

# Variable Partitioning and Scheduling for MPSoC with Virtually Shared Scratch Pad Memory

Lei Zhang · Meikang Qiu · Wei-Che Tseng ·  
Edwin H.-M. Sha

Received: 2 November 2008 / Revised: 14 March 2009 / Accepted: 16 March 2009 / Published online: 24 April 2009  
© 2009 Springer Science + Business Media, LLC. Manufactured in The United States

**Abstract** One of the most critical components that determine the success of an MPSoC based architecture is its on-chip memory. Scratch Pad Memory (SPM) is increasingly being applied to substitute cache as the on-chip memory of embedded MPSoCs due to its superior chip area, power consumption and timing predictability. SPM can be organized as a Virtually Shared SPM (VS-SPM) architecture that takes advantage of both shared and private SPM. However, making effective use of the VS-SPM architecture strongly depends on two inter-dependent problems: variable partitioning and task scheduling. In this paper, we decouple these two problems and solve them in phase-ordered manner. We propose two variable partitioning heuristics based on an initial schedule: High Access Frequency First

(HAFF) variable partitioning and Global View Prediction (GVP) variable partitioning. Then, we present a loop pipeline scheduling algorithm known as Rotation Scheduling with Variable Partitioning (RSVP) to improve overall throughput. Our experimental results obtained on MiBench show that the average performance improvements over IDAS (Integrated Data Assignment with Scheduling) are 23.74% for HAFF and 31.91% for GVP on four-core MPSoC. The average schedule length generated by RSVP is 25.96% shorter than that of list scheduling with optimal variable partition.

**Keywords** Variable partitioning · Scheduling · Scratch pad memory · MPSoC

---

This work is partially supported by NSF CCR-0309461, NSF IIS-0513669, HK CERG B-Q60B, NSFC 60728206, and China Scholarship Council[2007]3020.

---

L. Zhang (✉)  
School of Computer Science and Engineering,  
University of Electronic Science and Technology of China,  
Chengdu, Sichuan 610054, P. R. China  
e-mail: doczhanglei@hotmail.com

M. Qiu  
Department of Electrical Engineering,  
University of New Orleans, New Orleans, LA 70148, USA  
e-mail: mqi@uno.edu

W.-C. Tseng · E. H.-M. Sha  
Department of Computer Science,  
University of Texas at Dallas, Richardson, TX 75083, USA

W.-C. Tseng  
e-mail: wxt043000@utdallas.edu

E. H.-M. Sha  
e-mail: edsha@utdallas.edu

## 1 Introduction

Advances in VLSI circuits have made it possible to develop *Multiprocessor System-on-Chip* (MPSoC), which is widely used to solve a diverse spectrum of complex problems in embedded computing area, such as high-performance digital signal processing application and mobile communication system. In fact, many applications spend a significant portion of their execution time in accessing memory. The growing speed gap between powerful computational capability of MPSoC and low speed of memory components becomes a bottleneck for performance improvements. Memory system optimization is a critical issue for successfully executing parallel applications on MPSoC.

To close the processor-memory speed gap, the memory system of MPSoC is always organized as a two-level memory hierarchy: on-chip and off-chip memory.

On-chip memory can be directly accessed by processors with very low latency, whereas the access latency of off-chip memory is much higher. In modern embedded processors, Scratch Pad Memory (SPM) is increasingly being employed due to its inherent advantages in terms of chip area, energy-efficient and timing predictability compared to cache. SPM consists of an SRAM array and decoders, which can be easily integrated into the chip. The main difference between SPM and cache is that the SPM guarantees a single-cycle access time, whereas an access to the cache is subject to cache miss which may take thousands of cycles. Data storage onto the SPM is not automatically controlled by hardware. Therefore, a scratch pad memory has 34% smaller area and 40% lower power consumption than a cache memory of the same capacity [1]. Commercial embedded processors, such as Motorola MCore [2], Texas Instruments TMS370Cx [3], Motorola 68HC12 [4], etc., take SPM as their on-chip memory.

Since there is no hardware-controlled mechanism to transfer data between the SPM and off-chip memory, such transfers need to be explicitly managed by the software (compiler and/or application). Kandemir et al. [5] proposed a tiling-like transformation for data exchange between on-chip and off-chip memory. Xue et al. [6] introduced a data prefetching technique to hide the memory latency for accessing off-chip memory. Other previous works, such as [7–9] are also on the data transfer problem. In this paper, our target on-chip memory is organized as a *Virtually Shared SPM* (VS-SPM) architecture [10], where access to remote SPMs cost many more clock cycles than to local SPMs. We focus on data placement on the VS-SPM, and our goal is to maximize the locality of data references. So far, this issue has not been studied sufficiently.

For data-intensive applications, such as multimedia and DSP applications, the memory access operations account for approximately half of the cycle count [11], so optimizing the memory access is critical for performance improvement. For VS-SPM memory architectures, assigning variables to the appropriate SPMs will significantly reduce the overhead of remote SPM accesses. However, conventional task scheduling methods are computation-centric and ignore the effects of data layout. We strongly believe that jointly attacking both task scheduling and data partitioning is important in achieving an efficient solution for MPSoC with VS-SPM. However, each problem on its own is extremely complex, since variable partitioning decisions affect decisions on task scheduling and vice versa. To the best of our knowledge, there are no existing methods that can solve our problem directly. Some methods for solving similar problems are mainly Integer Linear Pro-

gramming (ILP) based techniques [12, 13]. The main deficiency of ILP based techniques is the very long run time for large scale problems. In particular, it is not fit for compile-time optimization. Thus, our approach is to decompose the complex problem into two simpler sub-problems that are solved in phase-ordered manner. First, we treat the VS-SPM as a large SPM with long access latency and use simple list scheduling to construct an initial schedule. Second, a heuristic variable partitioning algorithm is performed on the initial schedule to get a near-optimal variable partition. Then we use the variable partition to compact the schedule.

