Journal of Systems Architecture 61 (2015) 112–126

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Minimizing write operation for multi-dimensional DSP applications via a two-level partition technique with complete memory latency hiding

Yan Wang [a], Kenli Li [a,*], Keqin Li [a,b]

[a] College of Information Science and Engineering, Hunan University, Changsha 410082, China
[b] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 29 May 2014
Received in revised form 22 January 2015
Accepted 6 February 2015
Available online 16 February 2015

Keywords:
Chip multiprocessor (CMP)
Memory latency
Multi-dimensional DSP application
Partition technique
Schedule length
Scratchpad memory (SPM)
Write operation

ABSTRACT

Most scientific and digital signal processing (DSP) applications are recursive or iterative. The execution of these applications on a chip multiprocessor (CMP) encounters two challenges. First, as most of the digital signal processing applications are both computation intensive and data intensive, an inefficient scheduling scheme may generate huge amount of write operation, cost a lot of time, and consume significant amount of energy. Second, because CPU speed has been increased dramatically compared with memory speed, the slowness of memory hinders the overall system performance. In this paper, we develop a Two-Level Partition (TLP) algorithm that can minimize write operation while achieving full parallelism for multi-dimensional DSP applications running on CMPs which employ scratchpad memory (SPM) as on-chip memory (e.g., the IBM Cell processor). Experiments on DSP benchmarks demonstrate the effectiveness and efficiency of the TLP algorithm, namely, the TLP algorithm can completely hide memory latencies to achieve full parallelism and generate the least amount of write operation to main memory compared with previous approaches. Experimental results show that our proposed algorithm is superior to all known methods, including the list scheduling, rotation scheduling, Partition Scheduling with Prefetching (PSP), and Iterational Retiming with Partitioning (IRP) algorithms. Furthermore, the TLP scheduling algorithm can reduce write operation to main memory by 45.35% and reduce the schedule length by 23.7% on average compared with the IRP scheduling algorithm, the best known algorithm.

1. Introduction

The scaling limitations of uniprocessors [2] have led to an industry-wide turn towards *chip multiprocessor* (CMP) systems. CMPs are becoming ubiquitous in most computing domains. A *scratchpad memory* (SPM) is a small on-chip memory component managed by software, so that application programs can gain automated compiler support. Nowadays, SPM is widely employed in new embedded architectures, including some CMPs, rather than cache as on-chip memory. The SPM is shown to be both performance optimized and power efficient compared to the cache counterpart managed by hardware [8]. One real-world example of CMP employing SPM is IBM's Cell processor [3]. The processor has nine processing cores, and each core embeds a SPM, which is managed by software and applications. A core can prefetch data from the main memory or other cores' SPM. However, because many large-scale applications executed on the architectures which employ SPM as on-chip memory generate numerous write operation, the size of a SPM has become a constraint. The motivation of this paper is to propose a partition technique for multi-dimensional (MD) loops to reduce write operation while achieving full parallelism.

The major contributions of this paper are summarized as follows.

1. We target CMPs embedded with SPMs as on-chip memory (local memory) as our computing platform. We assume that a core contains multiple ALUs, which perform computations.
2. We propose a Two-Level Partition (TLP) algorithm that can (1) improve the loop parallelism to decrease the schedule length; (2) completely hide memory latency to achieve full parallelism; (3) reduce write operation to main memory.
3. In the TLP algorithm, a multi-level partition technique is proposed. Each level partition aims at different goal. The first level partition aims to minimize write operation, and the second level partition based on the first level partition can hide memory latency completely.

* Corresponding author.
E-mail addresses: bessie11@yeah.net (Y. Wang), lkl@hnu.edu.cn (K. Li), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.sysarc.2015.02.001
1383-7621/

The paper has made significant contributions to performance enhancement of MD loops on modern processors with state-of-the-art memory technology. Experimental results show that our proposed algorithm is superior to all known methods, including the list scheduling, rotation scheduling, PSP, and IRP algorithms. Furthermore, the TLP scheduling algorithm can reduce write operation to main memory by 45.35% and reduce the schedule length by 23.7% on average compared with the IRP scheduling algorithm, the best known algorithm.

The rest of the paper is organized as follows. Section 2 reviews related work. A motivating example is demonstrated in Section 3. Section 4 introduces basic concepts and definitions. The TLP scheduling algorithm is presented in Section 5, together with several theorems which discuss how to obtain the partition size. Experimental results are provided in Section 6. Section 7 concludes the paper.

2. Related work

Increasing latency gap between CPU operations and memory accesses makes it increasingly important to develop strategies for cutting the frequency and volume of data transfers between cache and main memory. Since recent research shows that caches result in suboptimal behavior for digital signal processing (DSP) applications with regular data access patterns, many designer adopt software managed on-chip memories for embedded computing system [1,30]. SPM, managed by software and hardware, is similar to conventional on-chip caches in that it resides on-chip and has low access latency and low power consumption. There are a lot of researches to manage data allocation in SPM so that the latency gap between CPU operations and memory accesses can be hidden [7,9–11]. In [7], the authors minimized the total execution time of embedded applications through partitioning the application's scalar and array variables into off-chip memory and SPM. Bai et al. proposed CMSM for software managed multicores by efficient and effective code management [9]. In [8], the authors proposed a highly predictable, low overhead, and yet, dynamic memory allocation strategy for embedded systems with scratchpad memory for reducing energy consumption.

More and more DSP applications involve multi-dimensional loop processing. A MD loop not only takes most computation time and consumes significant energy, but also generates huge amount of write operation. Furthermore, due to the slower increasing of memory speed than that of CPU speed, the slowness of memory hinders the overall system performance. MD problems are of particular interest. These problems (e.g., a large number of DSP applications) are characterized by nested loops with uniform data dependencies. Loop pipelining techniques [4–6] and partition techniques [12,13] are widely used to explore the instruction level parallelism, so that a good schedule with high performance can be obtained,

The partition (tiling) technique divides the entire iteration space into partitions, so that the partitions can be executed one at a time. Agarwal et al. proposed a partition technique to minimize communications in multiprocessors [14]. Wolf and Lam presented a partitioning method to increase data locality [15]. Xue et al. introduced an iterational retiming into partition technique and obtained a new algorithm called Iterational Retiming with Partitioning (IRP) [16,17]. The multi-level partition technique is widely investigated. Wang et al. proposed a two-level partition technique to hide memory latency for DSP applications [18]. Wang et al. presented a multi-level partitioning and scheduling technique to minimize memory access overhead [19]. In this paper, we propose a new partition technique, i.e., the TLP technique. It can minimize write operation while achieving full parallelism.

Most MD loop scheduling methods take the scheduling length into account, since a computation-intensive application usually depends on time-critical sections consisting of loops of instructions. To optimize the execution rate of such an application, designers need to explore the embedded parallelism in repetitive patterns of a nested loop. Chao and Sha proposed a MD retiming method, which does not guarantee to achieve full parallelism and is only applicable to a specific class of multi-dimensional data flow graphs (MDFGs) [21]. Passos and Sha first used a legal MD retiming to successively restructure the loop body represented by an MDFG [20]. However, these retiming techniques cannot break the performance bottleneck caused by the memory speed slower than CPU speed, which has been increased dramatically. Partition scheduling with MD retiming technique is widely used to hide memory latency. The IRP algorithm with iterational retiming technique obtains a schedule with complete memory latency hiding to achieve full parallelism [16]. However, to achieve full parallelism may require increased need of write operation. Therefore, we must obtain better partition size to reduce the number of data needed to be written to main memory while achieving full parallelism.

Many researchers have been addressing the problem of data intensity in MD loop scheduling to reduce write operation. Some techniques optimize the accessing time from hardware perspective. Zhou et al. proposed a durable phase change memory which is applied to main memory [22]. Qureshi et al. proposed a PCM-DRAM hybrid main memory system, in which DRAM is used to absorb the write activities to the PCM [23]. These techniques focus on hardware optimization and can be combined with the software technique proposed in this paper. Passos and Sha proposed a multi-level partition technique to reduce SPM data lost, so that it can minimize memory access overhead for multi-dimensional DSP applications [20]. Chen et al. presented a novel partition technique to reduce memory access overhead [4]. However, these two partition techniques cannot achieve full parallelism. The scheduling length of a memory part may be longer than that of a processor part because of large memory latency. In this paper, we propose a partition scheduling algorithm for MD loops. The algorithm developed in this paper can minimize write operation while completely hiding memory latency to achieve full parallelism.

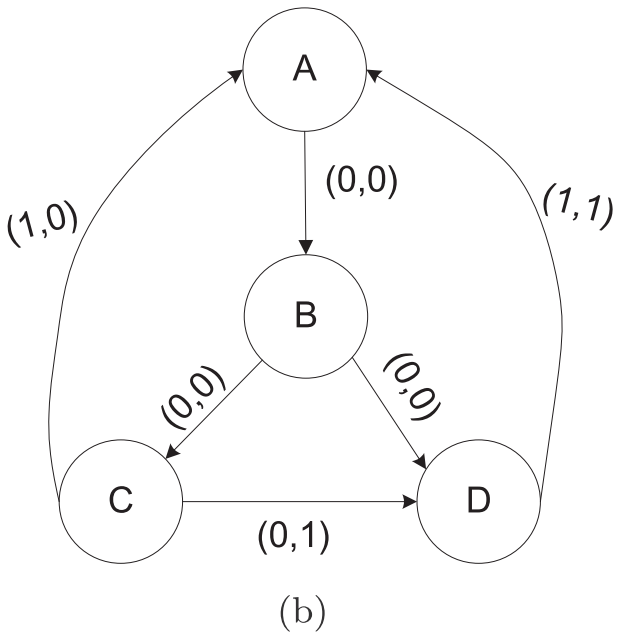
The widening gap between performance of processors and that of memories is the main bottleneck for modern computer systems to achieve high processor utilization. To hide memory latency, a variety of techniques have been developed. Dahlgren and Dubois proposed a prefetching scheme based on hardware [24], and Tcheun et al. presented a prefetching scheme based on software [25]. Various techniques have been proposed to consider both scheduling and memory latency hiding at the same time. Philbin et al. improved the inter-thread cache locality of sequential programs, which contain fine-grained threads, by using the idea of partitioning for scheduling threads [26]. Wang et al. proposed the first available technique that combines loop pipelining, prefetching, and partitioning to optimize the overall loop schedule [27,28,30]. However, their techniques are not able to take advantage of all the hardware resources available in CMPs. In this paper, we present the TLP algorithm that considers scheduling and memory latency hiding simultaneously. It generates two-part schedules that can completely hide memory latency and minimize write operation.

