Partition Scheduling on Heterogeneous Multicore Processors for Multi-dimensional Loops Applications

Yan Wang, Kenli Li & Keqin Li

International Journal of Parallel Programming

ISSN 0885-7458
Volume 45
Number 4

Int J Parallel Prog (2017) 45:827-852
DOI 10.1007/s10766-016-0445-2

Springer

Int J Parallel Prog (2017) 45:827–852
DOI 10.1007/s10766-016-0445-2

Partition Scheduling on Heterogeneous Multicore Processors for Multi-dimensional Loops Applications

Yan Wang[1,2] · **Kenli Li**[2] · **Keqin Li**[2]

Received: 10 November 2015 / Accepted: 7 July 2016 / Published online: 15 July 2016

Abstract This paper addresses the scheduling problem for multi-dimensional loops applications on heterogeneous multicore processors. In the multi-dimensional loops scheduling problem, a significant issue is how to hide memory latency to reduce the schedule length. With the increasing CPU speed, the gap between the processor and memory performance is an important bottleneck for modern high-performance computer systems. To solve the bottleneck problem, a variety of techniques have been studied to hide memory latency from intermediate fast memories (caches) to various prefetching and memory management techniques. Although there are a lot of algorithms in the literature to solve the scheduling with memory management problem for multiprocessor systems, they may not deliver good quality with high performance for heterogeneous multicore processors. In this paper, we first propose a scheduling algorithm *Recom_Task_Assign* to reduce the write activities to main memory. Then, in conjunction with the *Recom_Task_Assign* algorithm, we present a new partition scheduling algorithm called heterogeneous multiprocessor partition (HMP) based on the prefetching technique for heterogeneous multicore processors, which can hide memory latencies for applications with multi-dimensional loops. This technique takes advantage of memory access pattern information and fully considers the heterogeneity

✉ Yan Wang
bessie11@yeah.net

Kenli Li
lkl@hnu.edu.cn

Keqin Li
lik@newpaltz.edu

1 School of Computer Science and Educational Software, Guangzhou University, Guangzhou, China

2 College of Information Science and Engineering, Hunan University, Changsha 410082, China

of processors to achieve high processor utilization. Our HMP algorithm selects the appropriate partition size and shape according to different processors, which increases processor utilization and reduces memory latency. Experiments on DSP benchmarks show that our algorithm can efficiently reduce memory latency and enhance parallelism compared with existing methods.

Keywords Heterogeneous multicore processor · Memory latency · Multi-dimensional loops · Scheduling

1 Introduction

1.1 Motivation

Heterogeneous multicore processor systems, which adopt multicore processors and systems on chip, have gained popularity to meet ever increasing demands of high-performance computing. One of the increasingly significant performance bottlenecks of modern high-performance heterogeneous multicore processor system is memory latency, due to the gap between the processor and memory speeds continues to grow. Minimizing schedule length and power consumption are also big challenges to heterogeneous multicore processors. As a result, a variety of software managed memory techniques [6,24,31] have been studied to find an efficient solution for obtain a high performance. Data prefetching scheme [8] is an efficient software managed memory technique to reduce memory latency via overlapping memory access operations with processor computations. In this paper, we address the multi-dimensional scheduling problem based on the software prefetching technique to hide memory latency. A large number of applications, such as DSP application, involve multi-dimensional problems. The multi-dimensional problems involve more than one dimension and are characterized by nested loops with uniform data dependencies. These characteristics of multi-dimensional problems make the issue of loop scheduling essential to improve performance of heterogeneous multicore processor systems. To find a well planned solutions, we propose a scheme, i.e., the heterogeneous multiprocessor partition (HMP) algorithm. The HMP algorithm generate a multi-dimensional loop scheduling with software prefetching scheme in such way to achieve full parallelism at iteration level and to hide memory latency completely.

The HMP algorithm can be applied in heterogeneous multicore processor systems with three levels of memory. These three levels of memory are abstracted as local memory, main memory, and remote memory. We assume that each processor consists of multiple cores and multiple memory units. The processing core works on computations and the memory units perform memory operations such as prefetch data from various types of memories. All processors are connected by a shared bus or a high-speed channel. The real-world examples of such systems are blade servers and Tianhe-1. Given an application, the HMP algorithm generates a two-part schedule, one for the processing cores and the other for the memory units. In the processing core part, the HMP algorithm will call *Recom_Task_Assign* algorithm, is developed

to reduce the execution cost of each iteration. The memory part of a schedule adopts the partition techniques based on software prefetching scheme. Memory operations are arranged by the memory part of a schedule, so that all the data of computational tasks are prefetched into the main memory in advance. Since both parts of a schedule are executed simultaneously, the memory latency is hidden by overlapping with the processing core executions.

1.2 Related Work

There have been a lot of research effort on allocating and scheduling in heterogeneous multiprocessor systems [2,9,12,13,15,35,36]. The loop scheduling problem driven by overlapping computation and communication in heterogeneous multiprocessor systems has been studied in [3,4,34,37,40]. In these work, the authors exploited a scheduling and workload balancing scheme for execution of loops having dependent or independent iterations on heterogeneous multiprocessor systems. However, when performing multi-dimensional loop applications in a heterogeneous multiprocessor system, the effect of most previous work is not satisfactory. Also, the scheduling schemes do not make full use of memory requirements and information of a heterogeneous multiprocessor system.

In order to better overlap computation and communication for high performance, many researches have turned to software managed memory techniques for solutions [5,16,19,23,30,33]. In these work, prefetching is a significant technique that can tolerate the large latency of memory access for achieving high processor utilization. These prefetching techniques can be classified into three categories, i.e., those based on software [1,20,24], hardware [8,11,14], or both [7,41,43]. Software-based prefetching techniques depend on compiler techniques to analyze a program statistically and insert explicit prefetch instructions into the program code. In hardware-based prefetching techniques, the prefetching activities are controlled solely by the hardware and depend on the dynamic information available during program execution. However, the three prefetching techniques scarcely consider the processor part of the scheduling. In heterogeneous multicore processor systems, solely considering the prefetching is not enough for improving the overall system performance.

Both processor part and memory part are importance for improve performance, there have been various researches considering both scheduling and software prefetching scheme in multi-dimensional problems at the same time [21,25,27,28]. In [26], the authors considered the problem of scheduling parallel loops whose iterations operate on large array data structures and proposed a general parallel loop implementation template for message-pass distributed memory multiprocessors. Liu and Abdelrahman [22] develop a compiler transformation, which overlaps the communication time resulting form memory access with the computation time in parallel loops to effectively hide the latency of the remote accesses, that improves the performance of parallel programs on Network-of-Workstation shared memory multiprocessors. Qiu et al. [29] proposed an efficient algorithm, Energy Aware Loop Scheduling with Prefetching and Partition to maximize energy saving while hiding memory latency with the combination of loop scheduling, data prefetching, memory partition, and heterogeneous memory module type assignment.

In this paper, the partition technique is incorporated into the HMP algorithm based on the prefetching technique to hide memory latency. Partition techniques divide the entire iteration space into multiple partitions, and then execute partitions one by one. Partitioning to minimize communication in distributed memory multiprocessors to support task level parallelism for real-time applications were investigated in [32]. In [38], the partitioning techniques are adopted to increase data locality. In [10], the authors used the partitioning technique to reduce memory stalls and improve computation parallelization. In [39], iterational retiming is used in conjunction with partition techniques. The authors proposed a new loop scheduling with memory management technique that can completely hide memory latency for applications with multi-dimensional loops on architectures like the CELL processor. However, these previous techniques only explore instruction level parallelism and are not able to be applied in heterogeneous multiprocessors.

1.3 Our Contributions

In the present paper, we propose a new loop partition scheduling algorithm with memory management techniques to hide memory latency and increase parallelism for multi-dimensional loops applications on heterogeneous multicore processor systems. In our HMP algorithm, the *Recom_Task_Assign* algorithm is used in conjunction with a partition technique. The *Recom_Task_Assign* algorithm can obtain a good computational task assignment for each multicore processor. Then, the HMP algorithm partitions the iteration space into different partitions according to the computational task assignment on each multicore processor. We experiment with the HMP algorithm on a set of benchmarks and compare the HMP algorithm with the IRP algorithm [39] and the List scheduling algorithm, which is the most traditional algorithm. According to the experimental results, the HMP algorithm has better performance compared with the IRP and the List scheduling algorithms. The average schedule length obtained by the HMP algorithm is 48.67 and 16.8 % of that obtained by using the IRP and the List scheduling algorithms, respectively. The experimental results show that overlapping processor computation time and memory access time by partitioning the iteration space is essential for hiding the memory latency.

1.4 Computational Model

The major contributions of this paper include the following aspects.