Our proposed variable partition algorithms are effective heuristics, in which the variable access frequencies, variable sizes and data dependencies are taken into account. The first heuristic is the *High Access Frequency First* (HAFF) variable partitioning algorithm. HAFF is a local greedy algorithm, which finds the current last finished processor and assigns the variables with higher access frequencies to the local SPM while meeting the constraints of the SPM capacities. When the latency of remote accesses is low (e.g., 5 clock cycles), HAFF performs well. The second heuristic is the *Global View Prediction* (GVP) variable partitioning algorithm. In GVP, the concept of a *Potential Critical Path* (PCP) is introduced to construct a novel weight matrix to measure the global benefit of a possible variable assignment. In each variable assignment step of GVP, we not only consider reducing the length of the current critical path, but also consider the future data requirements of potential critical paths. The experimental results on the selected benchmarks illustrate that the main advantages of our algorithms are:

1. The schedules generated by our algorithms are close to optimal. The experiments show that when the access latency is 10 clock cycles, the schedule length generated by list scheduling with our GVP algorithm is on average only 8.74% longer than the optimal schedule length which was generated by time-consuming exhausted search. When the latency reduced to 5 clock cycles, the result is very close to the optimal schedule.
2. The execution efficiency of our algorithms is high. Since the problem solved in this paper is NP-complete, achieving the optimal solution by any search based method takes a very long time. On the contrary, our proposed algorithms HAFF and GVP can achieve near optimal solution in at most one minute with our test cases. Thus, our algorithms can be practically used.

In order to explore coarse-grained parallelism for loop portions of an application on MPSoC architec-

ture, we propose a graph model *Data Flow Graph* (DFG) and a loop pipeline scheduling algorithm *Rotation Scheduling with Variable Partitioning* (RSVP). The DFG model can be used to explore parallelism across the loop body from different iterations. The task nodes and related variable information are extracted based on the profiling and static analysis of the application [14]. In RSVP algorithm, variable partitioning is integrated into the loop pipeline scheduling process. Rotation scheduling [15] is a mature approach to implementing loop pipelining [16]. Rotation implicitly uses retiming [17], which maintains the loop carried data dependencies of DFGs. Both HAFF and GVP can be easily integrated into RSVP when variable repartitioning is necessary. The experimental results show that the average schedule length generated by RSVP is 25.96% shorter than that of list scheduling with optimal variable partition.

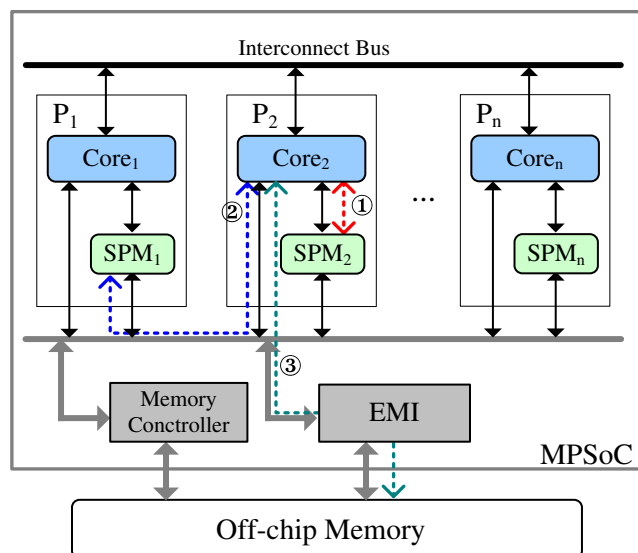
The remainder of this paper is organized as follows. Section 2 introduces basic concepts and related techniques. Section 3 formally describes the problem. Algorithms are proposed in Section 4. Experimental results and concluding remarks are provided in Sections 5 and 6, respectively.

## 2 Models and Basic Concepts

In this section, we introduce the basic concepts which will be used in the later sections. First, we describe the target architecture model *MPSoC with Virtually Shared Scratch Pad Memory* (VS-SPM). Second, we introduce the *Data Flow Graph* (DFG) model related to the variable partitioning problem. Third, we describe the retiming and rotation scheduling techniques.

### 2.1 Architecture Model

The organization of on-chip memory of our MPSoC architecture is depicted in Fig. 1. Each processor core is tightly coupled with a local SPM, and all the SPMs of individual processors make up a virtually shared SPM [10]. With respect to a specific processor, the SPMs of other processors are referred to as remote SPMs. For example, in Fig. 1, SPM<sub>1</sub> on processor P<sub>1</sub> is its local memory and SPM<sub>2</sub> ···, SPM<sub>n</sub> are regarded as remote SPMs. This on-chip memory architecture takes good characteristics of both private and shared memory [18]. Since memory already occupies up to 70% of the chip area [19], and chip area is limited, the capacity of on-chip memory cannot be easily increased. Only the data to be used soon or very critical data can be loaded into SPMs. Other data



**Figure 1** Architecture model MPSoC with VS-SPM.

is stored in off-chip memory. The off-chip memory is composed of large capacity DRAM. In our model, off-chip memory is a shared memory that can be accessed by processor cores via an External Memory Interface (EMI) [20]. The data exchange between on-chip and off-chip memory is under the control of a Memory Controller. A Memory Controller is a hardware component controlled by dedicated instructions responsible for prefetching and writing back data from/to the off-chip memory. This mechanism can help hide the very long communication latency of off-chip memory [6]. The individual processors are connected by an interconnect bus. In this work, we assume all the processor cores are homogenous, however, our proposed techniques can be extended to heterogenous MPSoCs.

**Definition 2.1** (MPSoC Architecture) An MPSoC  $M$  is a set of  $N$  processors  $\{P_1, P_2, \dots, P_n\}$ . Each processor  $P_i$  has a CPU core  $Core_i$  and a local SPM  $SPM_i$ . The size of  $SPM_i$  is denoted by  $Msize(SPM_i)$ .

Because of the speed gap between SRAM and DRAM technology and communication cost, we define three types of memory access patterns as depicted by the three dot lines in Fig. 1.

1. **Local Access.** If a processor accesses the data residing in its private SPM, the access speed is very fast. In this work, we assume local access only costs one clock cycle.
2. **Remote Access.** Due to the communication cost of the data bus, processor access to remote SPMs incurs a long access latency (e.g., 5 or 10 clock cycles).

3. **Off-chip Access.** A processor accesses data residing in the off-chip memory through the EMI. Because of the low performance of DRAM and the high communication cost, the access latency of off-chip memory is much longer than that of SPMs (e.g., 100 clock cycles).