3. A motivational example

In this section, we provide a motivational example that shows the effectiveness of the TLP technique. In the example, we compare the schedules generated by a classical scheduling technique, i.e., the Partition Scheduling with Prefetching (PSP) scheduling

```
For i=0 to n
   For j=0 to m
     A[i, j]=C[i-1, j]+D[i-1, j-1];
     B[i, j]=A[i, j]*4;
     C[i, j]=B[i, j]+9;
     D[i. j]=B[i, j]+C[i, j-1];
   End for
End for
```

(a)



(b)

Fig. 1. (a) The example loop program. (b) The MDFG representation of the loop.

algorithm [4], and the TLP scheduling algorithm, respectively. For simplicity, we only illustrate the results of these two techniques without going into the details of how each result is generated. The MD loop shown in Fig. 1(a) is used in this section. There are two dimensions indexed by i and j in this loop. The multi-dimensional data flow graph (MDFG) representation of this two-dimensional loop is shown in Fig. 1(b). A node in an MDFG represents a task (computation), and an edge in an MDFG represents 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, there are four tasks, namely, A, B, C, and D. Hence, there are four nodes in the MDFG, each representing the corresponding computation. The edge from C to A represents the fact that the computation of $A[i,j]$ depends on the value of $C[i-1,j]$. A detailed and formal definition of an MDFG is presented in Section 4.

In this example, we assume that in our system, there are two processing cores in the processor, and there are 2 ALUs inside each core. Each core is equipped with an SPM, and each SPM has two data blocks. We also assume that it takes 1 clock cycle to finish each computation node, 1 clock cycle to fetch a datum in another core's SPM, 2 clock cycles to finish a main memory datum fetch, and 4 clock cycles to write a datum back to the main memory.

In order to hide memory latency, Sha et al. proposed the PSP algorithm, which takes into account the balance between computation and memory access time. Based on the dependency constraints in the MDFG shown in Fig. 1(b), the PSP scheduling algorithm generates a schedule shown in Fig. 2(a). For convenience, only one iteration of the loop is shown in representing a schedule. The notation $A(1)$ in the figure represents the computation of node A in iteration 1. f$A(k)$ represents fetching a datum from the main memory for node A in iteration k. w$C(k)$ represents writing the datum $C(k)$ to the main memory, and w$D(k)$ represents writing the datum $D(k)$ to the main memory. In this schedule, there are 4 fetch operations and 4 write operation in the memory part. There are 4 iterations which are scheduled at a time as a partition in processor part, and it takes 12 clock cycles to finish all 4 iterations because of the overlap between processor execution and memory access. As a result, the average time to complete an iteration is 3 clock cycles. However, we cannot take full advantage of all

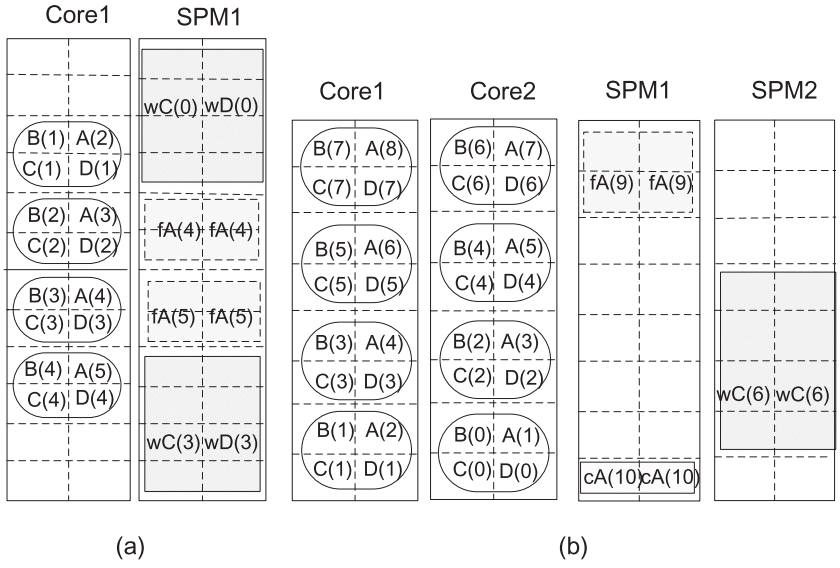


Fig. 2. Comparison between PSP scheduling and TLP scheduling. (a) PSP scheduling: the average schedule length is 3 clock cycle and the average write operation number is 1. (b) TLP scheduling: the average schedule length is 1 clock cycle and the average write operation number is 1/4.

the hardware resources available, because of the dependency constraints presented both inside each partition and between partitions. In PSP scheduling, only one core in the processor is used. Due to the memory size constraint and the significant amount of time of write/fetch operations in the main memory, we should explore a new technique to obtain a better schedule.

To minimize the write and fetch activities in the main memory and achieve full parallelism, we develop a schedule using our proposed TLP algorithm. The result is shown in Fig. 2(b). It takes 8 clock cycles to complete 8 iterations. In other words, the average time to complete 1 iteration is 1 clock cycle. The schedule length is equal to that of the IRP schedule. In the memory part, the 8 iterations only need two write operations. Hence, the average write operation to complete 1 iteration is $\frac{1}{4}$. Such noticeable performance improvement is obtained by considering the memory size while exploring iteration level parallelism, so that write operation is reduced, and high parallelism for cores in the processor can be utilized. As we can see, it is clear that TLP outperforms the PSP scheduling on multicore platforms.

4. Basic concepts and definitions

In this section, we introduce basic concepts which are used in this paper. First, we discuss the hardware model. Second, we introduce the models and notions that we use to analyze nested loops. Third, we introduce the loop partitioning technique and the iterational retiming technique. Fourth, we present the techniques of partitioning the partition space. Finally, we outline the TLP algorithm. In this paper, for clarity, our technique is presented using two dimensional notations.

4.1. Hardware model

In this paper, we target a multicore system with multiple ALUs and scratchpad memory (SPM) as its on-chip memory shown in Fig. 3 as our computing platform. In this architecture, each core has its own private on-chip SPM. All cores share a main memory with large capacity. Each core can access its own SPM and other cores' SPMs. Accessing data in other cores' SPM costs more time and energy than accessing data in its own SPM, but costs less than accessing data in an off-chip main memory. The goal of our technique is to fetch the operands of all computations into SPM before actual computations start. There are three types of memory operations in SPM, i.e., a read instruction which prefetches the computation data from the main memory, a write instruction which writes computation data back to the main memory, and a migration instruction which migrates computation data to another core's SPM. The three instructions are issued to ensure that computation data which will be employed soon will appear in the SPM.

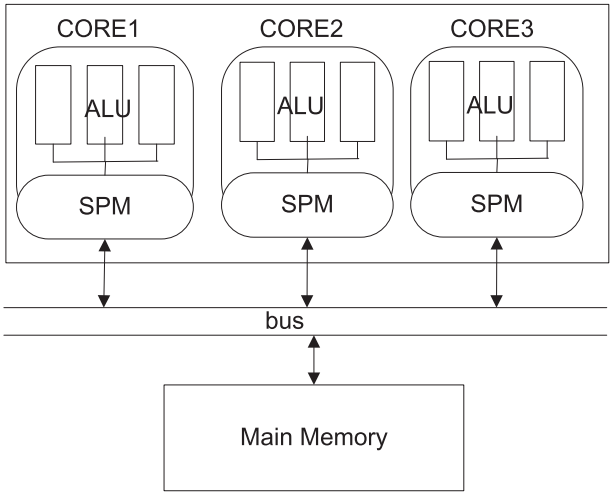


Fig. 3. A hardware model.

It is important to note that the lower level memories in the architecture model are SPM. There are two reasons for this. First, the local memory cannot be a pure cache, because we do not consider cache consistency and cache conflict in our paper. Second, SPM is a small on-chip memory component that is managed by software, and is more efficient in performance and power compared with the hardware-managed cache. Nowadays, many architectures are employing SPM rather than cache as their on-chip memories. SPM is shown to be both performance and power efficient as compared to the hardware-managed cache counterpart [4]. Our architecture is a general model. A real implementation is done in NVIDIA 8800 processor. Inside the processor, there are 12 processing cores. There are 12 ALUs and 64 K SPM in each processing core.

4.2. Modeling nested loops

The MDFG is used to model nested loops and is defined as follows. An MDFG $G = (V, E, d, t)$ is a node weighted and edge weighted directed graph, whose components are explained as follows. V is a set of computation nodes. $E \subseteq V * V$ is a set of dependence edges, where $(\mu, v) \in E$ means that μ must be scheduled before v. d is a function and $d(e)$ is the MD delays for each edge $e \in E$, which is also known as a dependence vector. We use $d(e) = (d_i, d_j)$ as a general formulation of any delay shown in a two-dimensional DFG (2DFG). For example, in Fig. 1(a), there is a two-dimensional loop program. Fig. 1(b) shows the corresponding MDFG, and these nodes represent the corresponding computations in the original loop program.

An *iteration* is one execution of all nodes in an MDFG. We regard an iteration as an iteration space node, which is represented by a vector (i, j). In Fig. 4(b), each node represents one iteration space node. The horizontal axis corresponds to the j index which is the inner-loop, and the vertical axis corresponds to the i index which is the outer-loop. Fig. 4(a) shows the detailed iteration space node graph. The solid vectors represent the inter-iteration data dependency. If a node in iteration (i, j) depends on a node in iteration (x, y), the dependence vector is $d(e) = (i - x, j - y)$. An edge with delay (0,0) represents data dependency within the same iteration. For example, the vector from node C in iteration (0,0) to node D in iteration (0,1) means that the computation of node D in iteration (0,1) depends on the computation of node C in iteration (0,0). The delay vector between nodes C and D is $(0, 1) - (0, 0) = (0, 1)$.

A schedule vectors s defines a sequence of an iteration space for a set of parallel equitemporal hyperplane. By default, a given nested loop is executed in a row-wise fashion, where the schedule vector is $s = (1, 0)$.

Rotation scheduling proposed by Chao and Sha [21] is a loop scheduling technique that optimizes loop scheduling under 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 the incoming edges of the node and adding one delay to each of its outgoing edges in the DFG. The details of rotation scheduling can be found in [21].

