Optimal Loop Scheduling for Hiding Memory Latency Based on Two Level Partitioning and Prefetching. [*]

Zhong Wang
Dept. of Comp. Science & Eng.
University of Notre Dame
Notre Dame, IN 46556
Tel:(219)631-8720
Fax:(219)631-9260
zwang1@cse.nd.edu

Timothy W. O'Neil
Dept. of Comp. Science & Eng.
University of Notre Dame
Notre Dame, IN 46556
Tel:(219)631-8720
Fax:(219)631-9260
toneil@cse.nd.edu

Edwin H.-M. Sha
Dept. of Computer Science
Erik Jonsson School of Eng. and C.S.
Box 830688, MS EC 31
University of Texas at Dallas
Richardson, TX 75083-0688
edsha@utdallas.nd.edu

Abstract

The large latency of memory accesses in modern computers is a key obstacle in achieving high processor utilization. As a result, a variety of techniques have been devised to hide this latency. These techniques range from cache hierarchies to various prefetching and memory management techniques for manipulating the data present in the caches. In DSP applications, the existence of large numbers of uniform nested loops makes the issue of loop scheduling very important. In this paper, we propose a new memory management technique that can be applied to computer architectures with three levels of memory, the scheme generally adopted in contemporary computer architecture. This technique takes advantage of access pattern information that is available at compile time by prefetching certain data elements from the higher level memory before they are explicitly requested by the lower level memory or CPU. It also maintains certain data for a period of time to prevent unnecessary data swapping. In order to take better advantage of the locality of references present in these loop structures, our technique introduces a new approach to memory management by partitioning it and reducing execution to each partition, so that data locality is much improved compared with the usual pattern. These combined approaches – using a new set of memory instructions as well as partitioning the memory – lead to improvements in average execution times of approximately 35% over the one-level partition algorithm and more than 80% over list scheduling and hardware prefetching.

[*]This work is partially supported by grants NSF MIP95-01006, NSF ACS 96-12028 and JPL 961097

1 Introduction

Memory latency is becoming an increasingly important performance bottleneck as the gap between processor and memory speeds continues to grow. While cache hierarchies are an important step toward addressing the latency problem, they are not a complete solution. Techniques that can cope with the large latency of memory accesses are essential for achieving high processor utilization. A number of different solutions have been proposed. Prefetching data into the cache nearest to the CPU before it is required can tolerate the long memory latency, making this an effective solution. The previous research on prefetching can be classified into three categories: prefetching based on hardware [6, 10, 23], software [12, 14], or both [5, 15, 29]. Hardware-based prefetching techniques, requiring some support units connected to the cache, rely on the dynamic information available during program execution. In contrast, software prefetching schemes depend on compiler technology to analyze a program statically and insert explicit prefetch instructions into the program code. One advantage of software prefetching is that much compile-time information can be explored in order to effectively issue the prefetching instruction. Furthermore, many existing loop transformation approaches can be used to improve the performance of prefetching. Bianchini *et al* developed a runtime data prefetching strategy for software-based distributed shared-memory systems [2]. Wallace and Bagherzadel proposed a mathematical model and a new prefetching mechanism. A simulation on the SPEC95 benchmarks showed an improvement in the instruction fetching rate [24]. Mowry proposes a new compiler algorithm [13] for inserting prefetches into multiprocessor code. This algorithm attempts to minimize overheads by only issuing prefetches for references that are predicted to suffer cache misses. In their work, the ALU part of the schedule is not considered. Nevertheless, solely considering the prefetching is not enough for optimizing the overall system performance. As we point out in this paper, too many prefetching operations may lead to an unbalanced schedule with a very long memory part. In contrast, our new algorithm gives more detailed analyses of both the ALU and memory parts of the schedule. Moreover, partitioning the iteration space is another useful technique for optimizing the overall system performance.

On the other hand, several loop pipelining techniques have been proposed. For example, Wang and Parhi presented an algorithm for resource-constrained scheduling of DSP applications when the number of processors is fixed and the objective is to obtain a schedule with the minimum iteration period [25]. Wolf *et al* studied combinations of various loop transformation techniques (such as fission, fusion, tiling, interchanging, etc) and presented a model for estimating total machine cycle time, taking into account

software pipelining, register pressure and loop overhead [28]. Passos and Sha proved that in the multi-dimensional case (e.g., nested loops), full-parallelism can always be achieved by using multi-dimensional retiming [16]. Modulo scheduling by Ramanujam [20] is a technique for exploiting instruction level parallelism (ILP) in the loop. It can result in high performance code but increased register requirements [11]. Rau and Eichenberger have done research on optimum modulo schedules, taking into consideration the minimum register requirement. They consider not only the data flow, but also the control flow of the program [8, 22]. None of the above research efforts, however, includes the prefetching idea or considers the data fetching latency in their algorithms.

We will restrict our study to nested loops with uniform data dependencies. Even if most loop nests have linear dependencies, the study of uniform loop nests is justified by the fact that an linear index loop nest can always be transformed into an uniform loop nest. This transformation (uniformization [7]) greatly reduces the complexity of the problem. In our model, we assume a system with a three-level memory hierarchy. It will take less time to access data from a lower level memory than a higher level memory, i.e., access to data from the first level memory is fastest while that from the third level memory is slowest. We also assume a processor consists of multiple ALU and memory units. The ALUs are for computations. The memory units are for performing memory operations to prefetch data from a higher level memory to a lower level memory. In our method, three schedules exist for different levels, the *ALU schedule* for the ALU computations, the *first level schedule* for the first level memory operations and the *second level schedule* for the second level memory operations. The longest of these three schedules determines the overall system schedule length. Given a uniform nested loop, our goal is to find the overall system schedule with the minimum length. This goal can be accomplished by overlaying the prefetching operations as much as possible with the ALU operations, so that the ALU can always keep busy without waiting for the operands. In order to implement prefetching, we find the best balance among these three schedules.

To increase the data locality, one good method is to group the elemental computation points. Related work can be found in the loop tiling and the one-level partition techniques. Loop tiling is mainly applied to the distributed system to reduce communication time. Wolf and Lam proposed a loop transformation technique for maximizing parallelism or data locality [27]. Boulet *et al* introduced a criterion for defining optimal tiling in a scalable environment. In his method, an optimal tile shape can be determined and the tile size obtained from the resources constraints [18]. Another interesting result was produced by Chame and Moon. They proposed a new tile selection algorithm for eliminating self interference and

simultaneously minimizing capacity and cross-interference misses. Their experimental results show that the algorithm consistently finds tiles that yield lower miss rates [3]. Nevertheless, the traditional tiling techniques only concentrate on reducing communication cost. They do not consider how to best balance the computation and communication schedules. There is no detailed schedule consideration in their algorithms. Special cases occur in DSP and image processing applications, where there exist a large number of uniform nested loops. Thus, detailed consideration of how to schedule the loop body efficiently is very important to these applications.