The Off-chip Access latency can be hidden by data prefetching, so we only focus on variable layout on the VS-SPM in our work. For simplicity, we avoid the data coherency issue by making the assumption that a memory location can be mapped to at most one SPM. We also avoid the problem of bus access conflicts.

The CELL processor [21] can be regarded as a real implementation with similar architecture. Inside each CELL processor, there are 8 homogeneous cores each containing 256 KB of local memory called local store. Since local store is not a cache, data prefetching is explicitly controlled by instructions, not automatically by hardware. With this type of architecture, the CELL processor performs about 10 times better than a modern top of line CPU.

### 2.2 DFG Model

A task graph is a directed acyclic graph (DAG) that represents the computation blocks (tasks) of an application as nodes and communication between tasks as edges. However, a task graph cannot describe the loop carried dependencies and variable access of an application. In order to explore the pipelined loop schedule and solve the variable partitioning problem on MPSoC with VS-SPM, the *Data Flow Graph* (DFG) model is introduced in this section. In this paper, we will use DFG as the description of a given application.

**Definition 2.2** (Data Flow Graph) A DFG  $G = \langle V, E, X, d, t \rangle$  is a weighted directed graph, where  $V$  is a set of tasks,  $E \subseteq V \times V$  is the edge set that defines the

dependence relations between the nodes in  $V$ ,  $X(v)$  is a set of variables accessed by task  $v$ ,  $d(e)$  is non-negative integer which represents the number of delays for an edge  $e$ , and  $t(v)$  is a vector composed of an upper bound  $t_{up}(v)$  and a lower bound  $t_{low}(v)$  of execution latency of  $v$ .

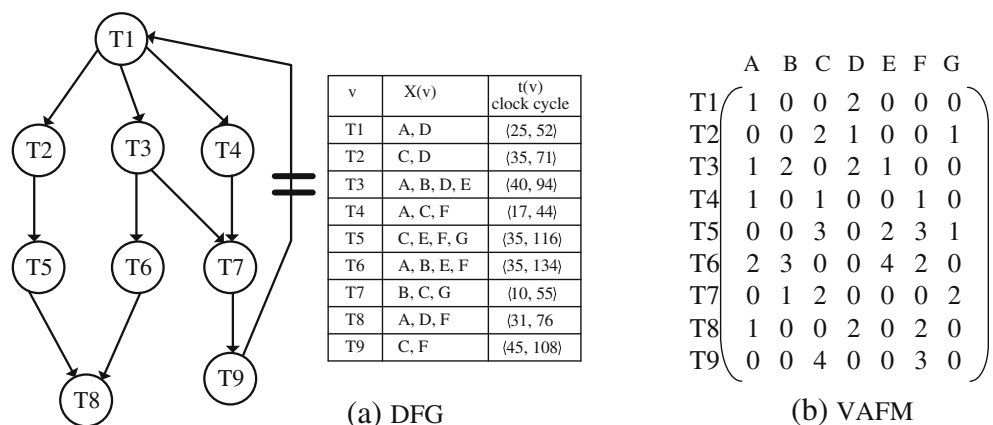
In our DFG model, each node  $v$  denotes a coarse-grain task of an application, which can be a function or a block of code. When compared to the coarsely grained task nodes, communication costs are very low. Due to the high *Computation-to-Communication Ratio* (CCR), the communication overhead can be neglected in scheduling. A task consists of computation operations and memory operations (load and store). For our target MPSoC, the execution latency of computation operations are identical, and the execution latency of memory operations depend on the data layout. In other words, the execution latency of a task cannot be an exact value until data location is determined. We can only determine the execution latency bounds of a task in the DFG generation phase. The  $t_{up}(v)$  and  $t_{low}(v)$  are defined in Eqs. 1 and 2

$$t_{up}(v) = comp_v + mem_v * R \tag{1}$$

$$t_{low}(v) = comp_v + mem_v \tag{2}$$

where  $comp_v$  is the number of computation operations of task  $v$ ,  $mem_v$  is the number of memory operations,  $R$  is the remote access latency. We assume each computation operation costs one clock cycle. Note that in this paper we only focus on the variable placement of on-chip memory, so when all the memory operations are *Remote Access*, the corresponding latency is regarded as the upper bound task execution latency. All of these parameters can be obtained by static analysis and application profiling. The detailed DFG generation process will be introduced in Section 5. An example

**Figure 2** Example of DFG (a) and VAFM (b).



DFG is illustrated in Fig. 2a, we assume the remote access latency is 10 clock cycle.

Loop carried dependencies can be represented in DFGs with cycle paths. An iteration is the execution of each node in  $V$  exactly once. Interiteration dependencies are represented by edges with delays. The edge  $e(u \rightarrow v)$  with delay count  $d(e) > 0$  means that task  $v$  at the  $j^{\text{th}}$  iteration depends on the data produced by task  $u$  at the  $(j - d(e(u \rightarrow v)))^{\text{th}}$  iteration. For example, in Fig. 2, the edge  $e(T9 \rightarrow T1)$  denotes that  $T1$  depend on the result of  $T9$  in the last two iterations. The dependencies within the same iteration are represented by edges with zero delay. The delay count of a path is the summation of delays of all the edges along the path. Note that if all the edges with nonzero are delays removed from the DFG, the subgraph must be a DAG. A static schedule obeys the precedence relations defined by the DAG part of the DFG. For a cyclic data flow graph, the delay count of any cycle needs to be a positive integer; otherwise, the corresponding DFG is illegal.

Variable access frequencies are critical for the variable partitioning problem. According to DFG and application profiling, we construct a matrix to reflect access frequency of each variable by each task.

**Definition 2.3** (Variable Access Frequency Matrix (VAFM)). For a given DFG  $G = \langle V, E, X, d, t \rangle$ , a VAFM  $F$  is one for which  $f_{i,j} \rightarrow \mathbf{Z}$ , where  $i \in [1, |V|]$  and  $j \in [1, |X|]$ .  $f_{i,j}$  represents the times of the  $j^{\text{th}}$  variable accessed by the  $i^{\text{th}}$  task.