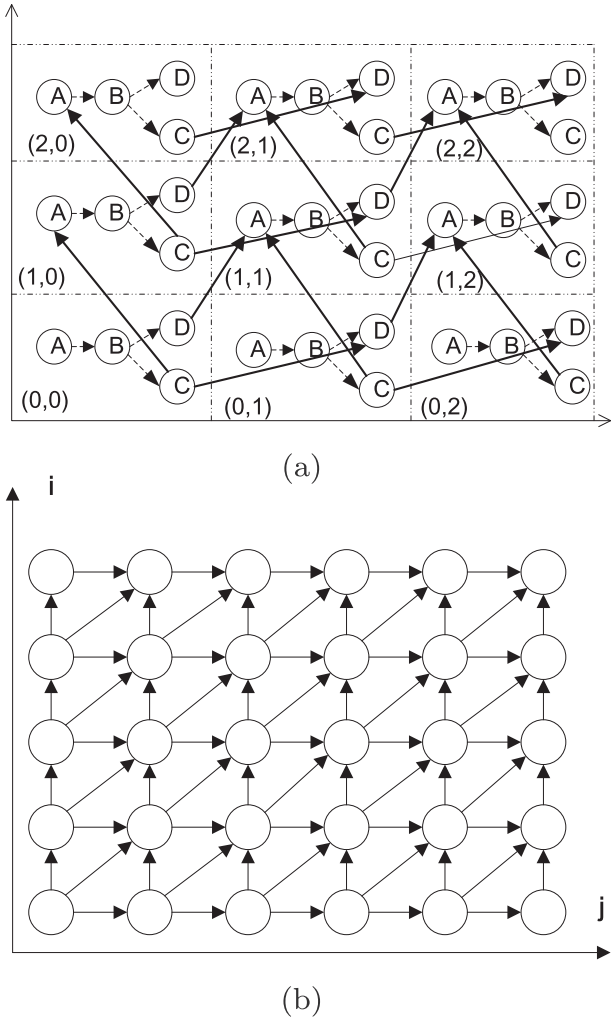


Fig. 4. An iteration space node graph.

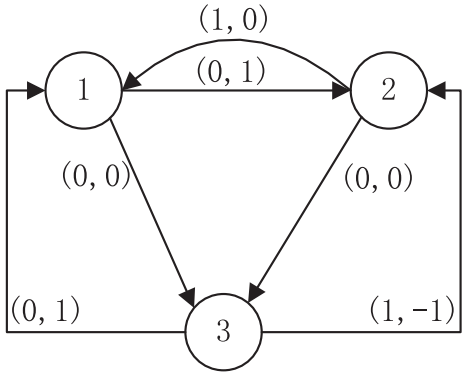


Fig. 5. An example of MDFG for partition.

4.3. Partition the iteration space and iterational retiming

Typically, execution of an iteration space is performed in the order of row-column or column-row. However, it takes lot of time and needs to store a lot of data in the main memory. In this paper, for overcoming these disadvantages, we can first partition an iteration space and then execute the partitions one by one. The two first level partition vectors are denoted by $P1_i$ and $P1_j$. We can get a *partition dependency graph* (PDG) according the first level partition. It is determined whether a partition is legal or not. A legal partition requires that there is no cycle in a PDG. As shown in Fig. 6(b), which is the PDG for the partition shown in Fig. 6(a), the partition is illegal because there are cycles in Fig. 6(b). Therefore, we cannot arbitrarily choose the partition vectors because of the dependencies in an MDFG. For example, let us consider the iteration space of the MDFG shown in Fig. 5, and the partitioning of the iteration space into rectangles, with $P1_j = (0,1)$ and $P1_i = (2,0)$, shown in Fig. 6(a), with the corresponding PDG shown in Fig. 6(b). As we see, it is illegal because of the forward dependency from partition (0,0) to partition (0,1) and the backward dependency from partition (0,1) to partition (0,0). Due to these two-way dependency between partitions, we cannot execute either one of them first and cannot execute the two partitions in different processors at the same time. Therefore, this partition is not implementable and is illegal. In contrast, let us partition the iteration space with partition vectors $P1_j = (0,1)$ and $P1_i = (2,-2)$ as shown in Fig. 6(c), and the corresponding PDG shown in Fig. 6(d). Since there is no two-way dependency, a feasible partition execution sequence exists. Thus, partition (0,0) can be executed first, then partition (0,1), and so on. Therefore, it is a legal partition.

Two extreme vectors, as shown in Fig. 7, i.e., the clockwise (CW) vector and the counterclockwise (CCW) vector are important to find the basic first level partition. They decide the directions of the legal first level partition vectors. They are defined as follows.

Definition 4.1. *A delay vectors set $D = \{d_1, d_2, \ldots, d_k\}$ contains all delay vectors in an MDFG. The extreme clockwise vector CW must satisfy the following two conditions: (1) $CW \in D$; (2) all the vectors in the set of $D - \{CW\}$ are in the counterclockwise region of CW.*

The definition of the CCW vector is similar.

A legal basic first level partition must satisfy the following condition, namely, the first level partition vectors $P1_i$ and $P1_j$ cannot lie between CW and CCW. In other words, they can only be outside of CW and CCW or be aligned with CW and CCW.

After a basic partition is identified via $P1_i$ and $P1_j$, an *iteration space graph* (IFG) 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 nonzero 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 [13].

4.4. Partition the first partition space

We regard a first level partition as a partition space, which is defined by a vector $(partition_\mu, partition_\nu)$. In Fig. 8(b), for simplicity, each node in the graph represents one partition space node. Instead of executing the entire partition space in order of rows and columns, we again partition the first level partition space and execute the second level partitions one by one. If we treat one second level partition as a cluster, then all first level partitions in the cluster can be executed at the same time. Similar to the first level partition, the second level partition vectors, denoted by $P2_i$ and $P2_j$, also cannot be arbitrary, due to the dependencies between

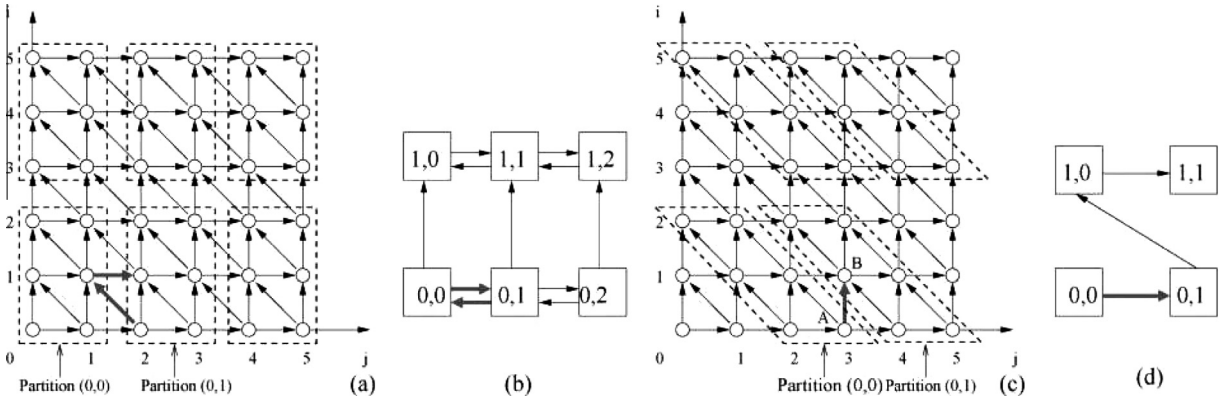


Fig. 6. (a) An illegal partition of the iteration space. (b) The partition dependency graph of (a). (c) A legal partition of the iteration space. (d) The partition dependency graph of (c).

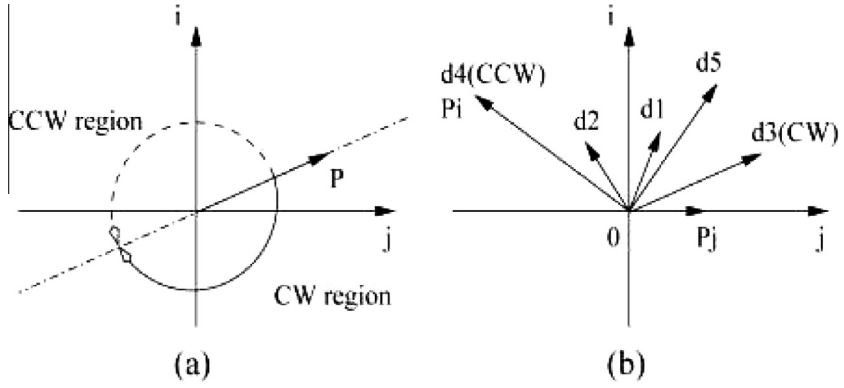


Fig. 7. (a) The CW and CCW regions relative to vector P. (b) The extreme CW and CCW vectors of vectors $d_1, d_2, \ldots, d_k$ and the partition vectors P_i and P_j.

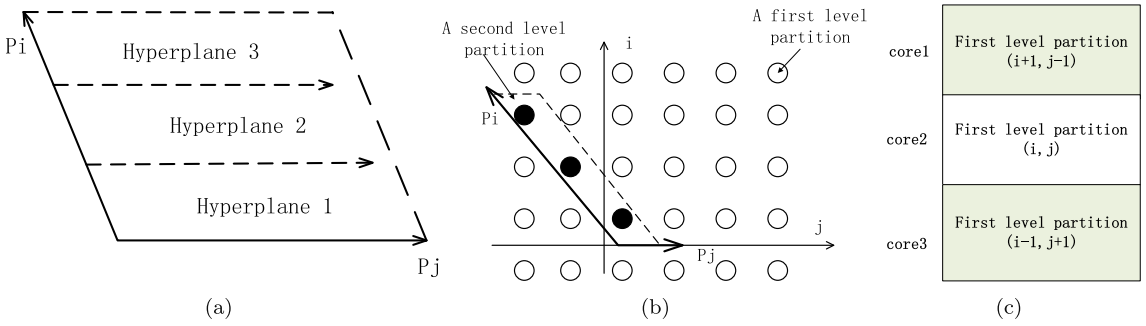


Fig. 8. (a) The second level partition will be executed from left to right in the $P2_j$ direction and proceed the next hyperplane along the direction perpendicular to $P2_i$. (b) The first level partition space. (c) A second level partition.

partition spaces. In this paper, we choose the second level partition vectors as $P2_i = N \times (1, -1)$ and $P2_j = (0, 1)$. N is related to the number of cores. In the next section, we will describe in detail how to determine the value of N.