The one-level partition technique in [9, 26] combines two aspects of instruction level parallelism and memory access latency reduction to decrease the overall schedule for uniform dependence loops. In order to well overlay the prefetch operations with the CPU operations, a balance between them is found under the first level memory size constraint. However, the algorithm in [9] assumes that only a two level memory hierarchy is present. No analysis is performed with three or more levels of memory, as is the case in most contemporary computer architectures. Moreover, this algorithm does not consider the memory constraints in the first level memory. The memory size required by this algorithm might be so large that it cannot produce any feasible result in the experiment when such constraints exist. Although the algorithm in [26] considers the first level memory constraints, it also assumes only two levels in the memory hierarchy. Our studies have shown that the one-level partition technique cannot take full advantage of the second level memory in such an architecture. For instance, assume that 3 clock cycles are needed to fetch data from the second to the first level memory, 7 clock cycles to fetch data from the third to the second level memory, and 10 clock cycles to fetch data from the third to the first level memory. Suppose also that whenever a block is brought into the first level memory directly from the third level memory, a copy is kept in the second level memory. Our experiments have shown that the one-level partition technique produces poor results on such three-level memory hierarchy systems. With the DPCM filter as a benchmark, using the one-level partition technique for the first level memory and dynamic scheduling for the second level memory will result in almost the same average access time as the model in which no second level memory exists. The one-level partition technique resulted in an average fetching time of 9.968 cycles/fetch when the iteration space is large. In other words, the CPU always has to load data from the third level memory, which has a cost of 10 cycles/fetch. On the contrary, our two-level partitioning technique, by appropriately choosing the partition shape and size, can prefetch all the data into the second level memory before the first level memory needs to access them. These prefetch operations execute in parallel with the ALU computations. Therefore, the average fetching time can be

regarded as 3 cycles/fetch in this case. The lower average fetching time is the key to better performance under the first level memory size constraint, which is demonstrated by the experimental results.

In this paper, we apply the idea of partitioning to the extra levels in the memory hierarchy. In both the first and second level memories, we adopt the partition technique and make extensive use of compile time information about the usage pattern of the information produced in the loop to generate an approximated "balanced schedule" in the first and second level schedules in order to minimize the overall average execution time and make better use of the second level memory. This paper presents methods for deciding the best partition in both levels in order to achieve this goal. The new algorithm exceeds the performances of existing algorithms by optimizing both the ALU and memory schedules in the first level and taking into consideration a balance between the schedules of the two levels. In the situation of no memory constraint, the results produced by our algorithm will come within one instruction time unit of the theoretic lower bound. Experiments show the improvement can reach about 35% over the one-level partitioning algorithm with memory constraints and more than 80% over the traditional hardware prefetching scheme or list scheduling.

Section 2 will introduce the terms and basic concepts used in this paper. Section 3 describes the idea of two-level partitioning. The next section presents the algorithms used to implement the two-level partition construct, while Section 5 is a comparison of the new technique with a number of existing approaches. A summary section that reviews the main points concludes the paper.

2 Basic Idea

Our technique can be applied to the uniform nested loop which exists in a lot of DSP applications. Moreover, more general linear index loops can be uniformized first, then use our technique to optimize the schedule. For a *linear parameterized recurrence equation* in the form of $U(z) = f(V(I(z - \vartheta_z), ...)$ with $\vartheta_z = A \cdot z + B \cdot p + C$ (where z is the loop index vector, p is the size parameter vector of the equation, A, B, C are all integer constant matrices), it is non-uniform if the number of different dependences depends on size parameter p. A system of linear recurrence equations is a finite set of equations in the above form. It has been proved that any system of linear recurrence equations can be transformed into the normal form in which all the computation equations are fully indexed, and have the same index dimension. In the normal form, all the equations can be classified into input equations, computation equations and output equations. The uniformization of computation equations is try to find

a finite set of integral vectors (independent on p), $A_1, A_2, \ldots, A_t$, such that any ϑ_z can be expressed as a non-negative integral combination of these vectors, i.e., $\vartheta_z = \sum_{j=1}^{t} \alpha_j \cdot A_j, \quad with \quad \alpha_j \in N$. With a set of such vectors, the linear recurrence can be transformed into a set of uniform recurrences. The detailed analysis of uniformization can be found in [7] and [19].

The technique in this paper is majorly used to optimize the schedule of the computation recurrences. Another work done by A. Agarwal et al [1] and F. Rastello et al [21] concentrates on the input and output equations. They use data footprint to capture the combined set of data accesses made by references within each uniformly intersecting class. A data partition size is selected to minimize the communication in multiprocessors with caches. In DSP application, the major overhead of memory references is caused by the intermediate data in computation equations.

2.1 Loop nest representation

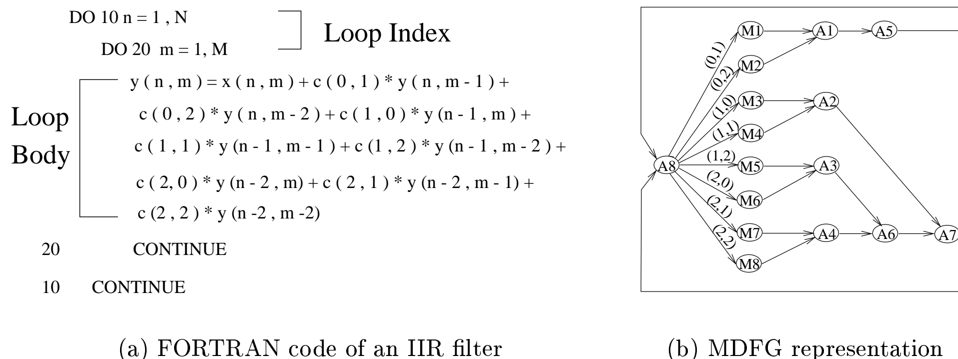
In a uniform nested loop, an *iteration* is the execution of the loop body once. It can be represented by a graph called *multi-dimensional data flow graph* (MDFG).

Definition 1 *A multi-dimensional data flow graph (MDFG) G =(V, E, d) is an edge-weighted directed graph, where V is the set of computation nodes,* $E \subseteq V \times V$ *is the set of dependence edges, and d is a function from* E *to* Z^n *representing the multi-dimensional data dependence (delay vector) between two nodes, where n is the depth of the nested loop.*

It is worthwhile to note that MDFG can be thought of as the computationally finer-grained description of data dependence than the Data Dependence Graph (DDG) or Statement Dependence Graph (SDG). The node in MDFG represents an ALU computation, thereby consumes one basic ALU computation time unit. On the contrary, a node in DDG or SDG represents a statement, which may consume uncertain number of ALU computation time units depending on the complexity of the statement. Therefore, it is much easier to use MDFG instead of DDG or SDG when scheduling the computation. Furthermore, Most DSP filters, such as IIR, two dimensional filter, etc, can directly modeled by MDFG.

Consider the example in Figure 1. The FORTRAN code derived from the IIR filter equation is shown in Figure 1(a). An equivalent MDFG is presented in Figure 1(b). The graph nodes represent two kinds of operations: nodes denoted by an 'A' followed by an integer are additions, while nodes labeled 'M' followed by an integer represent multiplications. Notice that dependence vectors are represented by pairs (d_x, d_y),

where d_x corresponds to the dependence distance in the Cartesian axis representing the outermost loop and d_y corresponds to the innermost loop.