The VAFM of the DFG in Fig. 2a is shown as Fig. 2b. Each row reflects the variable accesses of one task. From the VAFM, we can get a global view of relations between tasks and variables.

### 2.3 Retiming and Rotation Scheduling

The techniques described in this section will be used in our loop pipeline scheduling algorithm RSVP. Retiming [17] has been effectively used to obtain the minimum cycle period for a DFG by rearranging the delays. It is defined as a function  $r(v)$  from  $V$  to an integer. The value  $r(v)$  is the number of delays drawn from each of the incoming edges of node  $v$  and pushed to each of the outgoing edges of  $v$ . Given a DFG  $G = \langle V, E, X, d, t \rangle$ , a legal retiming function  $r$  with cycle period  $c$  must satisfy the following conditions:

1. For every edge  $e(u \rightarrow v) \in E$ ,  $d_r(e) = d(e) + r(u) - r(v) \geq 0$ . So  $r(v) - r(u) \leq d(e)$  for each edge  $e(u \rightarrow v) \in E$ .

2. For each path  $u \rightsquigarrow v$ ,  $d_r(p) = d(p) + r(u) - r(v) \geq 1$  if  $t(p) > c$ . So  $r(v) - r(u) \leq d(p) - 1$  for each path  $u \rightsquigarrow v$  if  $t(p) > c$

where:  $d_r(e)$  is the delay of edge  $e$  after retiming;  $d(p)$  is the number of delays of path  $p$ ;  $t(p)$  is the computation time of path  $p$ . In Fig. 3a,  $r(T3) = 1$  is a legal retiming, since the delay of incoming edges of  $T3$  are no less than 1. If retiming function is  $r(T3) = 2$ ,  $d_r(e(T2 \leftarrow T3)) = 1 - 2 = -1 < 0$ , that means  $T3$  in this iteration depends on the result of  $T2$  of a future iteration, so  $r(T3) = 2$  is an illegal retiming. After retiming, the length of the critical path would be reduced and a more compact iteration period will be obtained. For instance, the DFG in the Fig. 2a, when  $r(T1) = 1$  and  $r(Ti) = 0, \forall i = [2 \dots 9]$ , the retimed graph  $G_r$  shown as Fig. 3b. The critical path of DAG of  $G_r$  is  $T3 \rightarrow T6 \rightarrow T8$ , which is shorter than the original critical path  $T1 \rightarrow T3 \rightarrow T6 \rightarrow T8$ .

Rotation Scheduling presented in [15] is a loop scheduling technique used to optimize loop scheduling with resource constraints. It transforms a schedule to a more compact schedule iteratively. In most cases, the node level minimum schedule length can be obtained in polynomial time by rotation scheduling. In each step of the 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 removing one delay from each of incoming edges of the node and adding one delay to each of

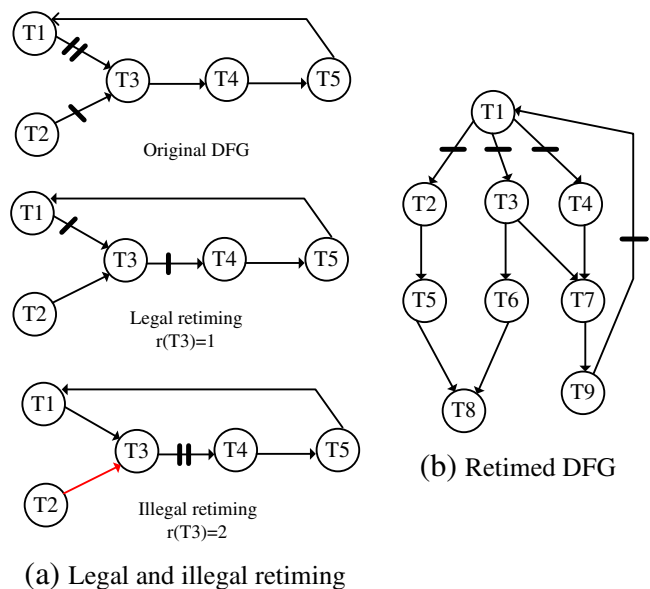


Figure 3 Retiming technology (a, b).

its outgoing edges in the DFG. The details of rotation scheduling can be found in [15].

### 3 Problem Statement

Task scheduling and variable partitioning are interdependent problems. The following motivational example demonstrates the interactions of these two problems. For simplicity, we assume that there are only two parallel executed tasks. The VAFM and the number of computation operations of each task are shown in Fig. 4a. There are two different variable partition schemas as shown in Fig. 4b and c. The dotted lines denote remote accesses (latency is 10 clock cycles) and the solid lines denote local accesses. Comparing these two variable partition, the execution latencies of the two tasks decrease by 49.5% and 38.1% respectively by the schema2. The schedule length of schema2 is only 61.9% compared to schema1.

**Variable Partitioning Problem** For a given DFG  $G = \langle V, E, X, d, t \rangle$  and target MPSoC  $M$ , variable assignment is a function  $\eta: X \rightarrow M$ , that maps a variable  $x_i$  onto a processor  $P_j$ . Then, the Variable Partition (VP) is defined as Eq. 3

$$VP = \{ \langle x_i, P_j \rangle : x_i \in X, P_j \in M, \eta(x_i) = P_j \} \tag{3}$$

A legal partition should satisfy the constraint:

$$\sum_{\forall x_i \in \{x_i : \eta(x_i) = P_j\}} \text{Size}(x_i) \leq \text{Msize}(\text{SPM}_j) \tag{4}$$

where  $\text{Size}(x_i)$  is the size of variable  $x_i$ . For example, in Fig. 4b, we assume that the capacity constraint (4) is satisfied,  $VP = \{ \langle A, P_1 \rangle, \langle B, P_2 \rangle, \langle C, P_1 \rangle, \langle D, P_2 \rangle \}$  is a

legal partition of variable set  $\{A, B, C, D\}$ . To avoid the complexity of data consistency, one variable can only be assigned to one SPM.