Fig. 8 shows the second level partitions and first level partitions scheduled in order. We execute the second level partition from left to right in the $P2_j$ direction, as illustrated in Fig. 8(a), and the next hyperplane along the direction of vector perpendicular to $P2_j$. We execute the first level partitions following this way, i.e., all iterations in the same first level partition must be scheduled in the same core; and all first level partitions in the same second level partition must be scheduled at the same time, as illustrated in Fig. 8(c). Then, we proceed the next second level partition. Fig. 8(b) shows an example of the first level partition order in a second level partition. The black dots represent the first level partition in the second level partition. After second level partition, the order of first level partitions scheduling must satisfy the following condition, i.e., if there are two first level partitions, partition (u, v) and partition (k, v), in the same hyperplane, and $u < k$, the partition (u, v) must be executed no earlier than partition (k, v).

4.5. TLP algorithm framework

TLP generates a schedule consisting of two parts, i.e., the processor part and the memory part. The original MDFG usually contains inter-iteration/partition dependencies and intra-iteration/partition dependencies. This causes that different first level partitions need different write/prefetch activities. In our algorithm,

the first level partitions executed on different cores have different write/prefetch activities, and the first level partitions executed on the same core have identical write/prefetch activities. The reason for this phenomenon is the first level partition size f_j.

We call the first level partition which is currently being executed in a core the *current first level partition*, the first level partition that will be executed next in the same core the *next first level partition*, and the first level partition in the *i*-axis aligned with the current first level partition the *top first level partition*. Any other first level partition which has not been executed is called *other first level partition* (see Fig. 9(a)). For the current first level partition, the number of computation data which will be stored for the next first level partition is denoted by NUM_{next}. The number of computation data which will be stored for the top first level partition is denoted by NUM_{top}. The number of computation data which will be stored for the other first level partition is denoted by NUM_{other}.

Fig. 9(b) gives an example of our overall schedule. There are 12 iterations in our second level partition. In the processor part, the TLP scheduling algorithm generates a schedule for one iteration with four control-steps. Four iterations can be executed at the same time by four different cores. This schedule is then duplicated 3 times for all iterations inside the second level partition. In a second level partition, all iterations, which are executed in the same core, belong to the same first level partition. There are four cores in the processor, which means that four first level partitions can be executed in different cores at the same time. In the memory part, all the write operation for the previous and current second level partitions are scheduled, and all the prefetch operations for the next second level partition are scheduled. A rectangle represents a write operation or prefetch operation. A white rectangle represents that write operation or prefetch operation does not exist; in other words, the memory does not execute any operation in the rectangle time. As we can see, in the overall schedule, 12 iterations are finished in 12 control steps, that is, on average, there is ave_len(overall) = 12/12 = 1 control step per iteration. There are 7 write activities and 7 prefetch operations, that is, on average, ave_write(overall) = 7/12 ≐ 0.58 write operation and ave_prefetch(overall) = 7/12 ≐ 0.58 prefetch operations per iteration.

5. TLP scheduling

In this section, we will first explain our TLP scheduling algorithm in detail. Then, we will show some theorems to determine the partition size, so that complete memory latency hiding can always be achieved, while reducing write operation.

5.1. Algorithm

A TLP schedule consists of two parts, i.e., the processor part and the memory part. The processor part of a schedule for one iteration is generated by using the rotation scheduling algorithm. Rotation scheduling is described in detail in [21], for scheduling cyclic DFGs using loop pipelining. The rotation technique repeatedly transforms a schedule into a more compact schedule. MD rotation, which is described in [20], implicitly changes the schedule vector and the execution sequence of the iterations to obtain a new compact scheduling. Since we wish to maintain row-wise execution in TLP scheduling, we schedule one iteration by using the one-dimensional rotation technique instead of the MD rotation technique.

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

A first level partition is identified by two first level partition vectors, i.e., $p1_i$ and $p1_j$, where $p1_i = f_i \times p1_{i0}$ and $p1_j = f_j \times p1_{j0}$. While $p1_{i0}$ and $p1_{j0}$ determine the direction and shape of a first level partition, f_i and f_j determine the size of a first level partition. How to choose the vectors $p1_{i0}$ and $p1_{j0}$ to identify the shape of a legal basic first level partition is discussed in detail in Section 4. How to choose f_i is shown in algorithm TLP. We will pay more attention on how f_j is chosen to achieve the goal of complete memory latency hiding, which means that the schedule length of the memory part will always be less than or equal to the scheduling length of the processor part.

After obtaining the direction and size of a basic first level partition, we apply Step 2 of the iteration retiming algorithm [16] to transform the basic first level partition into a retimed first level partition, so that all the iterations inside a retimed first level partition can be executed in parallel. After a retimed first level partition is identified, we can partition the first level partition space to obtain a second level partition.

Algorithm 1. Two-Level Partition (TLP) algorithm.

Input: An MDFG $G = (V, E, d, t)$, the capacity of each SPM, and the number of cores.
Output: A two level partition schedule that reduces the number of write back operations and increases parallelism.

$s \leftarrow$ rotation scheduling (G)
$L_s \leftarrow$ the schedule length of s
/* find first level partition size and shape */
$p1_{j0} \leftarrow (0, 1)$
$p1_{i0} \leftarrow$ CCW vector of all delays
obtain f_{i1} based on Theorem 5.1 where f_{j1}=1
obtain f_{i2} based on Theorem 5.2 where f_{j1}=1
if $f_{i1} > f_{i2}$ **then**
  $f_{i0} \leftarrow f_{i2}$
  obtain f_{j0} based on Theorems 5.1 and 5.2
  if $f_{j0} > 1$ **then**
    $f_j \leftarrow f_{j0}, f_i \leftarrow f_{i0}$
  end if
  if $f_{j0} = 1$ **then**
    $f_j \leftarrow 1, f_i \leftarrow f_{i1}$
  end if
end if
if $f_{i1} \leqslant f_{i2}$ **then**
  $f_j \leftarrow 1, f_i \leftarrow f_{i1}$
end if
$P1_i \leftarrow f_i \times p1_{i0}$
$P1_j \leftarrow f_j \times p1_{j0}$
obtain basic first level partition with $P1_i$ and $P1_j$
call the iteration retiming algorithm to transform the basic first level partition into a retimed first level partition
/* find second level partition size and shape*/
$N \leftarrow$ number the cores
$p2_i \leftarrow (N, -N)$
$p2_i \leftarrow (0, 1)$
do processor part scheduling
do memory part scheduling

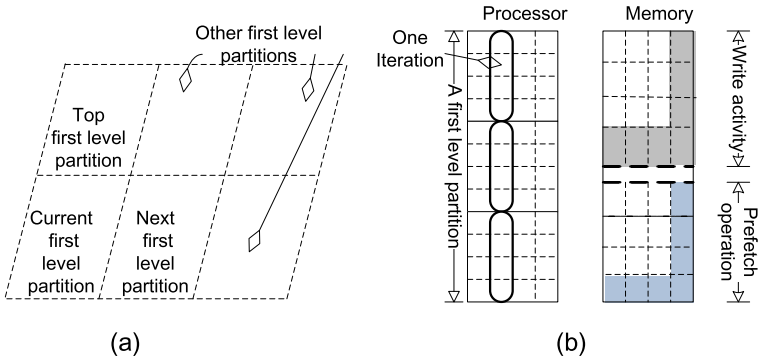


Fig. 9. (a) The different first level partitions. (b) The memory part and processor part of a TLP schedule.

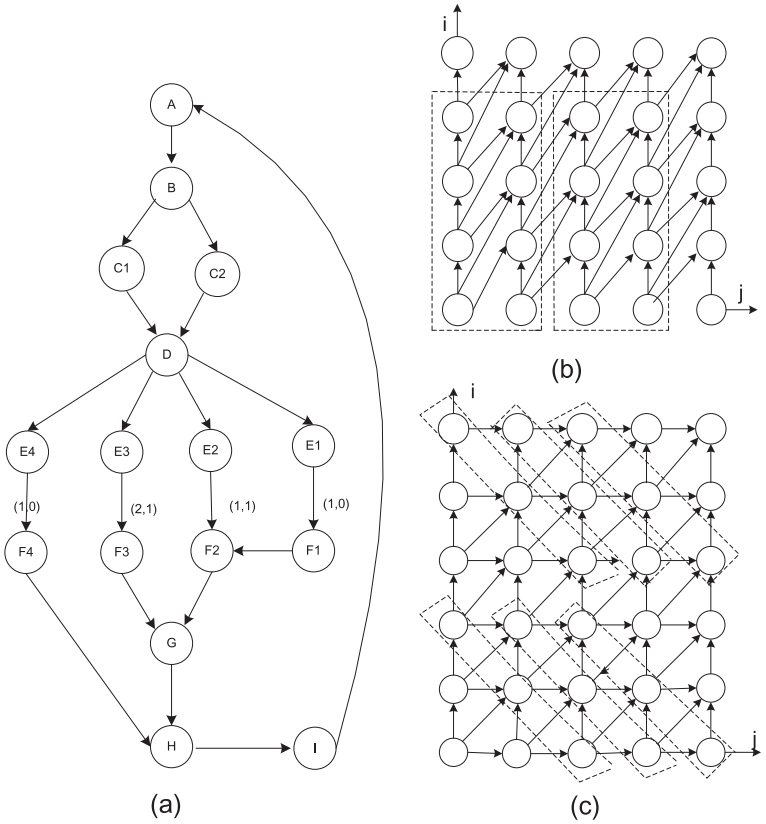


Fig. 10. (a) The MDFG of a Floyd filter. (b) The iteration space after first level partition. (c) The first level partition space after second level partition.

A second level partition is identified by two second level partition vectors, $p2_i$ and $p2_j$, which determine the direction, size, and shape of a second level partition. In this paper, $p2_i = N \times (1, -1)$ and $p2_j = (0, 1)$. The second level partition vectors are legal, because the first level partition (i, j) is independent of any another first level partition $(i, j + k), k \geqslant 1$. For the scheduling order of the first level partitions, we put the number of cores as the value of N. For example, assuming that the input MDFG is to be run in the hardware model shown in Fig. 3, which has 3 cores, we have $N = 3$. In other words, the second level partition vectors is $p2_i = (3, -3)$ and $p2_j = (0, 1)$. After a second level partition is identified, we can start to construct both the processor and memory parts of a schedule. We construct these two parts of a schedule by using write-back-data-pipelining (the scheme was described in detail by Xue et al. [16]) and the Automatic Data Movement scheme, which was explained by Baskaran et al. [29].