(a) FORTRAN code of an IIR filter (b) MDFG representation

Figure 1: The MDFG representation of IIR filter

Iterations are identified by a vector $\vec{i}$, equivalent to the nested loop index and starting from $(0, 0, \ldots 0)$. An edge with delay $(0, 0. \ldots , 0)$ represents an intra-iteration dependence, and an edge with non-zero delay $(d(e))$ represents an inter-iteration dependence. We will use delay dependence to represent the dependence among different iterations, i.e., the non-zero edge's weight. The execution of the entire loop will scan over all loop indices. All iterations constitute the *iteration space.* Each iteration is a node in the iteration space. Each inter-iteration dependence is an edge between the corresponding nodes in the iteration space.

2.2 Partitioning the iteration space

Regular execution of nested loops proceeds in either a row-wise or column-wise manner until the end of the row or column is reached. However, this mode of execution does not take full advantage of either the reference locality or the available parallelism, since dependences have both horizontal and vertical components. The execution of such structures can be made to be more efficient by dividing the iteration space into regions called *partitions* that better exploit spatial locality. This observation can be applied to both the first and second level memories.

Provided that the total iteration space is divided into partitions, the execution sequence will be determined by each partition. That is to say, each partition is executed in turn from left to right. Within each partition, iterations are executed in row-wise order. At the end of a row of partitions, we move up to the next row and continue from the far left in the same manner.

Assume that the partition in which the loop is executing is the *current partition*. Then the *next partition* is the partition adjacent on the right side of the current partition along the X-axis. The *second next partition* is adjacent to and lies on the right side of the next partition, with the definitions of *third next partition*, *fourth next partition*, etc, similar. The *other partitions* are all partitions except those on the current row. In the two-level partition algorithm, the iteration space will be partitioned on two levels. The *first level partition* consists of a number of iterations, and the *second level partition* is made up of a number of first level partitions.

We use two *partition vectors*, P_x and P_y, to identify a parallelogram as the partition shape. Assume without loss of generality that the angle between P_x and P_y is less than $180°$, and P_x is clockwise of P_y. Then the partition shape and size can be denoted by the direction and the length of these two vectors. To satisfy the partition execution order, there cannot exist the data dependence cycle between different parititons. Two partition vectors must sastify the follow property to maintain the data dependences.

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.*

2.3 Architecture model

Our technique is designed for use in a system containing a processor with multiple ALUs and memory units. The first level memory is also located in the processor. The second and third level memories are off-chip memories. The first level memory has the tightest memory size constraint and the fastest access speed. The second level memory has medium memory size and access speed. The third level memory has the largest memory size and slowest access speed. This architecture is similar to the real system with L1 cache, L2 cache and main memory. Our technique is to load data into the first level memory before its explicit use so that the overall cost of accessing data can be minimized. Therefore, overlapping the ALU computations and the memory accesses will lead to a shorter overall execution time. The goal of our algorithm is to tolerate the memory access time by overlapping the ALU computations as much as possible.

Our scheme is a software-based method in which some special prefetching instructions are added to the code when it is compiled. When the processor encounters these instructions during program execution, it will pass them to the special hardware *memory units* for handling. The function of the memory unit is the same in both two levels: get the data ready before the lower level cache needs to reference them. Two

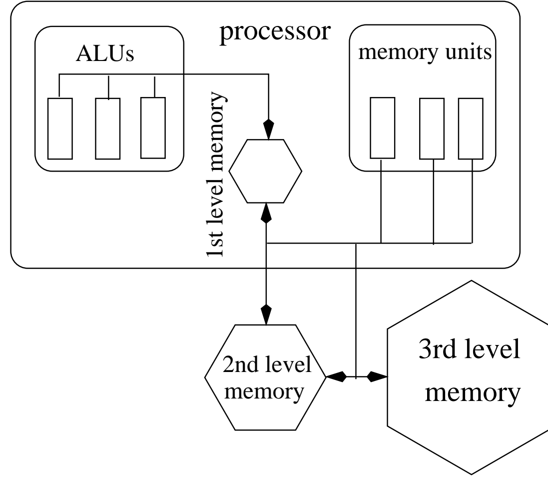


Figure 2: The architecture model

types of instructions, *prefetch* and *keep*, are supported by memory units. The *keep* instruction keeps the data in the cache for use during a later partition's execution. In the first level memory, depending on the partition size and different delay dependences, the data will need to be kept for different amounts of time. The advantage of using *keep* is that it can eliminate the time wasted for unnecessary data swapping.

In the first level memory, some data may need to be kept for different amount of time. We use the number of partitions to denote this time. It is reflected in the names of different kinds of keep operations. If a delay dependence starts from an iteration in the current partition and terminates in the mth next partition, a *keep_m* memory operation is used to keep this data in the first level memory for the time of m partition executions . Delay dependences that go into other partitions result in the use of prefetch memory operations to fetch data in advance. In the second level, only keep_1 operations exist.

Note that the lower level memories in the architecture model cannot be seen as pure caches, because issues such as cache consistency and cache conflict are not considered here. In other words, the lower level memory can be thought as a fully associative cache with some memory units connected to it.

Corresponding to ALU and memory units, there are ALU and memory schedules in one-level partition algorithm [26]. To form the ALU schedule, Multi-dimensional rotation scheduling algorithm [17] is used to obtain the schedule of a single iteration. Then this schedule is simply duplicated for each iteration in the partition. The memory schedule is formed by considering all the memory opertions in the entire partition. The partition size is select to balance the ALU and memory schedules such that the long memory latency is effectively tolerated.

3 Two-level partition

In reality, contemporary computer systems always have more than two levels of memory. It is necessary to study scheduling with more levels of memory. In our model, we use three levels of memory, not only because it is the most common case, but because we can apply the same idea to a memory hierarchy with more than three levels.

We first consider the situation with dynamic scheduling (FIFO) in the second level memory. Assuming that the access time from the third to first level memories is 10 time units, that from the second to first level memories is 3 time units and that from the third to second level memories is 7 time units, Table 1 gives the average access time of the first level memory for two different benchmarks under different iteration space sizes. This access time includes the time from the third to first level memories and from the second to first level memories. In the experiment, the first level memory size is the memory requirement, while the size of the second level memory is ten times larger than that of the first. Using the relative second level memory size can make us concentrate on the effect of more memory levels and ten times larger is a reasonable assumption.

