 9. (a) Iteration space. (b) IRP Sch.

4.2 Partition size

In this section, we will show how to determine the partition size, so that complete memory latency hiding can always be achieved. First we will define the number of prefetching operations given a partition size of f_i and f_j. The number of prefetch operations can be approximated by calculating the shaded area, as shown in Figure 10, with respect to every inter-iteration delay vector $d \in D$. Consider a delay vector $d = (d_i, d_j)$, in Figure 10, all of its duplicate vectors originating in the region PQRS will enter other

partitions, which is when the write_back and prefetch operations are needed. Let the area of PQRS be A_{goto_others}.

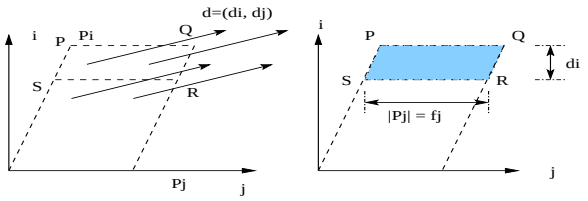


Figure 10. Calculating the number of delay edges crossing the boundary of the current partition.

Lemma 4.1. *Given a delay vector* $d = (d_i, d_j)$, $A_{goto\_others}(d) = f_j d_i$.

Proof. Shown as the shaded area in Figure 10, we have $A_{goto\_others}(d) = area(PQRS) = f_j d_i$. □

Summing up all of these areas for every distinct delay vector $d \in D$, we obtain the total number of prefetch operations as:

$$NUM_{pre} = \sum_{d} A(d) = \sum_{d} (f_j d_i), \quad \forall d = (d_i, d_j)$$

From this definition of number of prefetch operations, we know that it is proportion to the size of f_j. However, it does not change related to f_i. In order to hide the memory latency, we will try to have the least number of prefetch operations. So we will keep f_j fixed and find the right f_i to achieve an optimal schedule where memory latencies are hidden completely.

Theorem 4.1. *The number of prefetching remains unchanged before and after iterational retiming.*

Proof. Because iterational retiming only use $r = (0, 1)$ as the retiming vector, all the delays $d = (d_i, d_j)$ after retiming still have the same d_i, and the size of the partition does not change, hence the number of prefetching operations NUM_{pre} does not change after iterational retiming. □

From the above theorem, we know that the number of prefetching remains the same. So we do not need to calculate the number of prefetching operations after the iterational retiming transformation. The following theorem shows how we can use the above lemma and theorem to find the right partition factor of f_i.

Theorem 4.2. *Assume that* $N_{core} \leq N_{mem}$*, and the following inequality is satisfied:*

$$\left\lceil \frac{NUM_{pre}}{N_{mem}} \right\rceil \times T_{pre} + \left\lceil \frac{NUM_{write}}{N_{mem}} \right\rceil \times T_w \leq L_s \times \frac{f_i \times f_j}{N_{core}}$$

The length of the memory part of the schedule is at most the same as that of the processor part.

Proof. In the memory part of the schedule, the length of the prefetch part is $\left\lceil \frac{NUM_{pre}}{N_{mem}} \right\rceil \times T_{pre}$, and the length of the write_back part is $\left\lceil \frac{NUM_{write}}{N_{mem}} \right\rceil \times T_w$. Therefore, the left-hand side is the worst-case schedule length of the memory part. In the right hand side, L_s represents the schedule length of one iteration, and $\frac{f_i \times f_j}{N_{core}}$ represents the number of iterations to be executed by each core. Hence the right hand side represents the length of the processor part of schedule. By requiring the worst-case schedule length of the memory part to be less then the processor part, we guarantee that the memory part of the schedule is not longer then the processor part at any time. □

The number of write_back operations equals the number of prefetch operations. This is because we only write back those data that we will ever need to fetch from remote memory. We know how to calculate the number of prefetch operations, so we know how to calculate the number of write_back operations as well. Solving the above inequality, we get the right f_i to ensure an optimal memory latency hiding schedule. It is important to note that since f_j is already determined beforehand, f_i can always be calculated easily.

5 Experiments

In this section, the effectiveness of the IRP algorithm is evaluated by running a set of simulations on DSP benchmarks. The following DSP benchmarks with two-dimensional loops are used in our experiments: WDF (Wave Digital Filter), IIR (the Infinite Impulse Response Filter), 2D (the Two Dimensional filter), Floyd (Floyd-Steinberg algorithm), and DPCM (Differential Pulse-Code Modulation device).

Table 1 shows our scheduling results. The first column gives the benchmarks' names. The second and third columns are the parameters of the input MDFG, showing the processing core and memory unit resource constraints. The partition generated by the IRP algorithm is shown in the fourth to sixth columns. The final schedule is shown in the next three columns. Column "L" gives the length of the overall schedule and Column "L_{avg}" is the average length ($\frac{L}{N_{iter}}$). In order to compare our results with the lower bound, as well as the results from other algorithms, we calculated the lower bounds of the schedule length, $\lceil \frac{N_{nodes}}{N_{core} \times N_{alu}} \rceil \times T_{alu}$, and put them in Column "LB". Here, the lower bound is caused by limitation in hardware resources. We also ran the same set of benchmarks using list scheduling and partition scheduling with prefetching(PSP) [9]. The results are shown in Columns "List" and "PSP", respectively, where the sub-column "len" is the schedule length and sub-column "ratio" compares the IRP schedule length with that of list scheduling and PSP scheduling, i.e., ratio = $\frac{L_{avg}}{len}$.