5.2. Schedule generation scheme

In this subsection, we will use an example to illustrate our schedule generation scheme. We consider the Floyd filter executed on a three-core processor. Fig. 10(a) shows the MDFG representation of the Floyd filter. The shape and size of the iteration space of the Floyd filter after the first level partition is shown in Fig. 10(b). Fig. 10(c) shows the second level partition. As seen from the figure, we can obtain the first level partition vector (P_{i1}, P_{j1}) as (4,1). There is no dependency between the first level partitions within each partition.

With all the pieces now in place, we perform the TLP scheduling algorithm by considering both levels separately. The first level partition schedule of the Floyd filter, as shown in Fig. 11, consists of a memory part and a processor part. For each first level partition, the processor part of the schedule is generated in the default order. The

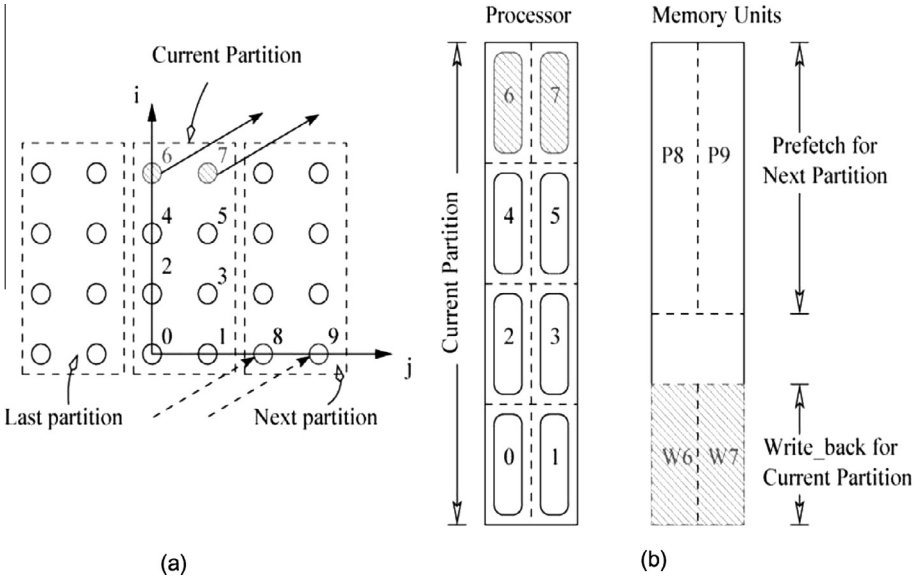


Fig. 11. (a) Iteration space. (b) First level partition schedule scheme.

memory part of the schedule is made up of write operation, prefetch operation, and migration operation. Here, we only consider the prefect operation and write operation. The memory executes the write operation for the current first level partition and the prefetch operations for the next first level partition. In the example, we first perform the prefetch operations to prefetch the data needed for iterations 8 and 9 in the next partition. Then, we schedule the write operations to write back the data generated by iterations 6 and 7. However, the migration operation should be considered in the multi-core processor. The memory part of different first level partitions scheduled on second level partitions are different. The difference depend on how many migration operations change into memory part.

In the second level partition schedule, different first level partitions are scheduled on different processing cores. The TLP algorithm will take into account the latency for memory units to finish the migration operations. As most of the data transferred via migration operation will be used in the top first level partition. Fig. 12 shows the second level partition schedule of the Floyd filter shown in Fig. 10. Each second level partition is scheduled by three multi-core processors, i.e., Core1, Core2, and Core3. Each first level partition is executed the same core. In the memory part, different SPM perform different memory schedules. Core1 will schedule the prefetch operations and write operations. Core2 and Core3 will schedule the prefetch operations, write operations, and migration operations. Once we ensure that memory operations will be completed before the processor schedule is completed for the currently partition, we can ensure that the updated data is in the right place before the execution of the next second level partition. As a result, data coherence among multiple cores processors is guaranteed under TLP scheduling.

5.3. Partition size

To determine the partition size, we will first define the number of data stored for *top*, *next*, and *other first level partitions* given a first level partition size of f_i and f_j. The number of data stored for *top*, *next*, and *other first level partitions* can be estimated by calculating these shared areas, as shown Fig. 13, with respect to every inter-iteration delay vector $d_k \in D$. Consider a delay vector $d_k = (d_{ki}, d_{kj})$ in Fig. 13, all duplicate vectors originating in the region PQVU will enter *top first level partition*, which is where data are needed to store for executing the *top first level partition*. All duplicate vectors originating in the region UVRS will enter *other first level partition*, which is where write and prefetch operations are needed. All duplicate vectors originating in the region VSWX will enter *next first level partition*, which is where data are needed to store for the *next first level partition* can be executed. We denote the area of PQVU as A_{goto_top}, UVRS as A_{goto_others}, and VSWX as A_{goto_next}.

Lemma 5.1. *Given a delay vector* (d_{ki}, d_{kj}), *we have* $A_{goto\_top}(d_k) = d_{ki}(f_j - d_{kj})$, $A_{goto\_next}(d_k) = d_{kj}(f_i - d_{ki})$, *and* $A_{goto\_others}(d_k) = d_{ki}d_{kj}$.

Proof. As shown by the shared areas in Fig. 13, we have

$$A_{goto\_top}(d_k) = area(PQVU) = d_{ki}(f_j - d_{kj}) \quad (1)$$

$$A_{goto\_next}(d_k) = area(VSWX) = d_{kj}(f_i - d_{ki}) \quad (2)$$

$$A_{goto\_others}(d_k) = area(UVRS) = d_{ki}d_{kj} \quad (3)$$

The lemma is proven.

Summing up all of these areas for every distinct delay vector $d_k \in D$, we obtain the values of NUM_{next}, NUM_{top}, and NUM_{other} as follows:

$$NUM_{other} = \sum_{d_k} A_{goto\_others}(d_k) = \sum_{d_k} (d_{ki})(d_{kj}) \quad \forall d_k \quad (4)$$

$$NUM_{top} = \sum_{d_k} A_{goto\_top}(d_k) = \sum_{d_k} d_{ki}(f_j - d_{kj}) \quad \forall d_k \quad (5)$$

$$NUM_{next} = \sum_{d_k} A_{goto\_next}(d_k) = \sum_{d_k} d_{kj}(f_i - d_{ki}) \quad \forall d_k \quad (6)$$

From these definitions of the numbers of write/prefetch activities, we know that they all are proportional to the size of f_j and do not change with f_i. The number of data, which are needed to be stored in SPM, is proportional to the size of f_i and does not change with f_j. However, the size of SPM is a constraint, and therefore the data kept in the SPM is limited. In order to hide the memory latency while reducing write operation, we will try to have the optimal number

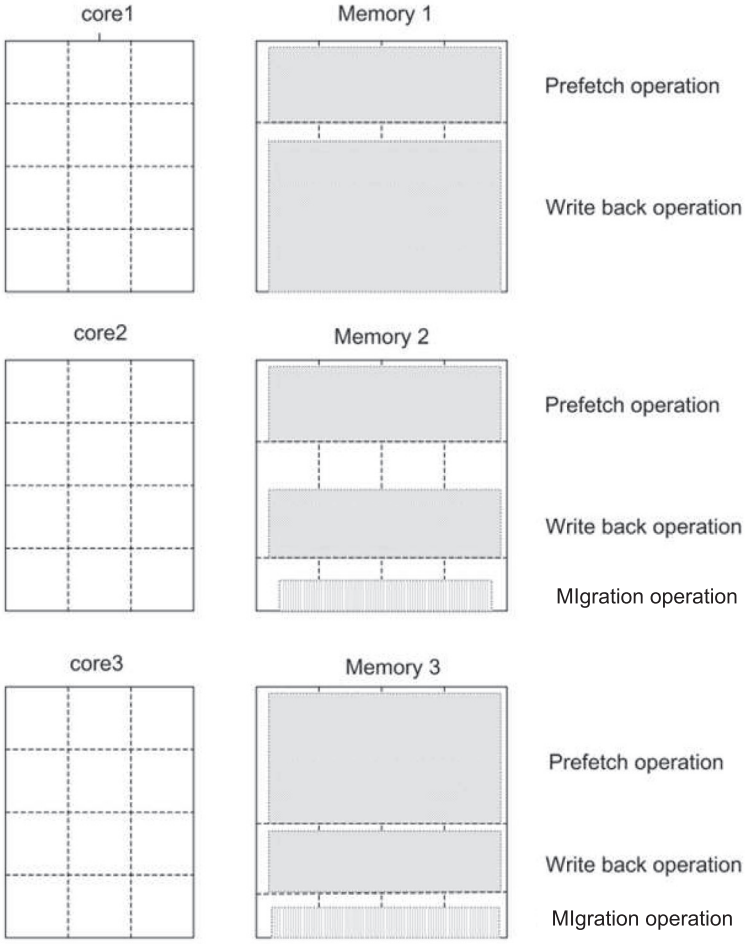


Fig. 12. TLP scheduling scheme.

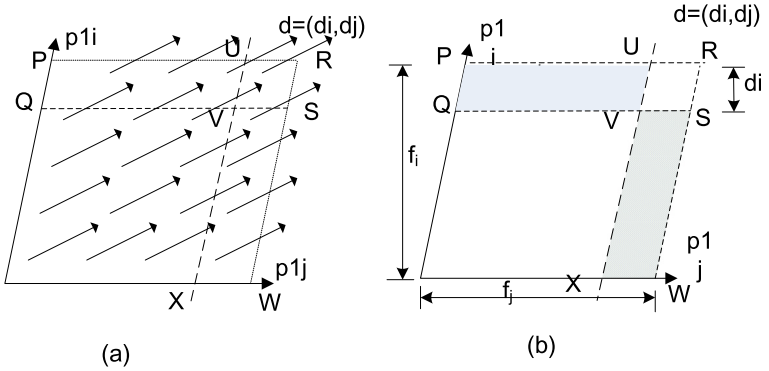


Fig. 13. Calculating the number of delay edges crossing the boundary of the current partition.

of data stored into SPM for a first level partition executed. So we will keep f_i fixed and find the right f_j to achieve an optimal schedule, where the write operation is the least and memory latencies are hidden completely. In this paper, the number of write operations for a first level partition is NUM_{other} and equals to the area of UVRS in Fig. 13. The number of prefetch operations is equal to that of write operations, because we only write back those data that we will ever need to fetch from main memory. The number of migration operations for a first level partition is NUM_{top} and equals to the area of PQVU in Fig. 13. However, the first core (core 1 in Fig. 12) does not have migration operations, because all data generated by the first core must be written into main memory to be used in the other second level partition.