In the table, we use two different benchmarks (DPCM and WDF) and two different algorithms (one-level partition algorithm and pen-tiling algorithm, which is introduced in the experiment section) to decide the first level partition. The table lists the partition size p_{size}, denoted by two vectors, and the first level memory requirements m_{req} for each benchmark and algorithm with the benchmark's name. The average access time is listed for each iteration space. For example, 30×30 in the table indicates the two dimensional loop with each dimension equal to 30.

one-level partition algo			Pen-tiling algo		one-level partition algo			Pen-tiling algo	
DPCM	50×50	6.211	DPCM	6.212	WDF	30×30	7.097	WDF	8.548
$p_{size} = (4,0)$	100×100	7.884	$p_{size} = (6,0)$	8.427	$p_{size} = (1,0)$	50×50	8.819	$p_{size} = (2,0)$	9.456
$\times(-16,8)$	400×400	9.872	$\times(-12,6)$	9.904	$\times(-12,4)$	100×100	9.709	$\times(-6,2)$	9.860
$m_{req} = 267$	800×800	9.968	$m_{req} = 265$	9.976	$m_{req} = 50$	400×400	9.981	$m_{req} = 48$	9.991

Table 1: The average access time under different iteration space sizes

If the dynamic scheduling in the second level memory can take good advantage of the second level memory, we expect to see an average access time close to 3, which is the access time from the second to first level memories. However, from the data in the table, we can see that having more memory levels does not result in a substantial improvement when the iteration space is large. The reason is that the dynamic algorithm can not predict the data access sequence. In addition, the data have already been

swapped out to the third level memory by the time they are needed. This demonstrates that dynamic scheduling cannot make use of the information obtained during compilation to improve data locality and reduce access time. This phenomenon leads us to think about partitioning in the second level memory as well as in the first level memory.

The objective of the second level partition is to prefetch some data from the third to the second level memories and keep some data in the second level memory for the near future use. Therefore, whenever the first level memory wants to access these data, it can always find them in the second level memory, thus eliminating the time used to fetch data from the third level memory.

3.1 The different delay dependences

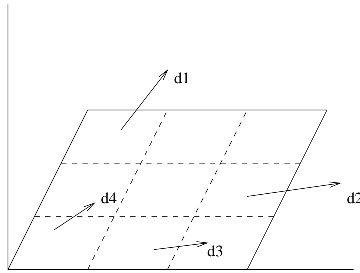


Figure 3: The different kinds of delay dependence for the second level memory

When considering the second level partition, we can treat the first level partition as the basic integral unit. Then, the second level partition will be the combination of a number of first level partitions. For instance, the second level partition size in Figure 3 is 3×3 first level partitions.

In the second level, the *current partition*, *next partition* and *other partition* have the same definitions as in the first level. In the case of two level partitions, there will be four kinds of delay dependences which need to be treated differently when scheduling the loop. They are indicated by d_1, d_2, d_3, and d_4 in Figure 3, the dotted lines delimiting first level partitions here. d_1 is a delay dependence that goes to one other partition from the current partition in both the first and second levels. For this delay dependence, the prefetch operation is needed in both levels. In the second level, d_3 is an intra-partition delay dependence, while it goes to the next partition in the first level. It only needs keep operations in the first level partition schedule and has no effect on the second level partition schedule.

Delay dependence d_4 is an intra-partition delay dependence in the second level but goes to one other partition in the first level. Therefore, it will be treated with a prefetch operation in the first level and a

keep operation in the second level in order to retain the corresponding data for near future use. Moreover, some write-back operations are needed to put back the data corresponding to this delay dependence into the second level memory.

The delay dependence d_2 goes to the next partition in both first and second levels. Due to the execution sequence (the loop is executed by the first level partition sequence along the x-axis until the right boundary of the second level partition is reached, then continues from the left boundary of the second level partition along the next row of first level partitions), this delay dependence gives rise to some complications in the schedule. In the first level partition it cannot be kept in the first level memory as usual, since it will not be needed until the execution reaches a partition in the next second level partition. Instead, from the standpoint of the first level partition schedule, the data will be prefetched from the second level memory before it is needed. Thus, we will have the following definition.

Definition 2 *A* **boundary partition** *is a first level partition with at least one delay dependence going to the next partition in the second level.*

In a boundary partition, some or all of the keep operations in an ordinary partition transform into prefetch operations. The number of keep operations which become prefetch operations depends on the relation between the first level partition size and the distance of the delay dependence. For instance, if there are only keep_1 operations in the ordinary first level partition, the boundary partitions are found only in the rightmost column of first level partitions. If a keep_2 operation also exists, then the boundary partition also includes the next right-most column of first level partitions. However in a first level partition of this column, only keep_2 operations become prefetch operations. After deciding which keep operations will become prefetch operations, the boundary partition schedule lengths can be determined accordingly. In the second level partition, the data corresponding to d_2 will be also prefetched from the third level memory to the second level for use in the next second level partition.

3.2 The write back operation

As about write back operations which write data back to the higher level memory, they can be considered as being hidden in the prefetch operations. We first prove that the number of write-back operations is the same as the number of prefetch operations in the first level partition.

Lemma 1 *In the first level partition, the number of the write-back operations is always the same as the number of prefetch operations.*

Proof:

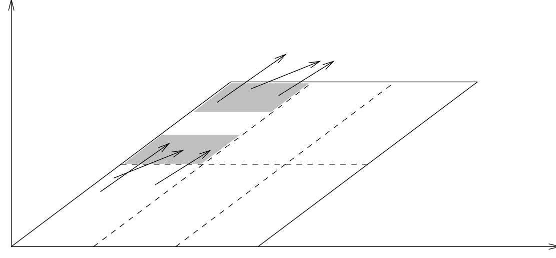


Figure 4: The area of the prefetch and writing back operation

In Figure 4, the shaded upper region denotes the area of write-back operations and the shaded lower region represents the area of prefetch operations. If there is a delay dependence to another partition in the shaded upper region, the corresponding data need to be written back to the higher level memory. At the same time, there must exist the same delay dependence coming into the shaded lower region from another partition, which means the corresponding data need to be prefetched in the previous partition. Therefore, they have identical number of operations. □

As discussed in [26], one memory operation (prefetch or keep) consumes two memory locations. One is used for the datum needed in the current partition, the other is used for next partition's computation. Provided that the numbers of write back and prefetch operations are identical, it is possible to treat them as an integrated operation. Next we will prove that the memory consumption is still two for such an integrated operation. At the same time, how to integrate these two operations into one operation is illustrated.

Theorem 2 *Every write back operation can be combined with a prefetch operation into an integrated operation, with two memory locations required.*

Proof: From the above proof, we know that there exists both a write back operation and a prefetch operation to each delay dependence going into another partition. We can treat them with one operation by the following method.

1. In the current partition, for each delay dependence going into one other partition, we use one memory location to store data prefetched for the use of the next partition, and one memory location to store data which need to be written back.

2. In the next partition, we first write one datum back to the higher level memory, then use this location to store the prefetched data. At the same time, whenever a datum to be written back is computed, a datum prefeched by the previous partition must have been consumed. This location can be used to store the computed data.

□

From this theorem, we can simply double the prefetch time to consider the write back operation. The model and algorithm can still be useful. The following theorem defines the relationship between the number of write back operations and prefetch operations in the second level partition.

Theorem 3 *The number of prefetch operations and write back operations, which are used to write data back to the third level memory, are identical. These two operation can be regarded as one integrated operation when considering the second level partition schedule.*

Proof: It is easy to verify this theorem based on the proof of Theorem 2. □

3.3 Two level partition scheduling

With all the pieces now in place, we perform two-level partition scheduling by considering both levels separately. The first level partition schedule consists of two parts as one-level partition technique [26], as shown in Figure 5(a). However the prefetch part is made up of integrated prefetch and write back operations as discussed in the last subsection.