- To the best of the authors' knowledge, this is the first paper to explore the memory latency problem for heterogeneous multicore processor systems with multiple levels of memory.
- By combining recomputing and a one-dimensional retiming technique, we propose a novel loop scheduling technique to reduce memory store activity.
- We propose a new multi-dimensional loops scheduling algorithm with memory management techniques, i.e., the HMP algorithm, for heterogeneous multicore processor systems, which can generate a two-part schedule and effectively hide memory latency.

The rest of this paper is organized as follows. The computational model and the architecture model used in this paper are described in Sect. 2. The main algorithms are developed and illustrated in detail in Sect. 3. The relationship between partition size and memory requirements is investigated in Sect. 4. The experiments and results are presented in Sect. 5. Finally, we conclude our paper in Sect. 6.

2 Models

In this section, we first describe the computational model for our algorithms. Then, we describe our heterogeneous multicore processor system model.

In this subsection, we describe the *multi-dimensional data flow graph* (MDFG), which is used to model a uniform nested loop. An MDFG $G = (V, E, d, t, w)$ is a node weighted and edge weighted directed graph, where V is a set of computational nodes; $E \subseteq V \times V$ is a set of edges that describe the precedence constraints among nodes in V; 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; t is a function from V to positive integers, representing the computation time of each node; and w is a function from E to positive integers, representing the amount of data required to be transmitted through each edge. In this paper, we consider two-dimensional DFG (2DFG) applications. We use $d(e) = (d_i, d_j)$ as a general formulation of any delay shown in a 2DFG. A two-dimensional loop program is shown in Fig. 1a. In this loop program, there are 4 computations, which are calculating $A[i, j]$, $B[i, j]$, $C[i, j]$, and $D[i, j]$, respectively. The corresponding equivalent MDFG is shown in Fig. 1b. In the MDFG, the nodes represent the corresponding computations in the original loop program, and an edge between two nodes means the precedence constraint.

An *iteration* is the execution of the loop body once. The computation time of the longest path without delay is called the *iteration period*. For example, the iteration period of the MDFG shown in Fig. 1b is 3, which is from the longest zero-delay path including 3 nodes, that are A, B, and C, respectively. We use $\boldsymbol{i}$ to identify iteration, equivalent to the nested loop index and starting from (0, 0). 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. The horizontal axis corresponds to the j index

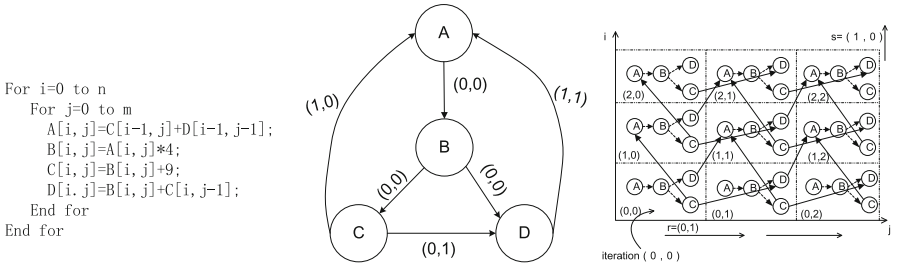

Fig. 1 The MDFG representation of the IIR filter

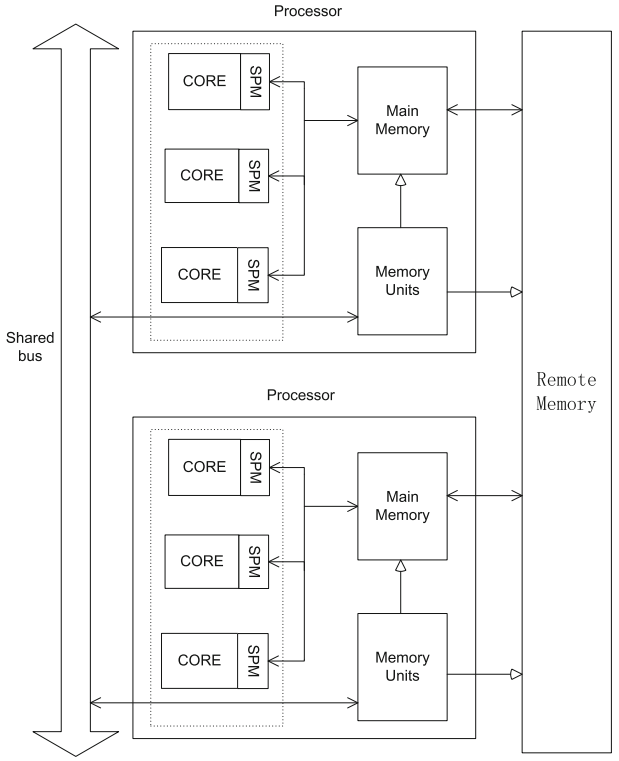

Fig. 2 The architecture diagram

which is the inner-loop, and the vertical axis corresponds to the i index which is the outer-loop. Figure 1c illustrates a representation of the iteration space for the MDFG shown in Fig. 1b. The solid vectors represent the inter-iteration data dependency. In this paper, the proposed algorithm is based on 2DFG. In 2DFG, an iteration node can be represented as $iteration(i, j)$. If a task in $iteration(i, j)$ depends on another task in $iteration(x, y)$, the dependence vector is $d(e) = (i - x, j - y)$. An edge with delay $d(e) = (0, 0)$ represents an intra-iteration dependence. For example, the vector from node C to node D with delay vector (0,1) means that the computational node D in $iteration(0, 1)$ depends on the computational node C in $iteration(0, 0)$.

A scheduling vector s is the normal vector such that $s \cdot d \geq 0$ for any delay d in the MDFG. The vector defines a sequence of execution of an iteration space for a set of parallel equitemporal hyperplane. In this paper, the schedule vector is $s = (1, 0)$, because we execute a given nested loop in a row-wise fashion.

2.1 Architecture Model

Our technique is designed for use in a system containing multiple multicore processors and special hardware called memory units inside each processor. The multiple

multicore processor system shown in Fig. 2 can be homogeneous or heterogeneous, in the sense that the numbers of cores in the processors can be identical or different, and the core speeds of different processors can be identical or different. A shared bus is presented to facilitate the data exchanges among all processors. A three-level memory hierarchy is adopted in our architecture model. Each processing core is equipped with a local memory called scratch pad memory (SPM), which has the tightest memory size constraint and the fastest accessing speed. Associated with each processor is a main memory. Accessing the main memory is slower than accessing the local memory. There is also a large multi-port remote memory. However, The remote memory has the slowest accessing speed. In order to minimize the total cost of accessing data and hide memory latency, our algorithm is to load data into SPM before its explicit access take place. Therefore, we can overlap the computations and access operations to reduce the total execution time.

Our scheme includes two phases. The first phase aims to reduce the execution cost of each iteration. The second phase aims to overlap processing computation and access operation between the main memory and the remote memory. There are five types of memory instructions, i.e., load, store, prefetch, write, and cross instructions, which are added to the code when our scheme is complied. These memory instructions are supported by the memory units. The load instructions are to load data from the main memory to a processing core's local memory. The store instructions are to store data from a processing core's local memory into the main memory. The prefetch instructions prefetch data from the remote memory to the main memory. The write instructions write data back to the remote memory for future accessing operations. The cross instructions migrate data from one processor's main memory to another processor's main memory. All of them are issued to make sure that those data which will be referenced soon will appear in the local memory of the corresponding processing core.

It is significant to note that the lower level memories in the architecture model are SPM. There are two reasons. First, the local memory cannot be pure caches, because we do not consider the cache consistency and cache conflict in this paper. Second, SPM is a small on-chip memory component that is managed by software. Furthermore, SPM is efficient in performance and power compared with hardware-managed cache. Our architecture is a general model. In a homogeneous multicore processor system, a real implementation was done in the CELL processor. The heterogeneous multicore processor architecture is similar to the super-computing system such as the Tianhe-1 system, and a blade server system. Samsung Exynos 5 Octa [17] is a heterogeneous multicore processor architecture. The usual setup involves utilizing ARM big.LITTLE technology with two groups of cores, i.e., one set of low-power cores, usually Cortex-A7, and one set of more powerful cores, at the moment Cortex-A15. In Samsung's initial software implementation, only half of the eight cores can be used at a time, depending on the workload required.

3 Algorithm

In this section, we first discuss task assignment and scheduling of the processor part and present the $Recom\_Task\_Assign$ algorithm to reduce the execution cost for the

processor part of a schedule. Then, we introduce the loop partition technique. Next, we illustrate the heterogeneous multicore processor scheduling framework. Finally, we present the heterogeneous multiprocessor partition (HMP) algorithm.

3.1 Task Assignment and Scheduling

In this subsection, we propose an efficient algorithm called *Recom_Task_Assign* to reduce the execution cost of each iteration. Before we present the *Recom_Task_Assign* algorithm, we will introduce the rotation technique and the recomputing technique.