Theorem 5.1. *Assume that the size of SPM is M_s, which has been known. The following inequality is satisfied:*

$$2NUM_{other} + NUM_{top} + NUM_{next} \leqslant M_s.$$

Proof. The TLP algorithm requires that the SPM should be large enough to hold all the data that are needed during the execution of the first level partitions. The SPM requirement consists of two

categories, i.e., storing all data generated by the *current first level partition* in the SPM, and prefetching all data required to execute the *next first level partition* into the SPM. The former includes those data stored for the *next first level partition*, stored for the *top first level partition*, and stored for the *other first level partitions*. Therefore, the SPM needed for the former is $NUM_{other} + NUM_{top} + NUM_{next}$. The latter includes all data generate by previous first partitions that have been executed and needed to execute the *next first level partition*. Since the data which are generate by the *current first level partition* and needed to execute the *next first level partition* are already kept in SPM, prefetching of these data is not required. Thus, the SPM needed for the later is NUM_{other}. Finally, the SPM needed to execute this first level partition is $2NUM_{other} + NUM_{top} + NUM_{next}$. Since the size of SPM is M_s and the SPM should be large enough to hold all the data that are needed during the execution of the first level partitions, the equality in Theorem 5.1 is satisfied. The theorem is proven.

From Theorem 5.1, we can obtain the set of first level partition sizes (f_i, f_j) which meets the requirement that the SPM should be large enough to hold all the data that are needed during the execution of the first level partitions. However, the TLP scheduling algorithm requires that memory latencies are hidden completely while reducing write operation. Therefore, the product of f_i and f_j must be less than certain numeric value, because the schedule length of the memory part must be not longer than that of the processor part at any time. The following theorem shows how we can use the above lemma and theorem to find the partition factors of f_i and f_j.

Theorem 5.2. *To achieve full parallelism, the following inequality must be satisfied:*

$$(NUM_{top} + NUM_{other})Tw + NUM_{other} \cdot T_{pre} \leqslant Ls \cdot f_i \cdot f_j,$$

where Tw represents the time required for a write operation, T_{pre} represents the time required for a prefetch operation, and Ls represents the schedule length of one iteration.

Proof. In our algorithm, all first level partitions, which are located in the same second level partition, are executed in parallel. This causes different first level partitions having different prefetch/write operation. Therefore, the schedule lengths of different cores' memory part show different values. For memory latencies to be hidden completely, the maximum of all schedule lengths of the memory part must be no longer than any schedule length of the processor part. In the memory part of a TLP schedule, the maximum length is that of *top first level partition* in a second level partition, because it must write the number (NUM_{top}+NUM_{other}) of data to the main memory and prefetch the number NUM_{other} of data to SPM. In the memory part of a schedule, the maximum schedule length consists of a prefetch part and a write-back part. The length of a prefetch part is $NUM_{other} \cdot T_{pre}$ and the length of a write-back part is $(NUM_{top} + NUM_{other})Tw$. Therefore, the left-hand side of the above inequality is the maximum schedule length of a memory part. In the right-hand side of the above inequality, *Ls* represents the schedule length of one iteration, and $f_i \cdot f_j$ represents the number of iterations to be executed by each core. Hence the right-hand side represents the length of the processor part of a schedule. To achieve full parallelism, it is required that the maximum schedule length of the memory part must be less than that of the processor part. This way, we can guarantee that the memory part of the schedule is not longer than the processor part at any time. Therefore, the inequality in Theorem 5.2 must be satisfied. The theorem is thus proved.

We have finished our discussion on the size of the first level partition. However, it is not enough for designing a real system. When designing a real system, a designer must use the SPM whose size is fit to implement the proposed TLP schedule. Therefore, we propose Theorem 5.3 and 5.4 to further restrict the partition factors f_i and f_j.

For ease of presentation, let us turn the inequality in Theorem 5.1 to an equality and call it Eq. 5.1. Similarly, we turn the inequality in Theorem 5.2 to an equality and call it Eq. 5.2.

Theorem 5.3. *We obtain f_{i1} based on Theorem 5.1 and f_{i2} based on Theorem 5.2, with $f_{j1} = f_{j2} = 1$. If $f_{i1} > f_{i2}$, there exists a pair (f_i, f_j) which satisfies both Theorem 5.1 and Theorem 5.2. In other words, we can minimize the write operation while achieving full parallelism.*

Proof. The basic idea of the proof is to show that Eq. 5.1 and Eq. 5.2 have at least two common points, and therefore, there is a nonempty common region of points which satisfy the inequalities of both Theorem 5.1 and Theorem 5.2. From the above lemmas and theorems, we know that f_i linearly depends on f_j. Furthermore, the inequality of Theorem 5.1 implies that the greater f_j is, the smaller f_i is. The inequality of Theorem 5.2 implies that f_i is inversely proportional to f_j. Since the left-hand side of the inequality in Theorem 5.1 is less than the right-hand side, a pair (f_{i1}, f_{j1}) that satisfies the inequality of Theorem 5.1 must be under the curve of Eq. 5.1. Furthermore, there must be a pair $(0, k)$ which satisfies Theorem 5.1, where k is a constant value. A pair (f_{j2}, f_{i2}) which satisfies the inequality of Theorem 5.2 must be located in the first quartile of the plane of f_j and f_i, and are located above the curve of Eq. 5.2. Therefore, $f_{j1} = f_{j2} = 1$ and $f_{i1} > f_{i2}$ imply that there exists a shared pair (f_i, f_j) satisfying both Theorem 5.1 and Theorem 5.2. This is because there exists a shared point (u_1, v_1) located on the curve of Eq. 5.1 as well as located on the curve of Eq. 5.2. Since f_j approaches 0 when f_i is infinitely large in Eq. 5.2, there must exist anothershared point (u_2, v_2) satisfying Eq. 5.1 and Eq. 5.2. Therefore, there must exist a pair (f_i, f_j) which satisfies Theorem 5.1 and Theorem 5.2. In other words, if $f_{i1} > f_{i2}$ and $f_{j1} = f_{j2} = 1$, we must be able to find a pair (f_i, f_j) that can minimize write operation while achieving full parallelism. This proves the theorem.

From Theorem 5.3, we can easily reach Theorem 5.4.

Theorem 5.4. *If $f_{i1} < f_{i2}$, we cannot find a pair which satisfies Theorem 5.1 as well Theorem 5.2. In other words, we cannot obtain a schedule that can minimize write operation while achieving full parallelism.*

By presenting the above theorems, we have finished our discussion on the first level partition size. In general, we use Theorem 5.1 and Theorem 5.2 to obtain the partition size. However, if there are large SPM accessing number, we obtain the partition size only by Theorem 5.1. Theorem 5.4 illustrates that when the situation of large SPM accessing number occurs, we cannot obtain a schedule that can completely hide memory latency. Therefore, the schedule length is decided by the schedule length of memory part, and the performance should be decreased when there are large SPM accessing number. In a TLP schedule, the second level partition vectors are fixed, and the second level partition size is only related to the number of cores.

6. Experiments

In this section, we present our experimental results. The effectiveness of the TLP algorithm is evaluated by running the DSPStone benchmarks which are found in [13]. The following DSP benchmarks with two-dimensional loops are used in our experimental:

Table 1
Benchmarks information.

Benchmark	Nodes	Edges	Benchmark	Nodes	Edges
IIR	16	23	WDF(1)	12	16
FLOYD(1)	16	20	WDF(2)	48	72
FLOYD(2)	64	80	DPCM(1)	16	23
FLOYD(3)	144	180	DPCM(2)	64	92
2D	34	49	DPCM(3)	144	207

WDF (Wave Digital Filter), IIR (Infinite Impulse Response filter), 2D (Two Dimensional filter), Floyd (Floyd–Steinberg algorithm), and DPCM (Differential Pulse-Code Modulation device). Table 1 shows the number of nodes and the number of edges of each benchmark. The DFGs are all extracted from the gcc compiler and then fed into a custom simulator framework which is similar to CELL Processor.

In the proposed simulator framework, there are 3 cores and each core has 3 ALUs and a SPM with capacity of 32 KB, and the main memory size is 32 MB. In the proposed simulator framework, we use a circular data bus similar to the Cell processor's Element Interconnect Bus (EIB). Each processor takes 1 time unit for a computation operation. Reading a datum from other cores' SPM takes 1 time unit. Reading a datum from the main memory takes 2 time units. Writing a datum to the main memory takes 4 time units. All the experiments are conducted by a simulator framework, which is similar to Cell processor, on an Intel core 2 Duo Processor E7500 2.93 GHz processor and 1 GB memory running ubuntu-12.10.

We performed experiments on four algorithms to show the effectiveness of TLP. They are list scheduling, rotation scheduling [6], PSP scheduling [4], IRP scheduling [13]. List scheduling is the most traditional algorithm. It is a greedy algorithm that seeks to schedule a MDFG node as early as possible while satisfying the data dependence and resource constraints. Rotation scheduling attempts to get a more compact scheduling with resource constraints. In our experiment, we use list scheduling to schedule the ALU operations, but the memory is not partitioned. PSP scheduling attempts to balance the computation and communication for MDFG. IRP algorithm attempts to completely hide memory latencies for applications with MD loops on architectures like CELL processor. However, the PSP and IRP scheduling algorithms do not account the capacity of local memory. In this paper, the memory operations include write operations, prefetch operations, and migration operations. The cost of migration operations is far less than that of prefetch and write operations. And the number of prefetch operations is equal to that of write operations. Therefore, although both prefetch and write operations are important for schedule length, we only choose the write operations as the objective.