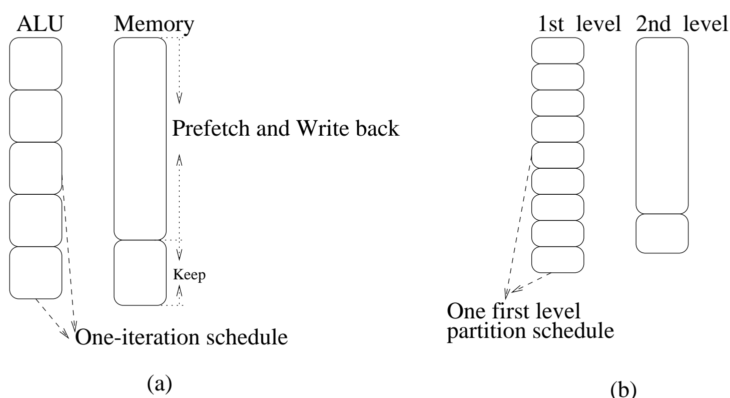


Figure 5: (a) The first level partition schedule. (b) Two level partition schedule

Because the keep operation can only been issued after the corresponding computation, we always issue prefetch operations ahead of all keep operations in the memory part schedule. Also, due to this

dependence between keep operation and the corresponding datum's finishing time, the length of the memory part is always longer than the ALU part. and some idle time may exist in the memory part. Therefore, the schedule length for one partition in the first level L_{first} satisfies the inequality.

$$L_{first} \geq \#prefetch_{first} \times T_{2-1} + \#keep_{first} \times T_{keep}$$

where $\#prefetch_{first}$ is the number of prefetch operations per first level partition, T_{2-1} is the time cost to load data from the second to first levels, $\#keep_{first}$ is the number of keep operations per first level partition, and T_{keep} is the time cost per keep operation. The boundary partition schedule length is different. Its value will depend on how many keep operations change into prefetch operations.

In the second level, there exist only two kinds of operation: *prefetch* to fetch those data corresponding to data dependences like d_1 and d_2 in Figure 6 and *keep* to keep the data corresponding to the data dependences like d_4 in Figure 6. The two-level partition schedule is shown in Figure 5(b). The length of the *first level schedule* is the summation of all the first level partition schedules in a second level partition and the length of the *second level schedule* is the sum of the lengths of the prefetch and the keep parts.

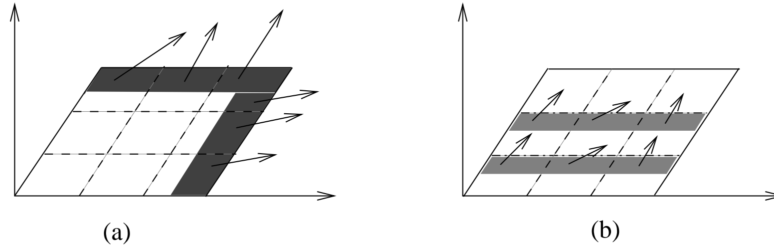


Figure 6: (a) Prefetch operations in the second level partition. (b) Keep operation in the second level partition

The length of the prefetch part can be calculated from the area shown in Figure 6(a). The length of the keep part can be calculated from the area shown in Figure 6(b). The overall length is

$$\begin{aligned} L_{second} &= L_{prefetch} + L_{keep} \\ &= \#prefetch_{second} \times T_{3-2} + \#keep_{second} \end{aligned}$$

where $\#prefetch_{second}$ is the number of prefetch operations of the second level partition, T_{3-2} is the time cost to load data from the third to the second level, and $\#keep_{second}$ is the number of keep operations in the second level partition.

4 Two Level Partition Scheduling algorithm

Knowing the constitution of the first and second level schedules, we need to find the relation between these two schedules in order to achieve the optimal average schedule length. From the analysis above, the first level schedule can be determined independently. Therefore, the lower bound of the average schedule length is the average schedule length of the ordinary first level partition, which equals the one-level partition schedule length where the prefetch time can be regarded as the fetching time from the second level to the first level. We now wish to determine the second level partition size which will make the overall schedule optimal.

4.1 The property of the operation amount

The first level schedule length depends on the ALU schedule, as well as the number of prefetch and keep operations in the first level. On the other hand, the second level schedule length only depends on the prefetch and keep operations in the second level. The following properties show the relationship between the numbers of operations in the first and the second level partitions.

The first level partitions in a second level partition can be classified into ordinary partitions and boundary partitions. Assume that the region size consisting of only ordinary partitions is $x_1 \times y$ and that consisting of only boundary partitions has size $x_2 \times y$. Let the number of prefetch operations in the first level be $\mathsf{pre}_{\mathsf{first}}$ and the number of keep operations be $\mathsf{keep}_{\mathsf{first}}$. The number of prefetch and keep operations in the second level partition can be calculated as follows.

Property 2 *In the region consisting of only ordinary partitions, the number of prefetches in the second level partition is* $\mathsf{pre2}_{\mathsf{ord}} = x_1 \times \#\mathsf{pre}_{\mathsf{first}}$ *and the number of keep operations is* $\mathsf{keep2}_{\mathsf{ord}} = x_1 \times (y - 1) \times \#\mathsf{pre}_{\mathsf{first}}$.

In the region consisting of only boundary partitions, there exist three sub-regions for a given delay dependence d, as shown in Figure 7. In the figure, the dotted lines delimit the first level partitions. The three different sub-regions are designated in the figure.

Definition 3 *In the region consisting of only boundary partitions, the* **top region** *for a delay dependence is the region from which this delay dependence starts and goes into one other second level partition. The* **boundary region** *for a delay dependence is the region from which this delay dependence starts and goes*

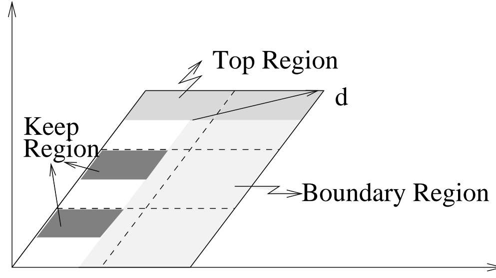


Figure 7: The different sub-regions in the boundary partitions

into the next second level partition. The **keep region** *for a delay dependence is the region from which this delay dependence starts and goes into one other first level partition, while remaining in this second level partition.*

The one-level partition scheduling algorithm guarantees that the P_y projection of a delay dependence will not be longer than the length of the first level partition size along the P_y direction. That is to say, the top region is only included in the top row of the first level partitions in a second level partition. The number of integer points in these regions is the number of corresponding operations for this delay dependence. These numbers can be calculated using the same parallelogram idea given in [26]

Property 3 *For a given delay dependence, the total number of integer points in the top region and the boundary region is the number of prefetch operations in the boundary partitions. The number of integer points in the keep region is the associated number of keep operations in the boundary partitions.*

4.2 The balanced two levels partition schedule

Since the lower bound of the length of the two-level partition schedule is determined by the first level schedule, we should make the first level schedule longer than the second level schedule to achieve good performance. Analyzing the relationship between the first and second level schedules, we first give the following definition.

Definition 4 *Given the first level partition schedule and a second level partition size, a* **balanced two-level partition schedule** *is a schedule in which the first level schedule is longer than the second level schedule and the schedule length difference between the first and second level is minimal.*

The following theorem proves the existence of such a balanced two level partition schedule.