**Task Scheduling Problem** For a given application specified as DFG  $G = \langle V, E, X, d, t \rangle$  and a target MPSoC  $M$ , task scheduling maps the tasks to processors and suggests a start time for each task while maintaining the dependencies defined by  $E$ . We define a function as  $p(v_i) = P_j$ , where  $v_i \in V$  and  $P_j \in M$ , which represents that the  $i^{\text{th}}$  task is mapped onto the  $j^{\text{th}}$  processor. If a task  $v_i$  is scheduled on processor  $P_j$ ,  $ST(v_i, P_j)$  and  $FT(v_i, P_j)$  denote the start time and finish time of  $v_i$  on processor  $P_j$ , respectively. It should be noted that  $FT(v_i, P_j) = ST(v_i, P_j) + \text{Latency}(v_i, P_j)$ . Since execution latency of a task depends on the variable partition, we define task latency in Eq. 5.

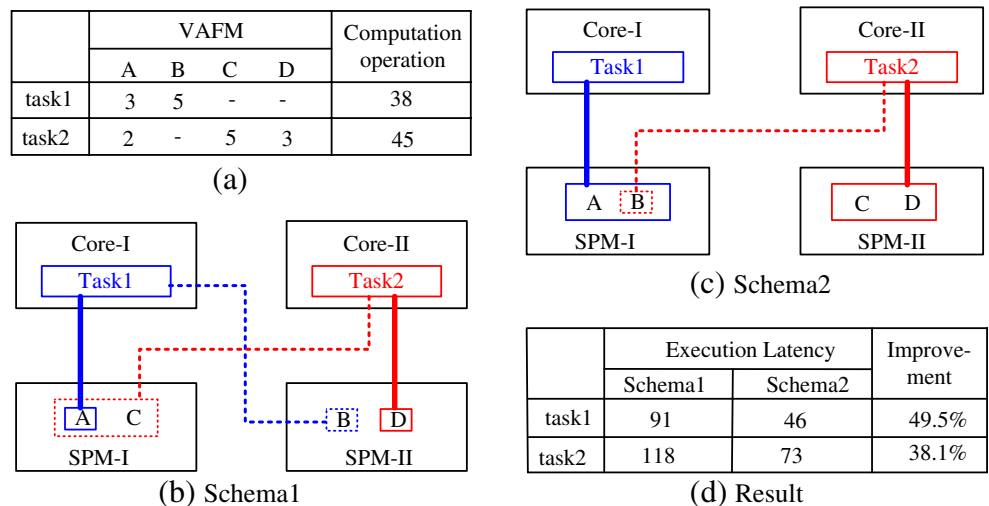
$$\text{Latency}(v_i, P_j) = \text{comp}_{v_i} + l_{\text{mem}_{v_i}} + r_{\text{mem}_{v_i}} * R \tag{5}$$

where  $\text{comp}_{v_i}$  is the number of computation operations in task  $v_i$ ,  $l_{\text{mem}_{v_i}}$  is the number of local access memory operation,  $r_{\text{mem}_{v_i}}$  is the number of remote access memory operation,  $R$  is the remote access latency. After all tasks have been scheduled, the schedule length  $SL$  is defined as Eq. 6.

$$SL = \max_{\forall v_i \in V, \forall P_j \in M} \{ FT(v_i, P_j) \} \tag{6}$$

**Goal** The goal of our algorithm is to find an appropriate variable partition and a corresponding task schedule to minimize  $SL$ . Scheduling problems with time and resource constraints are known to be NP-complete. We are going to solve this kind of scheduling problem with variable partitioning. Therefore, our problem is also NP-complete. The proposed heuristic algorithms

**Figure 4** Interaction between variable partitioning and task scheduling (a-d).



can approximate the optimal schedule in polynomial time. They are more efficient than ILP based solutions for large-scale problems (the number of processors, variables and tasks are large).

### 4 Algorithms

Solving variable partitioning and task scheduling problems simultaneously is extremely hard. We decouple these problems and solve them in a phase-ordered manner. In the first phase, for a given DFG, we generate an initial schedule  $S_{init}$  based on upper bound execution latencies of tasks. Any kind of heuristic algorithm can be used in the initial scheduling. For simplicity, we use list scheduling taking the number of descendants as the weight of a node. For example, considering the DFG in Fig. 5a, the  $S_{init}$  generated by list scheduling shown in Fig. 5b. The second phase, under the guide of the  $S_{init}$  and VFAM, the proposed variable partitioning algorithm will assign variables to appropriate SPMs to compact the  $S_{init}$ .

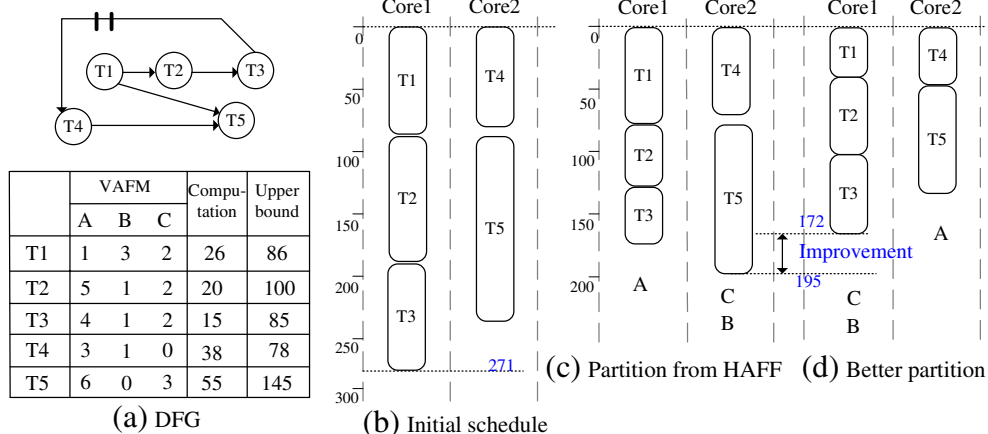
#### 4.1 High Access Frequency First (HAFF) Variable Partition

The HAFF algorithm is shown in Algorithm 4.1. Based on the  $S_{init}$  and VAFM, we define a matrix (Core-Variable Matrix) to represent the access frequencies of each variable accessed by different processor cores.