Figs. 14(a) and 15(a) show the results of the TLP scheduling algorithm compared with the List, Rotation, PSP, and IRP scheduling algorithms in the schedule length of each iteration for DPCM and Floyd. From the two figures, we see that TLP and IRP are better than the other three algorithms in schedule length. If the number of nodes in each iteration is small and a SPM has enough capacity, a TLP schedule has the same schedule length as an IRP schedule. However, if the number of nodes in each iteration is large and a SPM does not have enough capacity, a TLP schedule is superior to an IRP schedule. Figs. 14(b) and 15(b) show the results of the TLP scheduling algorithm compared with the List, Rotation, PSP, and IRP algorithms in the number of write operation of each iteration. From the two figures, we see that TLP is better than all other four algorithms.

The TLP scheduling algorithm is further compared with the List scheduling algorithm (the weakest of the known algorithms), and the IRP scheduling algorithm (the strongest of the known algorithms), which is widely applied to MD loop partition schedule to achieve full parallelism with consideration of write operation. Table 2 and Fig. 16 show the comparison results. The schedule length per iteration comparison results are shown in Fig. 16(a). In our experiment, we assume that

$$LB = \left\lceil \frac{Ls}{N_{core}} \times T_{alu} \right\rceil \tag{7}$$

indicates the average schedule length per iteration of the TLP algorithm, and *len* represents the average schedule length per iteration of the List scheduling or the IRP scheduling algorithm. As we can see, it is clear that the TLP scheduling algorithm is superior to the List scheduling and IRP scheduling algorithms. We compare TLP scheduling with List scheduling and IRP scheduling via computing the

$$ratio = \frac{LB}{len} \tag{8}$$

respectively. We can observe that the average ratios of the length of a TLP schedule to those of a List schedule and an IRP schedule are 17.75% and 76.30%, respectively.

Fig. 16(b) shows the write operation comparison results between the three scheduling algorithms. We assume that N_{write}

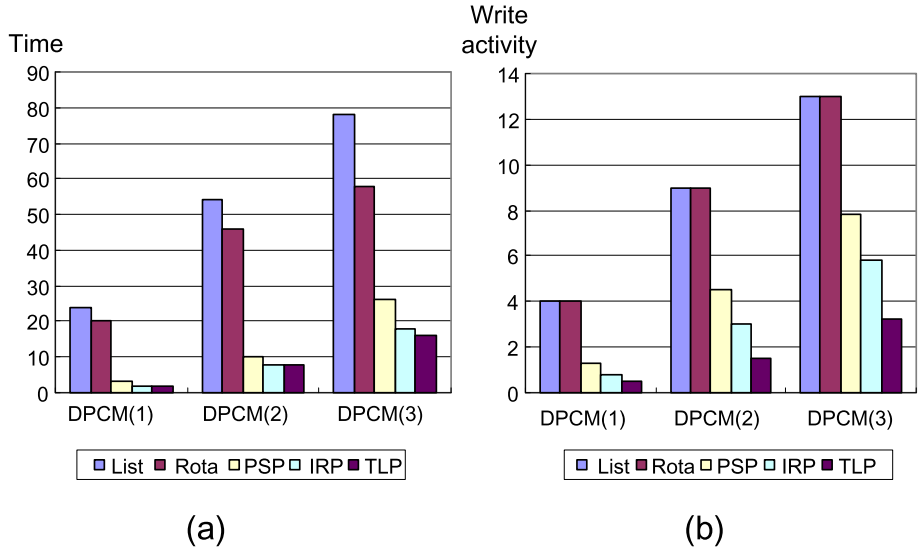


Fig. 14. (a) The time. (b) The write operation.

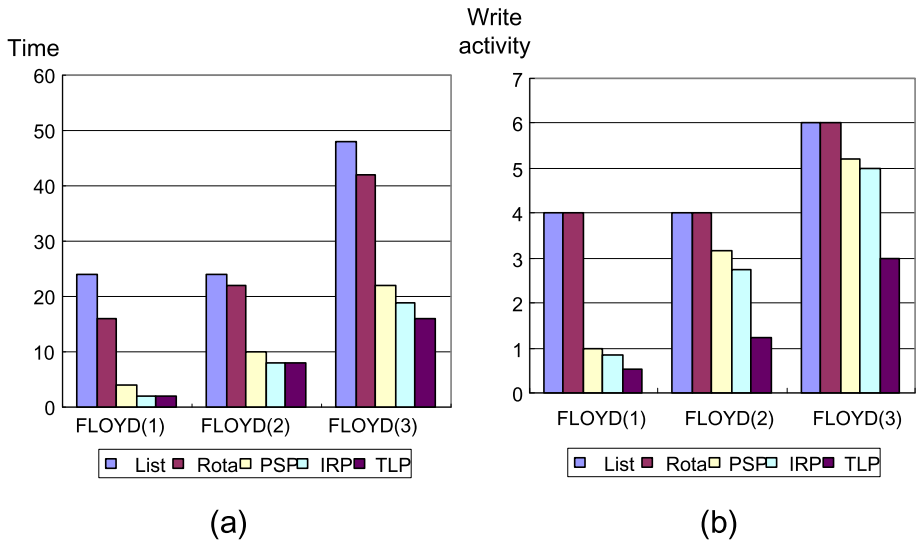


Fig. 15. (a) The time. (b) The write operation.

Table 2
Experimental results of DSP filter benchmarks ($T_{alu} = 1, T_{pre} = 2, T_w = 4$, and $N_{alu} = 3$).

Benchmark	TLP			List				IRP			
	Ls	LB	N_{avg}	Len	Ratio (%)	N_w	Ratio (%)	Len	Ratio (%)	N_w	Ratio (%)
IIR	6	2	0.52	36	5.6	6	9.3	2.00	100.0	0.82	63.4
2D	12	6	0.92	24	25.0	4	23.0	8.42	71.3	1.57	58.6
WDF(1)	4	2	0.27	24	8.3	4	6.8	2.00	100.0	0.50	54.0
WDF(2)	16	6	0.89	30	20.0	5	17.8	11.40	52.6	2.00	44.5
DPCM(1)	6	2	0.52	24	8.3	4	13.0	2.00	100.0	0.82	63.4
DPCM(2)	22	8	1.50	24	33.3	4	30.5	15.00	53.3	3.00	50.0
DPCM(3)	48	16	3.22	48	33.3	6	50.0	26.80	59.7	5.80	55.5
FLOYD(1)	6	2	0.52	24	8.3	4	13.0	2.50	80.0	0.83	62.7
FLOYD(2)	22	8	1.22	54	14.8	9	16.7	13.00	61.5	2.75	44.4
FLOYD(3)	46	16	2.50	78	20.5	13	14.8	19.00	84.2	5.00	50.0
Average ratio				–	17.75	–	19.50	–	76.30	–	54.65

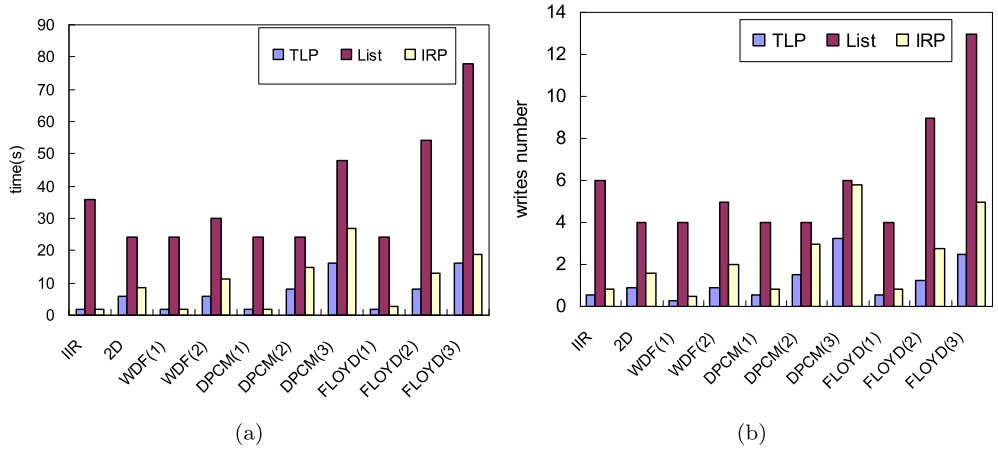


Fig. 16. Experimental results. (a) Schedule length comparison between the three schedules. (b) Write operation comparison between the three schedules.

indicates the number of write operation in a second level partition, and

$$N_{avg} = \left\lceil \frac{N_{write}}{N_{iter} N_{core}} \right\rceil \tag{9}$$

is the average number of write operation per iteration, and N_w is the number of write operation per iteration of the List scheduling or the IRP scheduling algorithm. The following ratio compares the write

Table 3
NVIDIA GeForce 8800 GPU GTS system specification.

Component	Description
GPU core	Number of ALU: 12; Frequency: 1.5 GHz
SRAM SPM	Size: 64 K; Migration latency: 100 ns; Migration energy: 1.72 nJ
Main memory	Size: 782 M; Access latency: 300 ns; Access energy: 9.41 nJ

Table 4
Experimental results of DSP filter benchmarks on NVIDIA GeForce 8800 GTS GPU.

Bench.	TLP				List				IRP			
	$f_i \times f_j$	Ls	LB	N_{avg}	Len	Ratio (%)	N_w	Ratio (%)	Len	Ratio (%)	N_w	Ratio (%)
IIR	14×2	40	5	0.13	305	1.64	6	2.17	6.02	83.05	0.28	46.43
2D	6×2	60	8	0.34	324	2.47	4	8.50	11.03	72.53	1.83	18.58
WDF(1)	10×4	20	3	0.05	310	0.97	4	1.25	4.06	73.89	0.17	29.41
WDF(2)	6×2	80	10	0.21	330	3.03	5	4.20	12.40	80.64	0.60	35.00
DPCM(1)	18×1	40	5	0.18	310	1.94	4	4.50	6.12	81.69	0.31	58.06
DPCM(2)	9×3	120	15	0.62	340	4.41	4	15.50	22.61	66.34	1.18	52.54
DPCM(3)	6×3	240	30	2.82	440	6.82	6	47.00	56.80	52.81	4.89	57.67
FLOYD(1)	11×2	40	5	0.18	310	1.94	4	4.50	6.12	81.69	0.31	58.06
FLOYD(2)	6×2	120	15	0.72	350	4.28	9	8.00	26.02	57.50	1.75	41.14
FLOYD(3)	3×3	240	30	2.05	480	6.25	13	15.76	59.00	50.85	4.62	44.37
Average ratio					–	3.38	–	11.14	–	70.10	–	44.13

operation in a TLP schedule with that in a List schedule and an IRP schedule, that is,

$$ratio = \frac{N_{avg}}{N_w}. \quad (10)$$

As we can see, it is clear that the number of write operation in a TLP schedule is less than that in a List schedule and an IRP schedule. The average ratios of the write operation of a TLP schedule to those of a List schedule and an IRP schedule are 19.50% and 54.65%, respectively.