Theorem 4 *If the schedule of the first level partition is determined, a second level partition size that makes the two level partition schedule a balanced two level partition schedule can always be found.*

Proof: Assume the size of the second level partition size is $x \times y$. We first prove that the schedule length difference between the first level and the second level will increase as the value of x is enlarged.

Suppose that the second level partition size increased from $x \times y$ to $(x+1) \times y$, where the value of x is large enough that the number of boundary partitions will not increase when x is enlarged. The length of the first level schedule increases by more than $(\#pre_{first} \times T_{21} + \#keep_{first}) \times y$, while that of the second level schedule length increases by more than $\#pre_{first} \times T_{32} + \#pre_{first} \times (y-1)$. The first item is always larger than the second item as long as $y \geq \frac{T_{32}-1}{T_{21}-1}$.

We now prove the existence of a second level partition size which can make the first level schedule longer than the second level schedule. Assume the size of a region consisting of only boundary partitions is $z \times y$. Then the first level schedule length satisfies:

$$L_{first} > (\#pre_{first} \times T_{21} + \#keep_{first})xy$$

The second level schedule length satisfies:

$$L_{second} < \#pre_{first}(x-z)T_{32} + \#pre_{first}xy + (\#pre_{first} + \#keep_{first})T_{32}zy$$

Therefore,

$$L_{first} - L_{second} > \#pre_{first}(T_{21}xy - T_{32}x - xy - T_{32}zy) + \#keep_{first}(xy - T_{32}zy)$$

When the following is satisfied, we can guarantee that the first level schedule length is longer than the second level schedule length.

$$\begin{cases} x & \geq & T_{32}zy \\ y & \geq & \frac{T_{32}+1}{T_{21}-1} \end{cases}$$

In fact, these are rather loose constraints for finding a feasible size.

□

4.3 The calculation of the memory requirement

In the two level partition schedule, we need to calculate both the first and second level memory requirements.

Theorem 5 *The first level memory requirement* $\mathtt{mem1}_{\mathtt{req}}$ *is calculated by the following equation:*

$$\mathtt{mem1}_{\mathtt{req}} = \mathtt{mem}_{\mathtt{inter}} + \sum \#\mathtt{keep_n} \times (\mathtt{n}+1) + 2 \times \#\mathtt{pre}_{\mathtt{first}}$$

In the equation, $\mathtt{mem}_{\mathtt{inter}}$ *is the memory requirement for the intermediate data in the first level partition,* $\#\mathtt{keep_n}$ *is the number of keep-n operations, and* $\#\mathtt{pre}_{\mathtt{first}}$ *is the number of prefetch operations.*

Proof:

From Theorem 2, the number of write-back operations equals the number of prefetch operations, and their integrated operation only requires 2 memory locations. Thus in the first level memory, the memory requirement consists of three parts: those used to store prefetched and written back data, those used to store kept data, and those used to store intra-partition intermediate data. Based on the knowledge of the memory consumption for each kind of memory operation introduced before, the above equation is true. □

We can use the above theorem to derive the memory requirement for the first level memory, in which the calculation of $\mathtt{mem}_{\mathtt{inter}}$ is the same as in the one-level partition algorithm.

Theorem 6 *The memory requirement for the second level* $\mathtt{mem2}_{\mathtt{req}}$ *is calculated by the equation*

$$\mathtt{mem2}_{\mathtt{req}} = 2 \times \#\mathtt{pre}_{\mathtt{second}} + \#\mathtt{keep}_{\mathtt{second}},$$

where $\#\mathtt{pre}_{\mathtt{second}}$ *represents the number of prefetch operations in the second level, and* $\#\mathtt{keep}_{\mathtt{second}}$ *is the number of keep operations.*

Proof: From the above analysis we know that the second level partition schedule consists of only two parts, one consisting of prefetch operations, the other of keep operations. The memory requirement also depends on these two parts. The keep operations in the second level partition are different from those in the first level partition. Corresponding data are only kept in the second level memory for the current partition to use; hence the lifetime is only one second level partition. This keep operation only needs one memory location to store the data. Thus we can directly obtain the above equation using previously derived information. □

Combining the above theorem and two properties in Section 4.1, the memory requirement for both the first and second level memories can be calculated.

In choosing the second level partition size, we select the size that can make the two level partition schedule a balanced two level partition schedule. This is the tradeoff between the performance improvement and the memory consumption. After achieving a balanced two level partition schedule, increasing x can still bring some performance improvement. On the other hand, the memory requirement is increasing at a much larger rate than the performance improvement. For instance for an IIR filter, when there is some first level memory constraint, the balanced two level partition schedule requires a second level partition size of 7×10. If the second level partition size is increased by one along the P_x direction, the memory requirement increases by 12.94%, while the schedule improvement increases by only 0.058%. Therefore it is more reasonable to adopt a second level partition size which can generate a balanced two level partition schedule.

We have demonstrated that we can always find such an x to generate a balanced two level partition schedule as long as y is large enough. Therefore, there exist different such x associated with different y. The following algorithm gives the method to calculate the minimum second level memory requirements and the corresponding second level partition size.

In this algorithm, the symbol $\#pre_1$ denotes the number of prefetch operations in one first level partition, $keep_{1_n}$ is the number of keep-n operations in one first level partition; T_{3-2} and T_{2-1} are the fetching times from the third level memory to the second level memory and from the second level memory to the first level memory, respectively; and lp_{ord} is the schedule length of the ordinary first level partition.

Algorithm 1 Calculate the minimum second level memory requirement and the corresponding second level partition size

Input: A given first level partition schedule and the corresponding set of delay dependences
Output: The minimum second level memory requirement and the associated second level partition size

1. Use the delay dependence set to determine the width w of the region consisting of only boundary partitions
2. Compute the amount of keep operations $\#keep_{keep}$ and prefetch operations $\#pre_{bound}$ in the bottom row of this region
3. Derive the function $f(y)$ to calculate the number of keep operations and $f'(y)$ to calculate the number of prefetch operations.

$$\begin{cases} f(y) = \#keep_{keep} \times (y-1) \\ f'(y) = \#pre_{bound} \times (y-1) + \sum n \times \#keep_{1_n} + \#pre_1 \times w \end{cases}$$

4. Compute $l(y)$ which is the summation of all boundary partition schedule lengths.
5. Let the first level schedule length equal to the second level schedule length and derive the expression for the second level memory requirement mem_{req2}

$$\begin{cases} lp_{ord}(x-w)y + l(y) = \#pre_1 T_{3-2}(x-w) + \#pre_1(x-w)(y-1) + f'(y)T_{3-2} + f(y) \\ mem_{req2} = 2\#pre_1(x-w) + \#pre_1(x-w)(y-1) + 2f'(y) + f(y) \end{cases}$$

6. Calculate x for a given y using the first equation.
7. Substitute x into the second equation. Calculate the value of y which make mem_{req2} minimum.
8. Calculate the value of corresponding x and the second level memory size.