**Definition 4.1** (Core-Variable Matrix (CVM)). For a given initial schedule  $S_{init}$ , VAFM and target MPSoC M, CVM is an  $|M| \times |X|$  matrix as follows:

$$CVM[i, j] = \sum_{\forall v_k \in \{v_k : p(v_k) = P_i\}} VAFM[k, j]$$

**Figure 5** HAFF VS. GVP (a–d).



where  $v_k$  is a task node,  $|M|$  is the number of processor cores and  $|X|$  is the number of variables.  $CVM[i, j]$  represents the times of the  $j^{th}$  variable accessed by the tasks on the  $i^{th}$  processor core.

**Step 1.** In lines 1-2, we initialize T and construct the CVM. Note that by assigning variables to a processor  $P_\alpha \in T$  we can compact the schedule.

**Step 2.** We find the target processor  $P_\alpha$  which is to be assigned a variable (line 4). In the variable assignment process, the processor in T with the longest finish time is to be the target processor. After a variable is assigned, the finish times of processors may change. So, we will choose the target processor in each round of variable assignment. For example, in Fig. 5(b),  $P_1$  is the target processor in first round of variable assignment. After variable “A” have been assigned to  $SPM_1$ ,  $P_2$  becomes to the target processor. Since the SPMs capacities constraint, after several rounds of variable assignment, we can not find the  $P_\alpha$  from T. We will assign the left variables only considering the SPMs capacities (line 15).

**Step 3.** After the target processor  $P_\alpha$  is determined, we will choose a variable which has higher access frequencies by  $P_\alpha$  and meet the capacity constraint as the target variable  $var_\beta$  (line 6). For our example as Fig. 5, in the first round variable assignment  $P_1$  is the target processor, variable “A” is the target variable, because it has the highest access frequencies by  $P_1$ . If we can not find the target variable  $var_\beta$  for  $P_\alpha$ , then the  $P_\alpha$  will not be the target processor in future rounds of variable assignment (line 12).

**Algorithm 4.1** High frequency first variable partition**Input:** (1)Initial schedule:  $S_{init}$ , (2)Target machine:  $M$ , (3)DFG:  $G$ , (4)VAFM**Output:** (1)Variable partition:  $VP$ , (2)Compact schedule:  $S_{comp}$ 

```

1:  $T \leftarrow M$ ;
2:  $CVM \leftarrow CVM\_Generate(VAFM, S_{init})$ ;
3: while (!Partition_finished()) do
4:    $P_\alpha \leftarrow$  find a target processor from  $T$  by descending finish time order;
5:   if ( $P_\alpha$ ) then
6:      $var_\beta \leftarrow$  find a target variable by descending frequencies order, and the  $var_\beta$  should satisfy
       constraints: 1)  $CVM[P_\alpha, var_\beta] > 0$  and 2)  $FreeSpace(SPM_\alpha) > Size(var_\beta)$ ;
7:     if ( $var_\beta$ ) then
8:        $VP \leftarrow VP \cup \{(var, P)\}$ ;
9:        $FreeSpace(SPM_\alpha) \leftarrow FreeSpace(SPM_\alpha) - Size(var_\beta)$ ;
10:       $S_{comp} \leftarrow Schedule\_Compact(S_{init}, G, VP)$ ;
11:     else
12:        $T \leftarrow T - P_\alpha$ ;
13:     end if
14:   else
15:     Assign the left unassigned variables to make full use of SPMs;
16:   end if
17: end while
18: Return  $VP$  and  $S_{comp}$ ;

```

**Step 4.** If a variable assignment  $(var_\beta, P_\alpha)$  is found in step 3, we will add it to the variable partition  $VP$  and mark the  $var_\beta$  as an assigned variable. We also update the free space of  $SPM_\alpha$ . Then, we call “Schedule\_Compact” to generate a more compact schedule. The “Schedule\_Compact” process compact the latency of related tasks based on the current  $VP$  to generate a shorter schedule. Note that “Schedule\_Compact” keep the relative start order in  $S_{init}$  while maintaining the dependencies defined in DFG. The pseudo-code of this step is shown in lines 8–10.

The “Partition\_finished( )” is determination condition, when all the variables in VAFM are assigned or the left space of SPMs are smaller than any unassigned variables. In Algorithm 4.1, we sort the processors by finish time and the variables by access frequencies to determine the  $P_\alpha$  and  $var_\beta$ . The time complexity of the two sorts are  $\Theta(|M|\log|M|)$  and  $\Theta(|X|\log|X|)$  respectively, where  $M$  is number of processors and  $|X|$  is number of variables. Thus, the overall time complexity of HAFF algorithm is  $\Theta(|X||M|\log|M| + |X|^2\log|X|)$ .

## 4.2 Global View Prediction (GVP) Variable Partition

In the HAFF algorithm, the processors with longer finish times have higher priorities to be assigned a variable with a higher access frequency. According to our experimental results, HAFF can often generate a near-optimal schedule. However, sometimes to compact the overall schedule length the  $var_\beta$  (in Algorithm 4.1) should not be assigned to the  $P_\alpha$ . The example in Fig. 5 illustrates this limitation of HAFF. For the DFG and  $S_{init}$  in Fig. 5a and b, we assume that remote access latency is 10 clock cycle. The variable partition generated by HAFF is:  $VP_1 = \{(A, P1), (B, P2), (C, P2)\}$ . Base on  $VP_1$ , the schedule length has been compact to 195 control steps as shown in Fig. 5c. It is 71.9% of the initial schedule, which shows that VS-SPM architecture is much better than shared SPM architecture. However, we can obtain a better schedule shown in Fig. 5d from another variable partition  $VP' = \{(A, P2), (B, P1), (C, P1)\}$ . The schedule length based on this variable partition is only 172 control steps, 11.9% shorter than the schedule based on  $VP$ .

From the above example, we know that the main limitation of HAFF is caused by its local greedy strategy. Future variable requirements are not taken into account when assigning each variable. In this section,

we propose a novel heuristic algorithm called Global View Prediction (GVP) variable partition shown in Algorithm 4.2. The main difference between GVP and HAFF is that GVP has the capability of predicting the variable requirements of potential target processors for global optimization.