The *one-dimensional retiming* technique is used in our algorithm. It evenly distributes the delays in the MDFG to optimize the iteration period. Then, we can rearrange the sequence of computations of each iteration. Given an MDFG $G = (V, E, d, t, w)$, the retiming function $r(v)$ from V to integers is the number of delays moved through node $v \in V$. The technique of retiming moves delays around in the following way, i.e., delays are drawn from each of the incoming edges of v, and then pushed to each of the outgoing edges of v, or vice versa. Let $G_r = (V, E_r, d_r, t, w)$ denote the retimed graph of G with retiming r. The number of delays can be calculated as $d_r(e) = d(e) + r(u) - r(v)$ for any edge $e(u \rightarrow v) \in E_r$ in G_r. We have $d_r(e) \geq 0$ for any edge, if the retiming r is legal, and the total number of delays remains constant in the MDFG. It is important to note that one-dimensional retiming is used instead of multi-dimensional retiming for maintaining the row-wise execution. This is because multi-dimensional retiming will change the execution sequence of the iterations and the schedule vector.

The recomputing technique, which is to further reduce the total completion time and number of store activities, is used in our algorithm. The idea is to discard some of the computed data, so that they are not written to main memory. The recomputing techniques reduce the number of store activities in the following way, i.e., when the data is requested by latter tasks, we will load the necessary operands and recompute the data. The *recomputing minimize write* algorithm [18] further reduces store activity on main memory by finding appropriate nodes to duplicate. The main idea of the *recomputing minimize write* algorithm is for each store operation to calculate the cost of recomputing the related nodes that produce this dirty page. If the cost of recomputing is lower than the cost of dirty eviction, we discard the dirty eviction of the dirty page and recompute the corresponding nodes before future load of the dirty page.

The goal of *Recom_Task_Assign* is to reduce the execution cost of each iteration. The execution cost consists of computational cost and data accessing cost between local memory and main memory. Given an MDFG $G = (V, E, d, t, w)$, the number of processing cores, and the size of SPM, algorithm *Recom_Task_Assign* generates a heuristic assignment and scheduling as shown in Algorithm 3.1.

The *Recom_Task_Assign* algorithm consists of several steps. First, one-dimensional rotation is applied to transform the graph into a retimed graph so that the iterations can be scheduled in parallel. Second, to satisfy the input condition of the *recomputing minimize write* algorithm, we eliminate edges with inter-iteration dependence to re-construct a DFG $G'_r = (V, E'_r, d'_r, t, w')$, which does not have any cycles. For example, Fig. 3b is re-construct graph from Fig. 3a. Third, we add new virtual nodes

Algorithm 3.1 Recom_Task_Assign algorithm

Require: An MDFG $G = (V, E, d, t, w)$, the number of cores, the capacity of each SPM of each processor.
Ensure: An assignment of tasks in each iteration, and the schedule length of the processor part on a processor.
1: Call the one-dimensional retiming algorithm to obtain a retimed MDFG $G_r = (V, E_r, d_r, t, w)$.
2: Reconstruct a graph DFG $G'_r = (V, E'_r, d'_r, t, w')$ by eliminating all edges except some edges with a delay $d(e) = (0, 0)$.
/*Add new virtual nodes to guarantee the right data access:*/
3: **for** each node v in G'_r **do**
4: **if** there exist incoming edges of the node have been eliminated **then**
5: **for** each eliminated incoming edges **do**
6: adding a virtual parent node
7: **end for**
8: **end if**
9: add the right weight for each corresponding added edges.
10: **end for**
11: Do task assignment and scheduling via the *recomputing minimize write* algorithm: for each node, regard the weight of incoming edge as its input data, and treat the weight of outgoing edges as output data;
12: **for** each computational node $v_i \in V$ **do**
13: $pn_i \leftarrow$ the number of pages in the local memory needed to execution the node;
14: $fp_i \leftarrow$ the number of free pages of local memory at the load step;
15: $cp_i \leftarrow$ the number of clean pages of local memory at the load step;
16: **end for**
17: call the *recomputing minimize write* algorithm to obtain task assignment and scheduling.

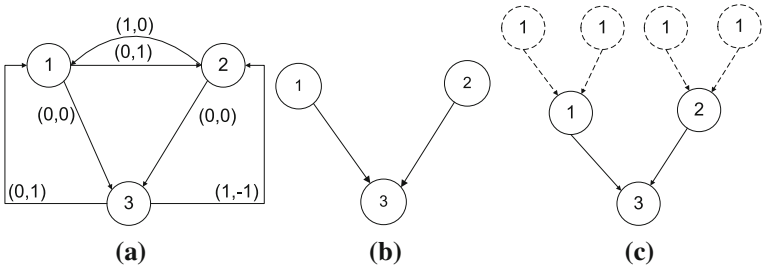

Fig. 3 An example of Algorithm 3.2

to guarantee the right data access, which does not affect the schedule. Adding virtual nodes must follow the following principle: for each eliminated edge, a parent node is added first for its corresponding outgoing node, And then the weight is added for its corresponding edge. For example, we add some virtual nodes for Fig. 3b, the result is shown in Fig. 3c. Finally, the *recomputing minimize write* algorithm is applied to obtain task assignment and scheduling. How to re-construct a graph and add new virtual nodes are shown in algorithm *Recom_Task_Assign* in detail. The retiming and recomputing techniques are explained in detail in the above paragraphs.

In the algorithm, it takes $O(|V||E|)$ time to retime the MDFG, where $|V|$ is the number of nodes and $|E|$ is the number of edges. Reconstructing the retimed MDFG takes $O(|V| + |E|)$ time. The time for the *recomputing minimize write* algorithm is $O(n^2)$, where n is the number of steps in the schedule without the recomputing technique. Therefore, the complexity of $Recom\_Task\_Assign$ is $O(|V||E| + |V| + |E| + n^2) = O(|V||E| + n^2)$.

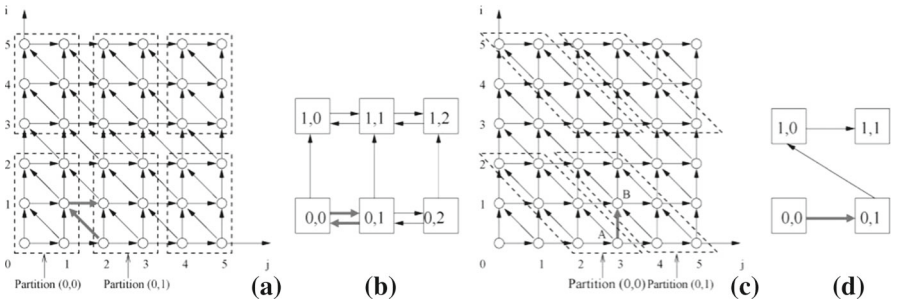

Fig. 4 **a** An illegal partition of the iteration space. **b** The partition dependency graph of **a**. **c** A legal partition of the iteration space. **d** The partition dependency graph of **c**

3.2 Partition Iteration Space

Regular execution of an entire multi-dimensional loop is performed in either a row-wise or a column-wise manner until we reach the end of the row or column. However, it takes a lot of time and generates huge amount of write activity. In this paper, for overcoming these disadvantages, a partition technique is applied instead of the regular execution. We first partition the iteration space, and then execute the partitions one by one. We use two boundaries of a partition called *partition vectors*, i.e., P_i and P_j, to identify a parallelogram as the partition shape. Without loss of generality, the angle between P_i and P_j is less than 180°, and P_j is clockwise of P_i. 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, partitions must be legal partitions, which require that there cannot exist cycles (two-way dependencies) among partitions. In other words, the partition vectors cannot be arbitrarily chosen due to the dependencies in MDFG. For example, we consider the iteration space of Fig. 3. If we partition the iteration space with partition vectors $P_i = (2, 0)$ and $P_j = (0, 1)$ shown in Fig. 4a, the partition 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) as shown in Fig. 4b. In contrast, let us partition the iteration space with partition vectors $P_j = (0, 1)$ and $P_i = (2, 2)$ as shown in Fig. 4c. The partition is legal because there is no two-way dependency as shown in Fig. 4d. To obtain legal partitions, the partition vectors P_i and P_j should surround all vectors d(e) and must satisfy the following property to guarantee executing partitions one by one.

Property 1 *It is a legal partition shape if and only if the cross products* $d(e) \times P_j \leq 0$ *and* $d(e) \times P_i \geq 0$*, for all delay dependence* $d(e)$ *in MDFG.*

From Property 1, we can obtain the following result.