In Algorithm 1, after the value of y which can lead to the minimum second level memory requirement is determined, we calculate an x value that makes the first level schedule length equal to the second level schedule length using this y. Then $\lceil x \rceil$ is chosen as the actual x. It can be seen from the proof of Theorem 4 that the difference between the first level and second level schedules is an increasing function of x. The second level partition size calculated using Algorithm 1 guarantees a balanced two level partition schedule.

4.4 Algorithm

From the above discussion, the balanced two level partition schedule is a good point to select the second level partition size. It can provide us both the optimal average schedule length and second level memory requirement. After knowing the partition size in both levels, we can generate the schedule for both levels using our previous knowledge. Note that there is a difference between the arrangement of memory operations in the first and second levels. In the first level partition, we can only issue keep operations after the corresponding data are ready, so there may be some idle time in the memory part. In the second level partition, we can arrange the keep operations as soon as the corresponding data have been written back. Since there is no keep operation in the top row of the second level partition, there will be no such idle time in the second level memory.

Algorithm 2 The two-level partition schedule

Input: An MDFG, the first level memory constraint

Output: The two level partition schedule

1. Retime the MDFG to get a compact schedule for one iteration. //Using multi-dimensional rotation scheduling algorithm. see section 2.1.
2. Based on the delay dependences in the retimed MDFG, determine the first level partition shape. // see section 2.1
3. Based on the first level memory constraint and the first level partition shape, calculate the first level partition size. //Using one-level partition algorithm, see section 2.1
4. Create the first level partition schedule.
5. Use this first level partition schedule and Algorithm 1 to compute the second level partition size. //see section 4.3
6. Calculate the corresponding second level memory requirement. //see section 4.3
7. Create the second level schedule.

To create the first level schedule, we duplicate the retimed schedule of one iteration in order to construct the ALU schedule. In the memory part, we first arrange prefetch operations, then keep operations. In the second level schedule, the prefetch and keep operations are arranged in turn as we have mentioned above.

In this paper, we illustrate the two-level partition scheduling with two dimensional iteration space, because it is the most general case in multi-dimensional DSP applications. It is important to note that

this scheme can be extended to more dimensional iteration space with little modification. All the concepts and algorithms under more dimensional case are easily deduced. Take 3-dimensional loop as an example, the partition will become a cube whose each face is a parallelogram. A partition is delimited by three partition vectors: P_x, P_y and P_z. All data dependences must lie inside these partition vectors to prevent the dependence cycle between different partitions. The partition execution sequence is along X axis first, then Y axis and Z axis. The identification of memory operations and the algorithm still applied. The partition size which can achieve the balanced partition schedule is still the preferred choice.

5 Experiment

In this section, the effectiveness of the two-level partition technique is evaluated by running a set of DSP benchmarks. In the experiments we assume that the time to load data from the third to second level memories is 9 clock cycles, from the second to first level memories is 3 clock cycles, and from the third to first level memories is 12 clock cycles. We compared six different schemes. They are two-level partition algorithm, one-level partition algorithm in [26], pen-tiling algorithm in [18], PSP scheduling in [9], list scheduling and hardware prefetch scheme, respectively. Pen-tiling algorithm presents a scalable criterion to define optimal tiling. This criterion, related to the communication to computation ratio of a tile, only depends upon its shape, not its size. The pen-tiling algorithm solves a combinatorial problem to find a basic tile, then determines the final tile size depending on the first level memory size constraints. It assumes a two-level memory hierarchy in the system. In the experiments, we loose the memory size constraints for the pen-tiling algorithm to demonstrate that our partition method, which balances the computation and communication, can get the better result even under the tighter memory size constraints. PSP scheduling attempts to balance the computation and communication. Nevertheless, the situation with the first level memory size constraints was not considered in PSP scheduling. This algorithm can obtain the same results as our algorithm under no first level memory size constraints, but can not get feasible results when this constraint is imposed. Therefore, their experimental results are not shown in the table.

List scheduling is the most traditional algorithm. It is a greedy algorithm which seeks to schedule a MDFG node as early as possible while satisfying the data dependence and resource constraints. In our experiment, we use list scheduling to schedule the ALU operations, but the memory is not partitioned. In hardware prefetching scheduling, we use the model presented in [4]. In this method, to take advantage of

the data locality, the next block in the higher level memory is also loaded whenever a block is loaded from the higher level to the lower level memories. We use the multi-dimensional rotation scheduling algorithm to arrange the computations in the ALU schedule. Furthermore, the prefetching operations are added in the memory part. However, no partition is considered here.

The first table presents the results without the memory constraint in the first level. In this table, we use the same first level partition size as shown in the table in both the one-level and two-level partition scheduling algorithms. Also, they have the same average schedule length. The other two tables describe the results with memory constraints. In the last two tables, the relative memory constraints for all benchmarks are used due to the large differences among their memory requirements.

Benchmark	Two-level		One-Level			pen-tiling			list		hardware pre	
	m_{req2}	size2	size1	len	m_{req1}	size1	len	m_{req1}	len	ratio	len	ratio
IIR	1159	6×2	4×8	**6.031**	358	7×7	6.021	346	40	85%	37	83.7%
DPCM	3982	5×3	12×10	**4.008**	840	11×11	4.008	801	27	85.16%	22	81.78%
WDF	486	6×2	4×5	**4.05**	129	4×4	4.625	128	18	77.5%	12	66.25 %
Floyd	477	3×2	7×4	**6.000**	216	5×5	6.000	227	33	81.82%	22	72.72%
2D	491	3×2	3×5	**12**	241	4×4	12	260	65	81.54%	60	80%

Table 2: Experimental results without memory constraints assuming $T_{prefetch}$ =12

Benchmark	One-level			pen-tiling			Two-level				
	size1	m_{req1}	len	size	m_{req1}	len	size1	m_{req1}	size2	m_{req2}	len
IIR	1×6	175	10	4×4	193	10.125	5×2	141	7×10	3825	**6.129**
DPCM	3×9	416	5	7×7	427	5.347	12×4	381	7×8	9230	**4.027**
WDF	1×5	74	5	3×3	89	6.111	4×2	75	8×5	936	**4.156**
FLOYD	2×3	109	8.333	3×3	128	7.889	4×2	89	6×4	742	**6**
2D	1×4	112	14	2×2	128	24	3×1	125	4×13	2080	**12.5**