**Step 1.** We construct a new directed graph  $DAG_{vp} = (V', E')$  based on the given DFG and its initial schedule  $S_{init}$ , where  $V'$  consists of node set of DFG  $V$  and dummy head/tail nodes,  $E'$  represents the execution order of tasks in  $S_{init}$ . For example, the Fig. 6a is the  $DAG_{vp}$  generated from DFG and  $S_{init}$  in Fig. 5a and b.  $e(T1 \rightarrow T5)$  reflects the precedence of tasks on different cores, T5 cannot start until T1 finished. The dummy nodes facilitate finding the critical path and the pathes through a specific node.

**Step 2.** We find the Current Critical Path (CCP), Potential Critical Path (PCP) of  $DAG_{vp}$  and variables access matrix ( $CVM_{ccp}$  and  $CVM_{pcp}$ ) related to these paths (line 4-6). From Eq. 5 we know that the length of tasks are uncertain until the variable partition is determined. So, we find the CCP and PCP while keeping in mind that some variables have already been assigned, which is called the current variable partition.

**Definition 4.2** (CCP and PCP) For a given  $DAG_{vp}$  and current variable partition VP, Current Critical Path CCP is the longest path from head to tail. A Potential Critical Path PCP from head to tail satisfy the condition:  $Length(PCP) > Length(CP_{lb})$ , where  $Length(PCP)$  is the length of PCP,  $Length(CP_{lb})$  is the length of critical path when all task execution latencies are  $t_{low}$ . PS denotes the set of all PCPs.

---

**Algorithm 4.2** Global view prediction variable partition

---

**Input:** (1)Initial schedule:  $S_{init}$ , (2)Target machine:  $M$ , (3) DFG:  $G$  (4)VAFM

**Output:** (1)variable partition: VP, (2)compact schedule:  $S_{comp}$

```

1:  $DAG_{vp} \leftarrow Generate\_DAG_{vp}(S_{init}, G)$ ;
2: Initialize  $\lambda_1$  and  $\lambda_2$ ;
3: while ( !Partition_finished() ) do
4:    $CCP \leftarrow Find\_CCP(DAG_{vp})$ ;
5:    $PS \leftarrow Find\_PCP(DAG_{vp})$ ;
6:    $(CVM_{ccp}, CVM_{pcp}) \leftarrow (CCP, PS, VAFM)$ ;
7:    $CP\_gain[i, j] \leftarrow SL(S_{init}) - Schedule\_Compact(S_{init}, VP \cup \{\langle var_i, P_j \rangle\})$ ;
8:    $Global\_gain[i, j] \leftarrow \sum_{p(task_k)=P_j} VAFM(var_i, P_j) * PCP\_through(task_k)$ ;
9:    $Weight[i, j] \leftarrow \lambda_1 * CP\_gain[i, j] + \lambda_2 * Global\_gain[i, j]$ ;
10:  if (find a legal assignment based on Weight) then
11:     $VP \leftarrow VP \cup \{\langle var, P \rangle\}$ ;
12:  else
13:     $IBN\_gain \leftarrow generate\ a\ IBN\_gain$ ;
14:    if (find a legal assignment with higher value in IBN_gain) then
15:       $VP \leftarrow VP \cup \{\langle var, P \rangle\}$ ;
16:    else
17:      find a legal variable assignment and try to make full use of SPMs;
18:       $VP \leftarrow VP \cup \{\langle var, P \rangle\}$ ;
19:    end if
20:  end if
21:   $FreeSpace(SPM_p) \leftarrow FreeSpace(SPM_p) - Size(var)$ ;
22:   $S_{comp} \leftarrow Schedule\_Compact(S, G, VP)$ ;
23:   $\lambda_1 \leftarrow \lambda_1 + \delta$ ;
24:   $\lambda_2 \leftarrow 1 - \lambda_1$ ;
25: end while
26: Return VP and  $S_{comp}$  ;

```

---

For example, the CCP of DAG<sub>vp</sub> in Fig. 6 is (head → T1 → T2 → T3 → tail). Usually, the number of potential critical path is larger than one. We will consider all the paths in PS. One variable is assigned to a SPM in each round of variable assignment. The original critical path may be compacted after some rounds, then one or more PCPs become the CPP. For example, in Fig. 6b the “PathII” (head → T1 → T5 →

tail) and “PathIII” (head → T1 → T4 → tail) are the potential critical paths. If variable “A” is assigned to SPM<sub>1</sub> then “PathII” will become the CCP.

**Definition 4.3** (CVM<sub>ccp</sub> and CVM<sub>pcp</sub>) For a given DAG<sub>vp</sub>, CCP, and PS, CVM<sub>ccp</sub> and CVM<sub>pcp</sub> are two |M| × |X| matrixes shown as Eqs. 7 and 8

$$CVM_{ccp}[i, j] = \begin{cases} 1, & \text{if } \forall \text{task}_k \in \text{CCP}, p(\text{task}_k) = P_i \text{ and } VAFM[\text{task}_k, x_j] > 0 \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$CVM_{pcp}[i, j] = \begin{cases} 1, & \text{if } \forall \text{task}_k \in \text{PS}, p(\text{task}_k) = P_i \text{ and } VAFM[\text{task}_k, x_j] > 0 \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

The CVM<sub>ccp</sub>[i, j] = 1 represents that the variable assignment (var<sub>j</sub>, P<sub>i</sub>) will improve the schedule length. Similarly, CVM<sub>pcp</sub>[i, j] = 1 represents that it is possible to improve the schedule length by the variable assignment (var<sub>j</sub>, P<sub>i</sub>). For our example in Fig. 5, in the first round of variable assignment, the CVM<sub>pcp</sub> is a 2 × 3 one matrix and

$$CVM_{ccp} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

**Step 3.** A matrix Weight is constructed to measure the schedule length improvements from a variable assignment (var<sub>i</sub>, P<sub>j</sub>). The pseudo-code of this step is shown in line 7-9. Weight is computed from two |M| × |X| matrixes CP<sub>gain</sub> and Global<sub>gain</sub>.

**Definition 4.4** (CP<sub>gain</sub>) Given a DAG<sub>vp</sub> and a VAFM, a CP<sub>gain</sub> matrix is one for which

$$CP_{gain}[i, j] = \begin{cases} SL_{init} - SL_{i,j}, & \text{if } CVM_{ccp}[i, j] = 1 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

where SL<sub>init</sub> is the length of initial schedule and SL<sub>i,j</sub> is the length of initial schedule compacted by VP ∪ {(var, P)}.