Theorem 1 *All delay dependencies in an MDFG are in the counterclockwise region of partition vector* P_j *and are in the clockwise region of partition vector* P_i*.*

Proof For two vectors $p1$ and $p2$, the magnitude of the cross product, denoted by $p1 \times p2$, is used to determine the relative position of the two vectors. If $p1 = (p1.i, p1.j)$

and $p2 = (p2.i, p2.j)$, then $p1 \times p2 = p1.j \times p2.i - p2.j \times p1.i$. If $p1 \times p2$ is positive, then $p1$ is clockwise related $p2$ with respect to the origin $(0, 0)$. On the contrary, if $p1 \times p2$ is negative, then $p1$ is counterclockwise related to $p2$. From Property 1, we know that the cross product $d(e) \times P_j \leq 0$ for all delay dependencies in MDFG. Therefore, all delay dependencies in an MDFG are in the counterclockwise region of partition vector P_j. In a similar way, we can obtain that all delay dependencies in an MDFG are in the clockwise region of partition vector P_i.

The objective of Property 1 and Theorem 1 is to legal partition iteration space. The advantage of the partition technique is that the data locality and parallelism are improved. Assume that we do not partition the iteration space and instead, execute the iterations in the row-wise order. It is possible that, after finishing one row/column of iterations, generating huge amount of data will create a large memory latency when we start a new row/column where those data are needed. Provided that the total iteration space is divided into partitions, the partitions is executed in turn left to right, and within each partition, iterations are executed in row-wise order. By this way, those new data are stored in the lower memory and prefetching operations are reduced.

3.3 HMP Algorithm Framework

HMP generates a schedule consisting of two parts: the processor part and the memory part. Incorporating load/store operations into the processor part, the processor part of a schedule for one iteration is generated by using the *Recom_Task_Assign* algorithm to increase parallelism and reduce the scheduling length. Since the *Recom_Task_Assign* algorithm uses a one-dimensional retiming technique to reduce the iteration period, it produces more inter-iteration dependencies, which require more memory prefetching operations and create a large memory latency. Therefore, the memory part of the schedule is lengthened. In order to hide memory latency, the partition techniques is applied in the HMP algorithm.

We call the partition just finished execution on a processor the *last partition*, the partition that is being executed on the same processor the *current partition*, and the partition that will be executed next on the same processor the *next partition*. All other partitions which have not been executed are called *other partitions*. A diagram is shown in Fig. 5a. When scheduling the memory part, we should consider the two types of delay dependence. The first type is the nonzero delay transits from the current partition into other partitions. For this type of delays, a prefetching and a write-back operations are needed. The second type is the directed edge from the current partition to the next partition. For this type of delays, a store operation is needed. When scheduling the memory part, we expect to prefetch all data needed by the next partition into the main memory at the same time as the processing core computations are being executed for the current partition. Figure 5b shows an example of our overall schedule.

In reality, different processors have different configurations in a heterogeneous multiprocessor system. It is necessary to study partition scheduling with multiple heterogeneous multicore processors. In the HMP algorithm, we will determine the partition size and shape for each processor. The values of partition sizes for different multicore processors are different, although all partition shapes are the same. In order

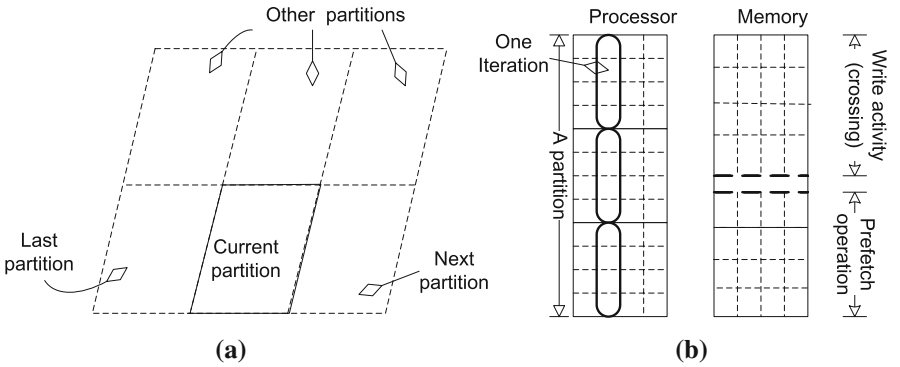

Fig. 5 **a** A representation of last partition, current partition, next partition, and other partitions. **b** The HMP schedule for a partition

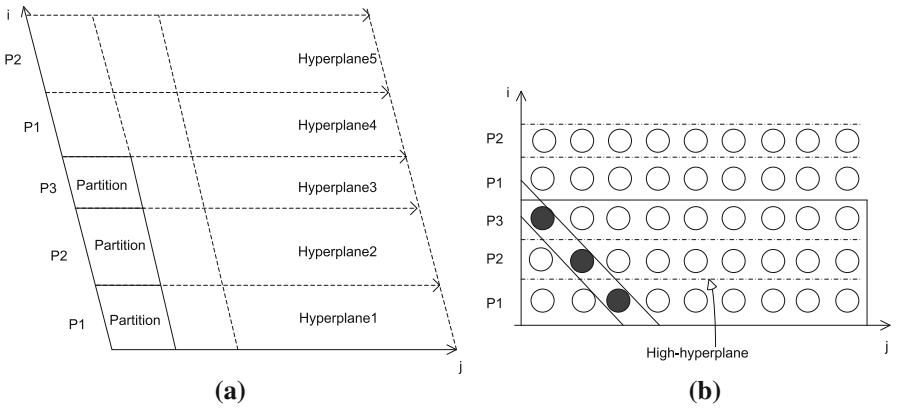

Fig. 6 The multiprocessor partition hyperplane

to reduce the waiting latency, we guarantee the same scheduling length of processor part for all partitions.

Figure 6 shows the multiprocessor partition scheduled in order. A hyperplane is denoted by hp_i. We use the value of $hp_i \mod n$ to determine which processor to perform partitions on each hyperplane, where n is the number of processors. If $hp_i \mod n = k$, all partitions in this hyperplane will be executed on processor $P(k)$. The example diagram is shown in Fig. 6a. The execution order of hyperplanes as follows. We first execute the n hyperplanes, then we proceed to the next n hyperplanes. We can treat each partition as a dot, as shown in Fig. 6b. We treat the partitions which will be executed at the same time as a cluster partition, and treat these hyperplanes which will be executed at the same time as a high-hyperplane. For example, the black dots in Fig. 6b represent the partitions in the same cluster. Then, we execute the cluster partitions from left to right in the P_j direction, and the next high-hyperplane along the direction of vector perpendicular to P_j. The order of partitions scheduling must satisfy the following condition, i.e., if there are two partitions, $partition(k, v)$ and $partition(u, v)$, which are in the same high-hyperplane,

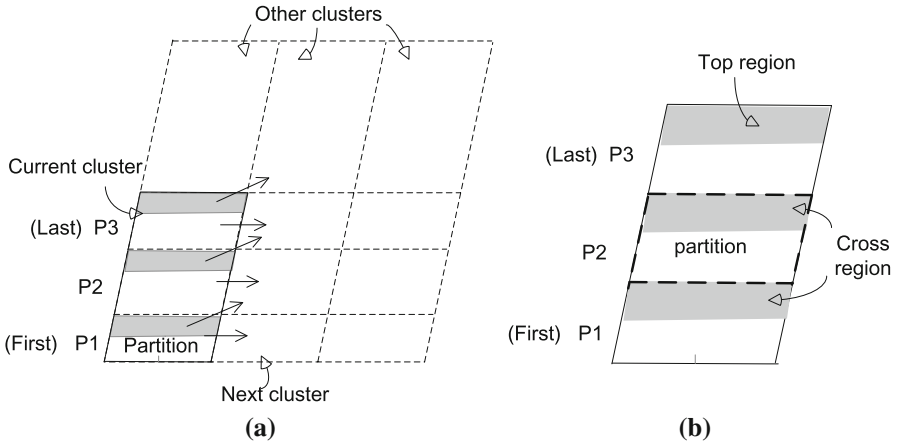

Fig. 7 The different kinds of delay dependence

and $u < k$, the $partition(u, v)$ must be executed no earlier than $partition(k, v)$. Therefore, we use two cluster vectors, $(0, 1)$ and $(n, -2n)$, to determine the cluster shape.

When considering the *cluster*, we treat *partitions* as the basic integral units. Then, the cluster will be a combination of a number of partitions based on the number of processors. For instance, the cluster size in Fig. 7 is 3×1, because there are three heterogeneous processors. We call the cluster currently being executed the *current cluster*, the cluster that will be executed next the *next cluster*, and all other clusters which have not been executed the *other clusters* (see Fig. 7). In the case of multi-processor partitions, there will be two kinds of delay dependence, which need to be treated differently when scheduling the loop. The first kind is the delay dependence that goes to the other cluster from the current cluster. For the delay dependence, it will be treated with a write operation in order to retain the corresponding data for near future use. The second kind is the delay dependence that goes to the next cluster which will be executed in different processor from the current cluster. For this delay dependence, the cross operations between two processors are needed.