Table 3: Experimental results with about 1/2 of original size

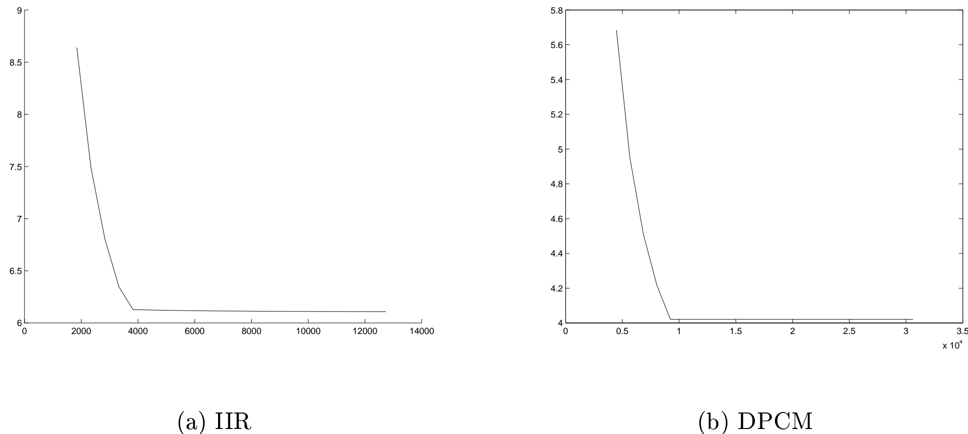
Benchmark	One-level			pen-tiling			Two-level				
	size1	m_{req1}	len	size	m_{req1}	len	size1	m_{req1}	size2	m_{req2}	len
IIR	1×3	85	15.67	2×2	83	19.75	2×2	74	13×6	1435	6.404
DPCM	2×6	216	7.083	2×2	205	9.062	4×4	205	15×8	6952	4.188
WDF	1×3	44	8	2×2	52	9	1×3	44	19×3	462	4.645
FLOYD	1×2	43	12.5	2×2	70	12	1×2	37	18×4	592	6.417
2D	1×2	65	25	2×2	128	24	1×2	65	11×4	755	12.615

Table 4: Experimental results with about 1/4 of original size

In our experiments, the five benchmarks used are *Infinite Impulse Response filter, Differential Pulse-Code Modulation device, Wave Digital filter, Floyd-Steinberg algorithm* and *Two Dimensional filter*. They are represented by "IIR", "DPCM", "WDF", "Floyd" and "2D", respectively, in all tables. All the "size1" columns list the partition size of the first level in both the one-level and two-level partition algorithms. The "size2" columns list the partition size of the second level in the two-level partition algorithm. All the "len" columns represent the average schedule lengths, and the "ratio" column in

the first table denotes the improvement the two-level partition algorithm can obtain compared with list scheduling and hardware prefetching schemes. The m_{req1} and m_{req2} columns represent the memory requirements of the first and second level memories for each algorithm, respectively.

As we can see from the first table, list scheduling and hardware prefetching scheduling have much worse performance than the other three algorithms. The reason is that, in list scheduling, the schedule is dominated by a long memory schedule, which is far from the balanced schedule. In hardware prefetching scheduling, little compiler-generated information is available. Although the performance differs with data locality, it has on average the same performance as list scheduling. The one-level partition algorithm and pen-tiling algorithm can compete in performance with the two-level partition algorithm in the case without the memory constraints, which is due to the fact that a large first level memory size can efficiently hide the long memory access time. When memory constraints are added, the performance difference is obvious from the last two tables. Moreover, we can see from the last two tables that the two-level partition algorithm is superior to the one-level partition algorithm, and the one-level partition algorithm superior to the pen-tiling algorithm, which illustrates the importance of balancing the different schedules.



(a) IIR

(b) DPCM

Figure 8: The relationship between the average schedule length and the second memory size for IIR and DPCM

In Figures 8(a) and 8(b), the curves depict the relationship between the average schedule length and the second level memory requirement. Two different benchmarks, IIR and DPCM, are used with the first level memory constraints the same as in the second table. The x axis in these two curves is the second level memory size, and the y axis is the corresponding average schedule length. Around the threshold,

it can be seen from the curves that a smaller second level memory size will degrade the performance greatly, while increasing the second level memory size will not result in much performance improvement. The memory size and the corresponding second level partition size obtained from our algorithm is just this threshold to determine the two level schedule.

6 Conclusion

In this paper, a new scheme that can obtain a minimal average schedule length under three levels of memory hierarchy was proposed. This algorithm not only explores the ILP among instructions by using retiming techniques, but combines it with data prefetching in both the first and second level memories to produce high throughput schedules. It uses partitions in both the first and second level memories. Through the study of the properties of the memory requirement and the schedule length in both levels, the algorithm gives a partition shape and size so that the overall minimal schedule length can be obtained. This scheme can take full advantage of the second level memory as compared to dynamic scheduling. Experiments on DSP benchmarks show that our scheme can always produce a minimal average schedule length.

References

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

[2] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software dsms. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 385–392, Jul, 1998.

[3] J. Chame and S. Moon. A tile selection algorithm for data locality and cache intere rence. In *Proc. of the 1999 ACM International Conference on Supercomputing*, pages 492–499, Rhodes, Greece, June 1999.

[4] T. F. Chen. *Data Prefetching for High-Performance Processors.* PhD thesis, Dept. of Comp. Sci. and Engr, Univ. of Washington, 1993.

[5] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

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

[7] Vincent Van Dongen and Patrice Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic array. In *International Conference on Systolic Arrays*, pages 473–482, 1988.

[8] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, 1994.

[9] F.Chen and E.H.-M.Sha. Loop scheduling and partitions for hiding memory latencies. *Proc. IEEE & ACM 12th Intl. Symposium on System Synthesis (ISSS)*, Nov 1999.

[10] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proc. of the 25th Intl. Symp. on Microarchitecture*, pages 102–110, December 1992.

[11] W. Mangione-Smith, S. G. Abraham, and E.S. Davidson. Register requirment of pipelined processors. In *Proceedings of the International Conference on Supercomputing*, pages 260–271, 1992.

[12] N. Manjikian. Combining loop fusion with prefetching on shared-memory multiprocessors. In *Proc. of the International Conference on Parallel Processing*, pages 78–82, 1997.

[13] T. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1), 1998.

[14] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general purpose programs. In *Proceedings of MICRO-28*, pages 243–248, 1995.

[15] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general purpose programs. In *Proceedings of MICRO-29*, pages 243–248, 1995.

[16] 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.

[17] 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.

[18] P.Bouilet, A.Darte, T.Risset, and Y.Robert. (pen)-ultimate tiling. *INTEGRATION, the VLSI Journal*, 17, 1994.

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

[20] J. Ramanujam. Optimal software pipelining of nested loops. In *Proceedings of the International Parallel Processing Symposium*, pages 335–342, 1994.

[21] Fabrice Rastello and Y. Robert. Loop partitioning versus tiling for cache-based multiprocessors. *Tech Report CNRS-INRIA-ENS Lyon n5668*, 1998.

[22] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov, 1994.

[23] M. K. Tcheun, H. Yoon, and S. R. Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proc. of the International Conference on Parallel Processing*, pages 306–313, 1997.

[24] S. Wallace and N. Bagherzadeh. Modeled and measured instruction fetching performance for superscalar microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(6), Jun 1998.

[25] Ching-Yi Wang and K. K. Parhi. Resource-constrained loop list scheduler for dsp algorithms. *Journal of VLSI Signal Processing*, 11(1-2), Oct.-Nov. 1995.

[26] Z. Wang, V.Andronache, and Edwin H.M. Sha. Optimal partitioning under memory constraints for minimizing average schedule length. In *Proc. 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, Nov 1999.

[27] M. E. Wolfe and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), Oct 1991.

[28] M. E. Wolfe, D. E. Maydan, and D. Chen. Combining loop transformation considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-29*, pages 274–286, DEC 1996.

[29] Y. Yamada, J. Gyllenhall, and G. Haab. Data relocation and prefetching for programs with large data sets. In *Proceedings of MICRO-27*, pages 118–127, 1994.