Benchmark	Parameters		Partition			IRP			List		PSP [9]	
	N_{core}	N_{mem}	f_j	f_i	N_{iter}	L	L_{avg}	LB	len	ratio	len	ratio
IIR	2	2	1	4	4	16	**4**	4	8	50%	6.08	65.8%
WDF	2	2	1	4	4	12	**3**	3	6	50%	4.25	70.6%
FLOYD	2	2	1	4	4	16	**4**	4	11	36.4%	6	66.7%
DPCM	2	2	1	4	4	16	**4**	4	7	57.1%	4.08	98%
2D(1)	2	2	1	4	4	34	**9**	9	16	56.3%	12	75%
2D(2)	2	2	1	4	4	4	**1**	1	4	25%	2.25	44.4%
MDFG1	2	2	1	4	4	4	**1**	1	7	14.3%	4.25	23.5%
MDFG2	2	2	1	4	4	32	**8**	8	10	80%	10.04	79.7%
Average Ratio									-	46.1%	-	65.5%

Table 1. Experiment results on DSP filter benchmarks when T_{alu}=1, T_{pre}=2 and T_{wri}=2.

In Table 1, we assume that each processor operation takes 1 time unit, each prefetch takes 2 time units, and each write_back takes 2 time units. And assume that inside each processing core, there are 2 ALU units available to execute computations. As we can see, list scheduling rarely achieves the optimal schedule length; the schedules are often dominated by a long memory part. In other words, the list schedules are not well balanced. Although PSP is better then list scheduling by generating a balanced schedule, it is not able to fully take advantage of all the hardware resources available by exploring higher level of parallelism.

IRP algorithm consistently produces optimal schedules, as shown by the bold figures in the table. All of the resulting schedules reach the optimal schedule length. In Table 1, the average ratio of the schedule length from the IRP algorithm to those from list scheduling and PSP are 46.1% and 65.5%, respectively.

6 Conclusion

In this paper, we propose a new loop scheduling with software prefetching technique. An algorithm, Iterational Retiming with Partitioning(IRP), is presented. IRP can achieve complete memory latency hiding for multi-core architectures. Experimental results show IRP is a very promising technique.

References

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, 2005.

[2] Chun Xue, Zili Shao, Meilin Liu, and Edwin H.-M. Sha, "Iterational retiming: Maximize iteration-level parallelism for nested loops," *Proc. of the IEEE Int. Symp. on System Synthesis*, pp. 309–314, Sep 2005.

[3] F. Dahlgren and M. Dubois, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Transaction on Parallel and Distributed Systems*, pp. 733–746, July 1995.

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

[5] T. Ozawa, Y. Kimura, and S. Nishizaki, "Cache miss heuristics and preloading techniques for general-purpose programs," *Proceedings of the 28th annual ACM/IEEE international symposium on Microarchitecure*, pp. 243–248, 1995.

[6] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems, "Guided region prefetching: A cooperative hardware/software approach," *Proc. of the 30th annual international symposium on Computer architecture*, pp. 388–398, May 2003.

[7] Tien-Fu Chen and Jean-Loup Baer, "A performance study of software and hardware data prefetching schemes," *International Symposium on Computer Architecture*, pp. 223–232, 1998.

[8] Fei Chen and Edwin H.-M. Sha, "Loop scheduling and partitions for hiding memory latencies," *International Symposium on System Synthesis*, 1999.

[9] Fei Chen, Timothy W. O'Neil, and Edwin H.-M. Sha, "Optimizing overall loop schedules using prefetching and partitioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 6, pp. 604–614, June 2000.

[10] Zhong Wang, QingFeng Zhuge, and E.H.-M. Sha, "Scheduling and partitioning for multiple loop nests," *International Symposium on System Synthesis*, pp. 183–188, October 2001.

[11] Zhong Wang, E.H.-M. Sha, and Yuke Wang, "Partitioning and scheduling dsp applications with maximal memory access hiding," *Eurasip Journal on Applied Singal Processing*, pp. 926–935, September 2002.

[12] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[13] Liang-Fang Chao, Andrea LaPaugh, and Edwin H.-M. Sha, "Rotation scheduling: A loop pipeling algorithm," *IEEE Transaction on Computer Aided Design*, pp. 229–239, March 1997.

[14] Nelson Passos and Edwin H.-M. Sha, "Scheduling of uniform multi-dimensional systems under resource constraints," *IEEE Transactions on VLSI Systems*, vol. 6, pp. 719–730, Dec. 1998.