3.4 HMP Scheduling Algorithm

In this subsection, we illustrate the *heterogeneous multiprocessor partition* (HMP) algorithm, which consists of two parts, the processor part and the memory part. The processor part, incorporated with load/store operations, of the schedule for iterations is generated by using the *Recom_Task_Assign* algorithm. The *Recom_Task_Assign* algorithm is a loop pipelining technique which introduces the retiming scheduling and recomputing technique to obtain a more compact scheduling with less store operation. The *Recom_Task_Assign* algorithm is described in detail in Sect. 3.1. In heterogeneous multiprocessor systems, we use *Recom_Task_Assign* to obtain a schedule for an application on each multicore processor. Since different processors have different

configurations in a heterogeneous multiprocessor system, different multicore processors show different scheduling sequence and length for each iteration. Therefore, the partition sizes in different hyperplane show different values.

Algorithm 3.2 Heterogeneous multiprocessor partition (HMP) algorithm

Require: An MDFG $G = (V, E, d, t, w)$, the number of processors, the number of cores of each processors, the capacity of each SPM of each processors.

Ensure: A heterogeneous multiprocessor partition schedule that hides memory latency, increases parallelism, and reduces the schedule length.

/* determine the direction vector, P_i' and P_j', of partition vectors P_i and P_j */
1: **for** each delay dependence $d_i(e)$ **do**
2: **if** $d(e) \times d_i(e) \geq 0, \forall d(e)$ in MDFG **then**
3: $P_i' = \frac{d_i(e)}{|d_i(e)|}$
4: **end if**
5: **end for**
6: $P_j' = (0, 1)$;
7: $f_j = k$,where (x, k) is the maximum innermost loop delay dependence;
8: $P_j = f_j \times P_j' = (0, k)$;
9: **for** each processor $P(h)$ **do**
10: call the $Recom\_Task\_Assign$ algorithm to obtain the task assignment and schedule length $L(h)$ of each iteration;
11: **end for**
12: $L_{max} \longleftarrow$ the maximum $L(h)$ in the schedule length sets;
13: obtain the partition size f_{max} based on the Theorem 4;
14: **for** each processor $P(h)$ **do**
15: $f(h) = L_{max} \times f_{max}/L(h)$;
16: $P_i(h) = f(h) \times P_i'$;
17: **end for**
18: do multiprocessor partition scheduling;
19: do memory part scheduling.

Scheduling of the memory part consists of several steps. First, we need to decide a legal partition vector direction. Second, partition sizes for different processors are calculated to ensure an optimal scheduling and guarantee that the same schedule length between two partitions should be executed on different processors. Third, we call multiprocessor partition frame to obtain clusters. Fourth, both the processor part and the memory part of a schedule are generated. We will illustrate these steps in detail.

For one partition should be performed on processors $P(h)$, the partition is identified by two partition vectors, $P_i(h)$ and P_j, where $P_i(h) = P_i' \times f(h)$ and $P_j = P_j' \times f_j$. While P_i' and P_j' are the direction vectors and determine the shape of partition, f_j and $f(h)$ determine the size of a partition based on processor $P(h)$. How to choose the vectors P_i' and P_j' are discussed in Sect. 3.2 in detail. The HMP algorithm shows how to choose f_j and $f(h)$. In the algorithm, the partition vector P_j is constant for all partitions. The direction vector is $P_j' = (0, 1)$, and the size f_j is equal to the value of the maximum innermost loop delay. Therefore, we will pay more attention on how $f(h)$ is chosen to achieve the goal of complete memory latency hiding, which means that the schedule length of the memory part will always be no more than the schedule

length of the processor part, and the schedule length of two different partitions based on different processors show the same value. How $f(h)$ is chosen is discussed in detail in Sect. 4.

After obtaining the partition directions and sizes, we can begin construct the processor part and the memory part of a schedule. Prefetching operations are scheduled as early as possible, because they do not have any data dependence. Therefore, the prefetching for the current partition is scheduled at the end of the last partition. Write-back and crossing operations have the data dependencies from the processor part of the schedule. Therefore, the write-back and crossing operations must be scheduled after the corresponding computational task is finished. In our paper, the write-back and crossing operations for the current partition is scheduled at the beginning of the next partition. In the processor part of a schedule, the execution time for all partitions are the same.

4 Partition Size and Memory Requirements

In this section, we will discuss how partition sizes are chosen to completely hide memory latency.

To determine the partition size, we will first define the number of prefetching operations given a partition size $f(h)$ and f_j. The number of prefetching operations can be approximated by computing the shaded area, as shown in Fig. 8, with respect to every inter-iteration delay vector $d(e) \in D$. Consider a delay dependence $d(e) = (d_i, d_j)$. All of its duplicate vectors originating in the region *PQRS* will enter other partitions, which is when the write-back and prefetching operations are needed when the two partitions are located in two different clusters. We denote the area of *PQRS* as A_{go_others}.

Lemma 1 *Given a delay dependence* $d(e) = (d_i, d_j)$, *we have* $A_{go\_others}(d) = f_j d_i$.

Proof As shown in the shaded area in Fig. 8, we have $A_{go\_others}(d)$ = the area of parallelogram $PQRS = f_j d_i$.

Summing up of these areas for every delay dependence $d(e)$ in an MDFG, we can obtain the total number of prefetch operations as:

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

From the definition of the number of prefetching operations, we know that it is proportional to the size of f_j. However, it dose not change with $f(h)$. We will try to have the least number of prefetching and write-back operations to hide memory latency. Therefore, we keep f_j fixed and find the applicable $f(h)$ to improve the schedule where memory latency can be hidden.

Theorem 2 *In a cluster, the number of write-back operations is equal to the number of prefetching operations.*

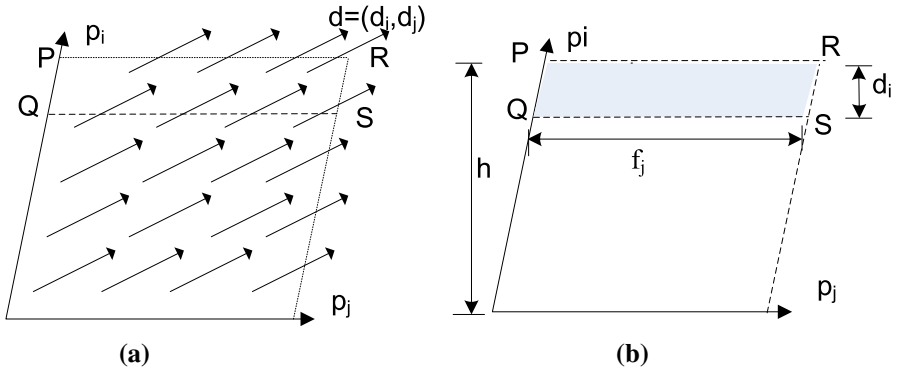

Fig. 8 The different kinds of delay dependence

Proof Since we only write back those data that we will ever need to fetch from remote memory, the number of write-back operations equals to the number of prefetching operations. □

Theorem 3 *If there exist crossing operations in a partition, the number of crossing operations for a processor is equal to the number of prefetching operations in the first processor.*

Proof The number of crossing operations and the number of prefetching operations are both shown as the shaded areas in Fig. 8a. This is because all the delay dependencies are uniform, and all the shaded areas are equal. We can see from Fig. 7b, that the area of each cross region is equal to the area of top region. And, the number of crossing operations for a processor is equal to the number of prefetching operation in the first processor.

When considering a cluster, as shown in Fig. 7b, the top region is located in the top partition of the cluster, and other partitions have a cross region. The memory units of processor $P1$ need to fetch the data in top region from the remote memory and need migrate the data in cross region to processor $P2$. We mainly concern the memory schedule of processor $P1$, which schedules the bottom partition of a cluster. This is because prefetching operations take longer time to complete than either write-back or crossing operations. Therefore, the processor which proceeds the bottom partition must satisfy Theorem 4.

Theorem 4 *Assume that $N_{core} \leq N_{mem}$. The following inequality is satisfied:*

$$\left\lceil \frac{N_{pre}}{N_{mem}} \right\rceil \times T_{pre} + \left\lceil \frac{N_{cro}}{N_{mem}} \right\rceil \times T_{cro} \leq L(h) \times f(h) \times f_j.$$

Proof In the memory part of the schedule, the length of the prefetch part for the bottom partition is $\lceil N_{pre}/N_{mem} \rceil \times T_{pre}$, and the length of the crossing part is $\lceil N_{cro}/N_{mem} \rceil \times T_{cro}$. Hence, the left-hand side represents the length of the memory part of the schedule for the bottom partition. In the processor part of the schedule, $L(h)$ represents the

schedule length of one iteration scheduled in processor $P(h)$, and $f(h) \times f_j$ shows the number of iterations to be executed by the processor. Thus, the right-hand side denotes the length of the processor part of the schedule for the bottom partition. To hide memory latency completely, the length of memory part of the schedule must be no less than that of processor part for any partitions. □

The top partition needs to write the data in top region back to remote memory. Thus, the processor which proceeds the top partition must satisfy Theorem 5.