To see the effects of different execution time and memory latency, we conducted a set of experiments, assuming that the processor is similar to NVIDIA GeForce 8800 GPU GTS. In NVIDIA GeForce 8800 GPU GTS, there are 12 cores. A set of parameters collected from NVIDIA GeForce 8800 GPU GTS by CACTI tools are shown in Table 3.

The experimental results are shown in Table 4. We can see that the TLP algorithm still outperforms the List scheduling and the IRP scheduling algorithms. The average ratios of the length of a TLP schedule to those of a List schedule and an IRP schedule are 3.38% and 70.10%, respectively. The average ratios of the write operation of a TLP schedule to those of a List schedule and an IRP schedule are 11.14% and 44.13%, respectively. Comparing Table 2 and Table 4, we can see that the TLP algorithm tends to create a large first level partition in order to compensate the long latency when the memory latency is increased, and improvement over the List scheduling and the IRP scheduling algorithms becomes more obvious.

7. Conclusion

In this paper, we propose a new MD loop scheduling algorithm called TLP. The algorithm employs a two level partition technique. TLP can reduce write operation and completely hide memory latency for multicore architectures. The experimental results show that our proposed algorithm is superior to the existing List, Rotation, PSP, and IRP algorithms. The TLP scheduling algorithm can reduce write operation to the main memory by $(100 - 54.65)\% = 45.35\%$ and reduce the schedule length by $(100 - 76.3)\% = 23.7\%$ compared with the IRP scheduling algorithm, the best known algorithm.

For further research, we plan to extend the TLP algorithm by investigating two intriguing issues. First, we will upgrade TLP to schedule tasks in MD DSP applications which finds an optimal solution. Second, we will apply the dynamic programming and the branch-and-bound approaches to scheduling MD DSP applications in heterogeneous computing systems. Furthermore, we will take account of the impact of register allocation on the schedule length of a benchmark in the future work. Registers have a number of advantages such as reducing scheduling length, saving storage, and improving the instruction-level parallelism. However, during compilation, the compiler must decide how to allocate variables to a small set of registers. Excessive parallelism will lead to a lot of spill code as variables cannot be stored simultaneously in the registers. It will affect the schedule length of a benchmark. There are two reasons: First, two variables/codes in use at the same time cannot be assigned to the same register without corrupting its value. Parallelism will increase the number of variables/codes in use. Second, the capacity of registers is limited. The variables and codes which can be stored in registers are limited. Excessive parallelism will increase the number of codes, which will result in a lot of spill codes and variables. It will increase the schedule length of memory part and slows down the execution speed of the compiled program. Therefore, we will exploit a high-performance partition scheme to avoid the impact of registers on the schedule length of a benchmark.

Acknowledgments

The authors would like to thank the two anonymous reviewers for their comments which have helped to improve the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), and the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61402400).

References

[1] Andrea Marongiu, Paolo Burgio, Luca Benini: Fast and Lightweight Support for Nested Parallelism on Cluster-based Embedded Many-cores. DATE 2012: 105–110.
[2] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, D. Burger, Clock rate versus IPC: the end of the road for conventional microarchitectures, in: Proceedings of the 27th Intl. Symp. on Computer Architecture, June 2000.
[3] J.A. Kahle, M.N. Day, H.P. Hopstee, C.R. Johns, T.R. Maeurer, D. Shippy, Introduction to the cell multiprocessor, IBM J. Res. Dev. 49 (2005).
[4] Fei Chen, Timothy W. O'Neil, Edwin H.-M. Sha, Optimizing overall loop schedules using prefetching and partitioning, IEEE Trans. Parallel Distrib. Syst. 11 (6) (2000) 604–614.
[5] C. Xue, Z. Shao, M. Liu, M. Qiu, Edwin H.-M. Sha, Optimizing nested loops with iterational and instructional retiming, J. Embedded Comput. (JEC) (2006).
[6] S. Tongsima, C. Chantrapornchai, N. Passos, Edwin H.-M. Sha, Efficient loop scheduling and pipelining for applications with non-uniform loops, IASTED Int. J. Parallel Distrib. Syst. Networks 1 (4) (1998) 204–211.
[7] Preeti Ranjan Panda, Nikil D. Dutt, A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications, in: Proceedings of the 1997 European Design and Test Conference (EDTC '97), pp. 1066–1409.
[8] Udayakumaran, Sumesh, Dominguez, Angel, Barua, Rajeev. Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems (TECS), pages. 472–511. 2006.
[9] Ke Bai, Jing Lu, Aviral Shrivastava, Bryce Holton, CMSM: an efficient and effective code management for software managed multicores, in: CODES+ISSS 2013.
[10] Jing Lu, Ke Bai, Aviral Shrivastava, SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs), in: DAC 2013.

[11] Ke Bai, Aviral Shrivastava, Automatic and efficient heap data management for limited local memory multicore architectures, in: DATE 2013.
[12] Q. Wang, N. Passos, Edwin H.-M. Sha, Minimization of memory access overhead for multi-dimensional DSP applications via multi-level partitioning and scheduling, IEEE Trans. Circuits Syst. II 44 (9) (1997) 741–753.
[13] C. Xue, Z. Shao, M. Liu, M. Qiu, E.H.-M. Sha. Loop scheduling with complete memory latency hiding on multi-core architecture, in: Proc. The 12th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2006), Minneapolis, MN, July 2006, pp. 375–382.
[14] A. Agarwal, D.A. Kranz, V. Natarajan, Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors, IEEE Trans. Parallel Distrib. Syst. 6 (1995) 943–962.
[15] M.E. Wolf, M.S.A. Lam, Data locality optimizing algorithm, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2, 1991, pp. 30–44.
[16] C. Xue, J. Hu, Z. Shao, E.H.-M. Sha, Iterational Retiming with Partitioning: Loop Scheduling with Complete Memory Latency Hiding in ACM Transaction on Embedded Computing System (TECS), 9(3), Feb. 2010, pp. 1–26.
[17] C. Xue, Z. Shao, M. Liu, E.H.-M. Sha, Iterational retiming: Maximize iteration-level parallelism for nested loops, Accepted, in: Proc. The 2005 ACM/IEEE/IFIP International Conference on Hardware – Software Codesign and System Synthesis (ISSS-CODES'05), New York, New York, Sept. 2005.
[18] Z. Wang, M. Kirkpatrick, E.H.-M. Sha, Optimal two level partitioning and loop scheduling for hiding memory latency for DSP applications, in: Proc. ACM 37th Design Automation Conference, Los Angeles, California, June 2000.
[19] Jenny Qingyan Wang, Edwin Hsing-Mean Sha Nelson Luiz Passos, Minimization of Memory Access Overhead for Multi-Dimensional DSP Applications via Multi-Level Partitioning and Scheduling. IEEE Transactions on Circuits and Systems II, 44(9), September 1997, pp. 741–753.
[20] N. Passos, Edwin Hsing-Mean Sha, Achieving full parallelism using multi-dimensional retiming, IEEE Trans. Parallel Distrib. Syst. 7 (11) (1996) 1150–1163.
[21] L.F. Chao, Edwin Hsing-Mean Sha, Retiming and Unfolding Data-Flow Graphs. International Conference on Parallel Processing, St. Charles, Illinois, August 1992, pp. II 33–40.
[22] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A durable and energy efficient main memory using phase change memory technology, in: ISCA 09, Austin, Texas, USA, 2009.
[23] M.K. Qureshi, V. Srinivasan, J.A. Rivers, Scalable High Performance Main Memory System Using Phase-Change Memory Technology, ISCA09, June 20–24, 2009, Austin, Texas, USA.
[24] F. Dahlgren, M. Dubois, Sequential hardware prefetching in shared-memory multi-processors, IEEE Trans. Parallel Distrib. Syst. (1995) 733–746.
[25] M.K. Tcheun, H. Yoon, S.R. Maeng, An adaptive sequential prefetching scheme in shared-memory multiprocessors, in: Proceedings of the International Conference on Parallel Processing, 1997, pp. 306–313.
[26] J. Philbin, J. Edler, O. Anshus, C. Douglas, K. Li, Thread scheduling for cache locality, Comput. Architect. News 24 (1996) 60–71.
[27] Z. Wang, E.M. Sha, Y. Wang, Partitioning and scheduling DSP applications with maximal memory access hiding, Eurasip J. Appl. Signal Process. 9 (2002) 926–935.
[28] Z. Wang, Q. Zhuge, E.-M. Sha, Scheduling and partitioning for multiple loop nests, in: Proceedings of the International Symposium on System Synthesis, 2001, pp. 183–188.
[29] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories, ACM SIGPLAN PPoPP 2008.
[30] M. Qiu, J. Wu, Loop Scheduling and assignment with prefetching to minimize energy while hiding memory latencies, in: ACM GLSVLSI, Orlando, Florida, May 2008.

Yan Wang received her BSc in Information management and Information techniques from Shenyang Aerospace University in 2010. She is currently a PhD candidate in Hunan University, China. Her research interest includes modeling and scheduling for parallel and distributed computing systems, high performance computing.

Kenli Li received the PhD in computer science from Huazhong University of Science and Technology, China, in 2003, and the MSc in mathematics from Central South University, China, in 2000. He was a visiting scholar at University of Illinois at Champaign and Urbana from 2004 to 2005. Now, he is a professor of Computer science and Technology at Hunan University, a senior member of CCF. His major research includes parallel computing, Grid and Cloud computing, and DNA computer.

Keqin Li received B.S. degree in computer science from Tsinghua University, Beijing, China, in 1985, and Ph.D. degree in computer science from the University of Houston, Houston, Texas, USA, in 1990. He was an assistant professor (1990–1996), an associate professor (1996–1999), a full professor (1999–2009), and has been a SUNY distinguished professor of computer science since 2009 in State University of New York at New Paltz. He was the acting chair of Department of Computer Science during Spring 2004. He is also an Intellectual Ventures endowed visiting chair professor at the National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China.