Theorem 5 *Assume that $N_{core} \leq N_{mem}$. The following inequality is satisfied:*

$$\left\lceil \frac{N_{write}}{N_{mem}} \right\rceil \times T_{write} \leq L(h) \times f(h) \times f_j.$$

Proof The proof is similar to the Proof of Theorem 4. □

All partitions besides the top partition and bottom partition need cross the data in cross region to another processor. Thus, the processors must satisfy Theorem 6.

Theorem 6 *Assume that $N_{core} \leq N_{mem}$. The following inequality is satisfied:*

$$\left\lceil \frac{N_{cro}}{N_{mem}} \right\rceil \times T_{cro} \leq L(h) \times f(h) \times f_j.$$

Proof The proof is similar to the Proof of Theorem 4. □

Lemma 2 *When $f(h) = L_{max} \times f(max)/L(h)$, the schedule length in the memory part must be equal to or shorter than the schedule length in the processor part. In other words, $f(h) = L_{max} \times f(max)/L(h)$ satisfies Theorems 4–6, where L_{max} is the longest schedule length of iterations on processor part, and $f(max)$ is the partition size obtained from Theorem 4 based on L_{max}.*

Proof We put $f(h) = L_{max} \times f(max)/L(h)$ into the inequalities of Theorems 4–6. Their right-hand sides are all equal to $L_{max} \times f(max) \times f_j$. Since we obtain $f(max)$ from the inequality of Theorem 4, $f(h) = L_{max} \times f(max)/L(h)$ must satisfy Theorem 4. For Theorem 5, the left-hand side of the inequality equals to the length of crossing part of Theorem 4 and must be less than the length of the memory part of bottom partition of a cluster. Hence, $L_{max} \times f(max) \times f_j$ also satisfies Theorem 5. In Theorem 6, the left-hand side of the inequality is the length of the memory part of the top partition. Since the number of write-back operations equals to that of prefetching operations in a cluster, the length of memory part of the top partition must be less than $L_{max} \times f(max) \times f_j$. Therefore, by requiring the length of memory part of the bottom partition of a cluster to be less than the length of processor part, we guarantee that the length of memory schedule is not longer than the length of processor schedule on any processors.

The discussion on partition sizes has been finished. However, it is not enough for designing a real heterogeneous multiprocessor system. When designing a real system,

the designers want to know how much main memory is required to implement the proposed HMP schedule. In the following, we will concentrate on this problem.

The main memory should be large enough to hold all the data that are needed during the execution of partitions. For a processor, we classify the requirements of main memory into two categories, i.e., the first category used to store the data for the partition to be executed next, and the second category used to store prefetched, crossed, and written back data. The requirements of main memory must satisfy the following theorem.

Theorem 7 *The requirements of main memory men_{req} is calculated by the following equation:*

$$mem_{req} = 2(N_{pre} + N_{cro}) + \sum(d_j \times P(h)).$$

Proof In a partition, the first category corresponds to all the delay dependencies into the partition which will be executed next in the same processor. Similar to Lemma 1, for a delay dependence $d(e) = (d_i, d_j)$, we will store $d_j \times P(h)$ data in the main memory. Summing up all of these for every delay dependence $d(e)$ in an MDFG, we can obtain memory locations needed by the first category as $\sum(d_j \times P(h))$. The second category reserves memory space for prefetching, crossing, and write-back operations. These operations indicate the data pre-loaded or pre-stored in the memory before we execute each partition. We need to pre-load data for the current partition and store newly generated data for the partition which will be executed next on the same processor. Therefore, the size of the second category of memory is $2(N_{pre} + N_{cro})$. Finally, the memory size needed to execute a partition is $mem_{req} = 2(N_{pre} + N_{cro}) + \sum(d_j \times P(h))$.

5 Experiments

To evaluate the efficiency of the HMP algorithm, we experiment with our algorithm on a set of DSPstone benchmarks which are found in [42]. The MDFGs are extracted from the gcc compiler and then the MDFGs are fed into a custom simulator. We use both homogeneous and heterogeneous multiple multicore processor systems to perform these benchmarks. The configuration of the memory part for all experiments is shown in Table 1.

Table 1 Target system specification

Component	Description
Local memory	SPM (SRAM) access latency: 3.94 ns
Main memory	Read latency: 40 ns; write latency: 60 ns; cross latency: 150 ns
Remote memory	Access latency: 300 ns

5.1 Evaluation of HMP on Homogeneous Systems

To evaluate the efficiency of the HMP algorithm on a homogeneous system, we conduct two groups of experiments. In our empirical studies, we compare our HMP algorithm with the IRP algorithm. The IRP algorithm is employed to partition schedule tasks for hiding memory latency completely in a homogeneous system. To make fair comparisons, we implement the IRP and HMP algorithms with the same scheduling framework. Thus, the implementations of the two algorithms share identical data structures (e.g., a schedule queue and multiple local queues) and supporting modules (e.g., the task dispatcher). In doing so, we ensure that the performance disadvantages of IRP is not due to fundamental limitations of the implementations.

In the first group of the experiment, we focus on the scheduling problem for benchmarks running on one multicore processor. We assume that the multicore processor contains 4 cores, and the execution time for each task is 10 ns. The experimental results are shown in Fig. 9. From the figure, we can see that the schedule length of the HMP algorithm is shorter than that of the IRP algorithm. Therefore, the HMP scheduling algorithm has better performance compare with the IRP scheduling algorithm when evaluating the efficiency of the two algorithms on one multicore processor.

In the second group of the experiment, these DSPstone benchmarks are performed on multiple homogeneous multicore processors. We assume that there are 3 multicore processors, and each processor is equipped with 3 cores. The execution time of each processing core for each task is 10 ns. Figure 10 shows the experimental results. From the figure, we can see that the HMP scheduling algorithm also outperforms the IRP scheduling algorithm in schedule length when the benchmarks are executed on a homogeneous multiprocessor system.

Through the two groups of experimental results from Figs. 9 and 10, we find that the HMP algorithm is superior to the IRP algorithm when evaluating the efficiency of

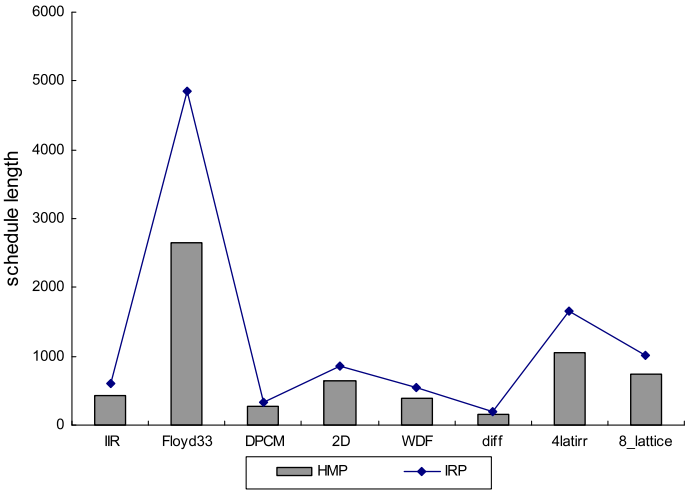

Fig. 9 Comparison results between IRP and HMP on one multicore processor

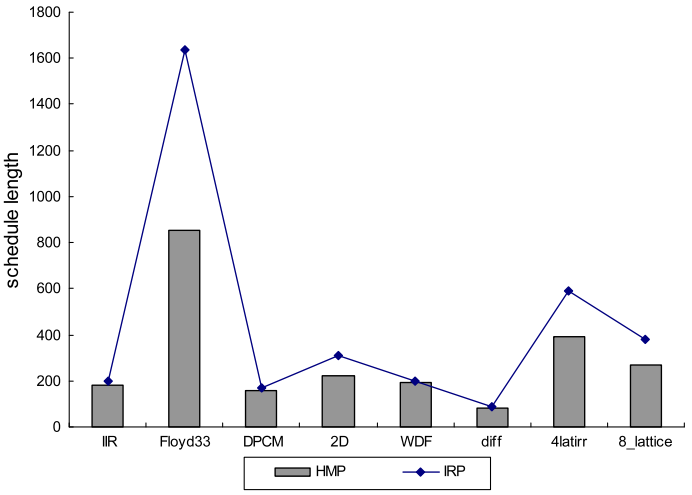

Fig. 10 Comparison results between IRP and HMP on multiple multicore processors

the two algorithms on homogeneous systems. This is because the IRP algorithm does not consider the load/store operations.

5.2 Evaluation of HMP on Heterogeneous Systems

In this subsection, we will present the experimental results of evaluating the HMP algorithm on heterogeneous multicore processor systems. Each processor in a heterogeneous multicore processor system consists of multiple processing cores and a special hardware called memory units inside each processor. A three-level memory hierarchy is adopted in our architecture model. Each processing core is equipped with a local memory called scratch pad memory (SPM), and associated with each processor is a main memory, and there is also a large multi-port remote memory. We simulate the cases when there are three multicore processors. The three processors have different configurations on processing cores and the same configuration on memory part. The system specification of the memory part used to evaluated the HMP algorithm is shown in Table 1. Three groups of experiments are conducted. In the set of experiments, we regard memory latency as the primary optimization objective. Meanwhile, the schedule length is also reduced.

We implemented the HMP algorithm as a stand-alone program which takes profile MDFG and the memory access time as input. In the set of experiments, the HMP algorithm is compared with the IRP algorithm and the List algorithm. In this subsection, the IRP algorithm has been adjusted, so that it is suitable to a heterogeneous multiprocessor system. To make fair comparisons, we implement all the three algorithms, i.e., IRP, HMP, and List algorithms, within the same scheduling framework. In doing so, we ensure that the performance disadvantage of the IRP and the List algorithms are not due to fundamental limitations of the implementations.

Table 2 The experimental results of the first group of experiments

Benchmark	Node	Partition size				HMP		IRP		List	
		f1	f2	f3	fj	L_{avg}	M_{req}	*len*	*ratio* (%)	*len*	*ratio* (%)
IIR	16	11	13	15	2	167.83	495	265.26	63.26	595.00	28.07
Floyd	144	4	5	7	1	1215.00	816	1850.00	65.67	4860.00	25.00
DPCM	16	7	8	10	4	104.00	957	302.00	34.43	600.00	17.33
WDF	12	6	7	8	1	157.14	96	265.32	59.27	550.00	28.57
2D	26	12	13	15	1	255.00	490	562.00	45.37	1125.00	22.67
8_*lattice*	42	8	9	10	1	299.26	807	492.18	60.80	1010.00	29.63
4latirr	52	8	9	12	1	456.55	954	709.25	64.37	2025.00	22.55
diff	10	15	16	20	1	60.19	180	87.75	68.59	325.00	18.52
Average ratio									57.72		24.04

Tables 2, 3 and 5 show our experimental results. The first column gives the benchmarks' names. The second column gives the node numbers of the input MDFGs. The partition generated by the HMP algorithm is shown in the third to sixth columns. The next two columns show the final schedule generated by the HMP algorithm. The column L_{avg} is the average schedule length for each iteration. The column M_{req} shows the requirements of main memory. In the set of experiments, the HMP algorithm is compared to the IRP and the List algorithms. The results are shown in the columns IRP and List, where the subcolumn *len* is the schedule length of each iteration and the subcolumn *ratio* compares the HMP schedule length with the IRP and List schedule length, that is, $ratio{=}L_{avg}/len$.

In the first group of experiments, we consider heterogeneous processor sizes, where benchmarks are running on a three-processor heterogeneous computing system. The three processors consist of three, four, and five processing cores, respectively. We assume that all processing cores have the same characteristics. The computation time for each task is 10 ns. Table 2 shows the first group of experimental results. As we can see, List scheduling rarely achieves the optimal schedule length. The List schedule length is longer than the HMP schedule length. This is because a List schedule is often dominated by a long memory part. Although the IRP algorithm is better than the List scheduling algorithm by generating a balanced schedule, it is not able to take full advantage of all the hardware resources available by exploring higher level of parallelism. In Table 2, the average ratio of the schedule length of the HMP scheduling algorithm to those of the IRP scheduling algorithm and the List scheduling algorithm are 57.72 and 24.04 %, respectively.

In the second group of experiments, we consider heterogeneous execution times when all processors have the same number of processing cores. We assume that each processor has four processing cores, and different processors have different execution times. In our experiments, the execution times of the processors P1, P2, and P3 for a computational task are 8, 10, and 12 ns, respectively. Table 3 shows the second group of experimental results. From the table, we can see the HMP schedule length is shorter than the IRP schedule length and the List schedule length for all benchmarks. This is

Table 3 The experimental results of the second group of experiments

Benchmark	Node	Partition size				HMP		IRP		List	
		f1	f2	f3	fj	L_{avg}	M_{req}	*len*	*ratio* (%)	*len*	*ratio* (%)
IIR	16	16	17	20	2	161.85	621	226.45	71.47	485.00	33.37
Floyd	144	5	6	7	1	1301.10	828	1908.00	68.19	4684.00	27.78
DPCM	16	8	9	11	4	95.14	1111	306.00	31.09	625.00	15.22
WDF	12	8	8	10	1	156.92	106	228.72	68.61	480.00	32.69
2D	26	13	13	14	1	267.15	424	558.00	47.87	1068.00	25.01
8_*lattice*	42	8	8	8	1	306.33	744	502.15	61.00	919.00	33.33
4latirr	52	16	17	19	1	476.00	1368	708.00	67.23	1906.00	24.97
diff	10	19	23	24	1	67.93	228	108.21	62.78	361.00	18.82
Average ratio									59.78		26.98

Table 4 The processor configurations in the third group of experiments

Processor	P1	P2	P3
Time (ns)	10	12	8
Core	3	4	4

Table 5 The experimental results of the third group of experiments

Benchmark	Node	Partition size				HMP		IRP		List	
		f1	f2	f3	fj	L_{avg}	M_{req}	*len*	*ratio* (%)	*len*	*ratio* (%)
IIR	16	14	17	20	2	163.85	603	239.24	68.48	526.00	31.15
Floyd	144	5	6	8	1	1278.10	834	1904.00	67.12	4860.00	26.30
DPCM	16	7	8	11	4	96.29	1100	302.00	31.88	575.00	16.75
WDF	12	8	9	11	1	157.15	124	245.52	64.01	460.00	34.16
2D	26	12	13	14	1	268.42	426	587.60	45.68	1025.00	26.19
8_*lattice*	42	8	9	9	1	310.77	786	510.72	60.84	948.00	32.78
4latirr	52	10	11	13	1	486.76	1044	734.84	66.24	2116.00	23.00
diff	10	12	23	23	1	64.71	232	100.91	64.13	345.00	18.76
Average ratio									58.55		26.13

because different processors having different execution times affect the processors' schedule lengths, but do not affect how the HMP algorithm performs memory latency hiding. In Table 3, the average ratio of the schedule length of the HMP scheduling algorithm to those of the IRP scheduling algorithm and the List scheduling algorithm are 59.78 and 26.98 %, respectively.

In the third group of experiments, we consider heterogeneous processor sizes and speeds. Table 4 shows the processors' configurations, where the row *time* shows the execution time of a task when it is executed on a processor, and the row *core* shows the core number of a processor. Table 5 shows the third group of experimental results.

Table 6 The experimental results on the effect of different memory latency

Benchmark	Node	Partition size				HMP		IRP		List	
		f1	f2	f3	fj	L_{avg}	M_{req}	*len*	*ratio* (%)	*len*	*ratio* (%)
IIR	16	15	18	21	2	165.28	918	289.00	57.17	1200	13.77
Floyd	144	8	7	14	1	1215.00	984	2168.48	56.03	6210	19.57
DPCM	16	16	17	18	4	101.96	1617	400.16	25.48	1200	8.50
WDF	12	16	19	23	1	151.72	140	291.71	52.01	620	34.47
2D	26	19	20	24	1	256.35	600	702.91	36.47	2250	11.39
8_*lattice*	42	12	13	16	1	295.61	986	569.25	51.93	1780	16.61
4latirr	52	20	22	31	1	453.42	1746	816.00	55.56	4050	21.20
diff	10	27	28	34	1	60.67	274	110.89	54.71	650	9.33
Average ratio									48.67		16.86

From the table, we can see the HMP schedule is superior to the IRP and the List schedules in schedule length for all benchmarks. The row *average ratio* shows that the HMP algorithm can reduce the schedule length by $1-58.55\,\% = 41.45\,\%$ and $1-26.13\,\% = 73.87\,\%$ compared with the IRP and List algorithms, respectively.

Through the three groups of experimental results from Tables 2, 3, and 5, we find that the HMP algorithm has better performance compared with the IRP and the List algorithms when evaluating the efficiency of the three algorithms on heterogeneous multiprocessor systems.

To see the effect of different memory latency, we conducted a set of experiments assuming that the processor part are the same as that of the third group of experiments, when the prefetch takes 600 ns, write back takes 600 ns, and cross takes 300 ns. The experimental results are shown in Table 6. We can see that the HMP algorithm still outperforms the List scheduling algorithm. The average ratio of the schedule length of the HMP algorithm to those of the IRP and the List algorithms are 48.67 and 16.86 %, respectively. Comparing Tables 5 and 6, we can see that the HMP algorithm tends to create a larger partition in order to compensate for this long latency, when the memory latency is increased, and the improvement over IRP and List scheduling becomes more obvious.

6 Conclusion and Future Work

In this paper, we present a novel loop partition algorithm, i.e., the HMP algorithm, for MDFG applications to hide memory latency on heterogeneous multicore processor systems. By fully exploiting the properties of schedule length and memory requirement, the algorithm can give a partition shape and size, so that the overall minimal schedule length can be obtained. For experimental studies, we employed two homogeneous multicore processor systems and three heterogeneous multicore processor systems to execute various applications. In the experiments conducted on both homogeneous and heterogeneous multiprocessor systems, the HMP algorithm

can effectively reduce the overall schedule length compared with the IRP and the List scheduling algorithms.

Acknowledgements This research was partially funded by the Key Program of National Natural Science Foundation of China (61133005, 61432005), the National Science Foundation of China (Grant Nos. 61070057, 90715029, 61370095, 61472124), and the National Science Foundation for Distinguished Young Scholars of Hunan (12JJ1011).

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

Human and Animal Rights This article does not contain any studies with human participants or animals performed by any of the authors.

Informed Consent Informed consent was obtained from all individual participants included in the study.

References

1. Bala, K., Kaashoek, M.F., Weihl, W.E.: Software prefetching and caching for translation lookaside buffers. In: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation, p. 18. USENIX Association (1994)
2. Beaumont, O., Boudet, V., Robert, Y., et al.: A realistic model and an efficient heuristic for scheduling with heterogeneous processors (2001)
3. Belviranli, M.E., Bhuyan, L.N., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim. (TACO) **9**(4), 57 (2013)
4. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: Dague: a generic distributed dag engine for high performance computing. Parallel Comput. **38**(1), 37–51 (2012)
5. Chen, J., Tao, X., Yang, Z., Peir, J.-K., Li, X., Lu, S.-L.: Guided region-based gpu scheduling: utilizing multi-thread parallelism to hide memory latency. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 441–451. IEEE (2013)
6. Chen, T., Zhang, T., Sura, Z., Tallada, M.G.: Prefetching irregular references for software cache on cell. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 155–164. ACM (2008)
7. Chen, T.-F., Baer, J.-L.: A performance study of software and hardware data prefetching schemes. In: Proceedings the 21st Annual International Symposium on Computer Architecture, 1994, pp. 223–232. IEEE (1994)
8. Chen, T.-F., Baer, J.-L.: Effective hardware-based data prefetching for high-performance processors. IEEE Trans. Comput. **44**(5), 609–623 (1995)
9. Chen, Y., Liao, H., Tsai, T.: On-line real-time task scheduling in heterogeneous multi-core system-on-a-chip. IEEE Trans. Parallel Distrib. Syst. **24**, 118–130 (2013)
10. Chu, M., Ravindran, R., Mahlke, S.: Data access partitioning for fine-grain parallelism on multicore architectures. In: MICRO 2007. 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007, pp. 369–380. IEEE (2007)
11. Dahlgren, F., Dubois, M., Stenstrom, P.: Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. **6**(7), 733–746 (1995)
12. Daoud, M.I., Kharma, N.: A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **68**(4), 399–409 (2008)
13. Eryigit, S., Bayhan, S., Tugcu, T.: Energy-efficient multi-channel cooperative sensing scheduling with heterogeneous channel conditions for cognitive radio networks. IEEE Trans. Veh. Technol. **62**, 2690–2699 (2013)
14. Ganusov, I., Burtscher, M.: Future execution: a hardware prefetching technique for chip multiprocessors. In: 14th International Conference on Parallel Architectures and Compilation Techniques, 2005. PACT 2005, pp. 350–360. IEEE (2005)

15. Hagras, T., Janeček, J.: A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Comput. **31**(7), 653–670 (2005)
16. Hoogerbrugge, J., Terechko, A.: A multithreaded multicore system for embedded media processing. In: Transactions on High-performance Embedded Architectures and Compilers III, pp. 154–173. Springer, Berlin (2011)
17. http://www.androidheadlines.com/2013/09/samsung-upgrades-exynos-5-to-true-octa-core-status-with-heterogeneous-multi-processing.html (2013)
18. Hu, J., Xue, C.J., Tseng, W.-C., Zhuge, Q., Sha, E.-M.: Minimizing write activities to non-volatile memory via scheduling and recomputation. In: 2010 IEEE 8th Symposium on Application Specific Processors (SASP), pp. 101–106. IEEE (2010)
19. Jeong, J., Kim, H., Hwang, J., Lee, J., Maeng, S.: Rigorous rental memory management for embedded systems. ACM Trans. Embed. Comput. Syst. (TECS) **12**(1s), 43 (2013)
20. Klaiber, A.C., Levy, H.M.: An architecture for software-controlled data prefetching. In: ACM SIGARCH Computer Architecture News, vol. 19, pp. 43–53. ACM (1991)
21. Lilja, D.J.: The impact of parallel loop scheduling strategies on prefetching in a shared memory multiprocessor. IEEE Trans. Parallel Distrib. Syst. **5**(6), 573–584 (1994)
22. Liu, G., Abdelrahman, T.: Computation–communication overlap on network-of-workstation multiprocessors. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1635–1642 (1998)
23. Luk, C.-K.: Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In: Proceedings. 28th Annual International Symposium on Computer Architecture, 2001, pp. 40–51. IEEE (2001)
24. Mowry, T., Gupta, A.: Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. J. Parallel Distrib. Comput. **12**(2), 87–106 (1991)
25. Nishiyama, H., Kikuchi, S.: Method for compiling loops containing prefetch instructions that replaces one or more actual prefetches with one virtual prefetch prior to loop scheduling and unrolling, Sept. 7 1999. US Patent 5,950,007
26. Orlando, S., Perego, R.: Exploiting partial replication in unbalanced parallel loop scheduling on multicomputer. Microprocess. Microprogram. **41**(8), 645–658 (1996)
27. Page, A.J., Naughton, T.J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In: Proceedings. 19th IEEE International, Parallel and Distributed Processing Symposium, 2005, p. 189a. IEEE (2005)
28. Poulsen, D.K., Yew, P.-C.: Data prefetching and data forwarding in shared memory multiprocessors. In: International Conference on Parallel Processing, 1994. ICPP 1994, vol. 2, pp. 280–280. IEEE (1994)
29. Qiu, M., Liu, M., Hu, F., Liu, S., Wang, L.: Energy aware loop scheduling for high performance multimodule memory. In: Sixth IFIP International Conference on Network and Parallel Computing, 2009. NPC'09, pp. 16–22. IEEE (2009)
30. Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. ACM SIGARCH Comput. Archit. News **37**(3), 24–33 (2009)
31. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005)
32. Shukla, S.B., Agrawal, D.P.: Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In: ACM SIGARCH Computer Architecture News, Vol. 19, pp. 222–231. ACM (1991)
33. Stone, J.E., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66 (2010)
34. Tang, X., Li, K., Liao, G., Li, R.: List scheduling with duplication for heterogeneous computing systems. J. Parallel Distrib. Comput. **70**(4), 323–329 (2010)
35. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)
36. Tosun, S.: Energy-and reliability-aware task scheduling onto heterogeneous mpsoc architectures. J. Supercomput. **62**(1), 265–289 (2012)
37. Wang, L., Siegel, H.J., Roychowdhury, V.P., Maciejewski, A.A.: Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. J. Parallel Distrib. Comput. **47**(1), 8–22 (1997)

38. Wolf, M.E., Lam, M.S.: A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans. Parallel Distrib. Syst. **2**(4), 452–471 (1991)
39. Xue, C.J., Hu, J., Shao, Z., Sha, E.: Iterational retiming with partitioning: loop scheduling with complete memory latency hiding. ACM Trans. Embed. Comput. Syst. (TECS) **9**(3), 22 (2010)
40. Zhong, C., Qu, Z.-Y., Yang, F., Yin, M.-X., Li, X.: Efficient and scalable thread-level parallel algorithms for sorting multisets on multi-core systems. J. Comput. **7**(1), 30–41 (2012)
41. Zhuang, X., Pande, S.: Power-efficient prefetching for embedded processors. ACM Trans. Embed. Comput. Syst. (TECS) **6**(1), 3 (2007)
42. Zivojnovic, V., Velarde, J.M., Schlager, C., Meyr, H.: Dspstone: a DSP-oriented benchmarking methodology. In: Proceedings of ICSPAT 94 (1994)
43. Zucker, D.F., Lee, R.B., Flynn, M.J.: Hardware and software cache prefetching techniques for mpeg benchmarks. IEEE Trans. Circuits Syst. Video Technol. **10**(5), 782–796 (2000)