CP<sub>gain</sub> matrix reflects the improvements of the schedule length after possible variable assignments are applied. For example, in Fig. 5a, the CP<sub>gain</sub> of first round variable assignment is

$$CP_{gain} = \begin{pmatrix} 49 & 48 & 48 \\ 0 & 0 & 0 \end{pmatrix}$$

CP<sub>gain</sub>[1][1]=49 means when variable “A” is assigned to SPM<sub>1</sub>, the schedule length will be compacted by 49 control steps. If we only use CP<sub>gain</sub> as the weight to determine a variable assignment, it is still a local greedy strategy. So, we will consider all the potential critical pathes of DAG<sub>vp</sub> and get a Global<sub>gain</sub> weight matrix.

**Definition 4.5** (Global<sub>gain</sub> Matrix) Given a DAG<sub>vp</sub> and related VAFM, a Global<sub>gain</sub> matrix is one for which

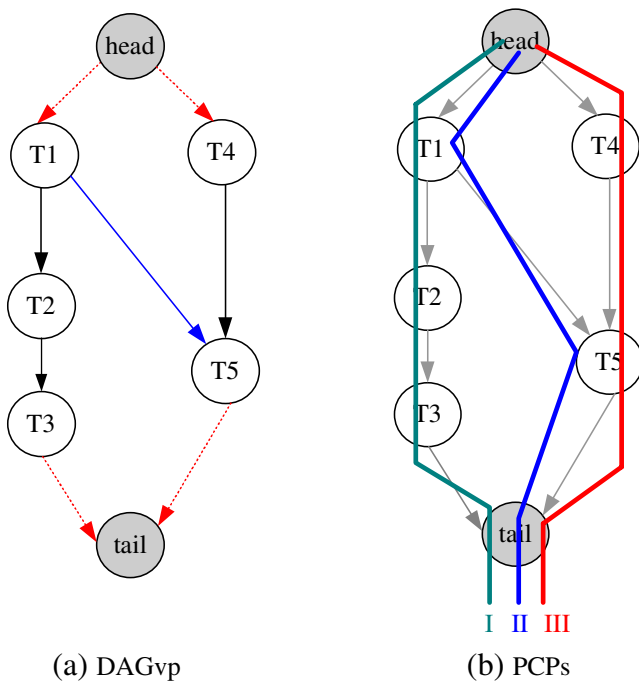
$$Global_{gain}[i, j] = \begin{cases} \sum_{p(\text{task}_k)=P_i} VAFM(\text{task}_k, x_j), & \text{if } CVM_{pcp}[i, j]=1 \\ * PCP_{through}(\text{task}_k), & \\ 0, & \text{otherwise} \end{cases}$$

where PCP<sub>through</sub>(task<sub>k</sub>) represents the number of potential critical paths that pass through the node task<sub>k</sub>.

For example, in Fig. 6b, PCP<sub>through</sub>(T1) = PCP<sub>through</sub>(T5)=2, and PCP<sub>through</sub>(i) = 1, ∀i ∈ {T2, T3, T4}. The Global<sub>gain</sub> of the first variable assignment round is:

$$Global_{gain} = \begin{pmatrix} 11 & 8 & 8 \\ 15 & 1 & 6 \end{pmatrix}$$

Global<sub>gain</sub> is global view measurement of this round variable assignment, which reflect the potential improvements from this assignment. To reflect the tradeoff between CP<sub>gain</sub> and Global<sub>gain</sub>, we use a weighted sum formula to compute the Weight



**Figure 6** DAG<sub>vp</sub> and potential critical path (a, b).

matrix. The average weight of a variable assignment Weight[i, j] is defined as

$$\text{Weight}[i, j] = \lambda_1 * \text{CP\_gain}[i, j] + \lambda_2 * \text{Global\_gain}[i, j] \tag{10}$$

where  $\lambda_1, \lambda_2$  are two coefficients that represent the weight of CP\_gain and Global\_gain. So, the Weight provides a complete measurement of benefit obtained from a candidate variable assignment. In the beginning of variable partition, we should pay more attention to global benefit of a variable assignment, such as  $\lambda_1 = 0.1, \lambda_2 = 0.9$ . After some variables have been assigned, the number of possible variable assignments decrease rapidly, so the Global\_gain will be less important. The two coefficients are defined as follow:

$$\lambda_1[i] = \lambda_1[i - 1] + \delta, \lambda_2[i] = 1 - \lambda_1[i],$$

where  $\delta = \frac{1}{2^{*|X|}}$ . Note, in order to ensure the elements of CP\_gain and Global\_gain are in the same range, each member of the CP\_gain and Global\_gain matrices is normalized into [0,1] with the formula  $\text{CP\_gain}[i, j] = \frac{\text{CP\_gain}[i, j]}{\max_{i, j} \{\text{CP\_gain}[i, j]\}}$  and  $\text{Global\_gain}[i, j] = \frac{\text{Global\_gain}[i, j]}{\max_{i, j} \{\text{Global\_gain}[i, j]\}}$ , respectively. In each variable assignment round, the variable assignment that has the largest weight and satisfies the memory space constraint is added into the

current variable partition VP. For example, in the first round of variable assignment of Fig. 5a, the Weight matrix is as follows:

$$\begin{aligned} \text{Weight} &= 0.17 * \begin{pmatrix} 1 & 0.97 & 0.97 \\ 0 & 0 & 0 \end{pmatrix} \\ &+ 0.83 * \begin{pmatrix} 0.73 & 0.53 & 0.53 \\ 1 & 0.07 & 0.4 \end{pmatrix} \\ &= \begin{pmatrix} 0.77 & 0.44 & 0.44 \\ 0.83 & 0.06 & 0.33 \end{pmatrix} \end{aligned}$$

The variable assignment (‘‘A’’, P<sub>2</sub>) is the result of first variable assignment round.

**Step 4.** After several rounds, due to the SPM capacity constraint, legal assignments cannot be found based on the Weight matrix. We will consider other variable assignment options. The pseudo-code of this step is shown in lines 13–18.

**Definition 4.6** (CPN, IBN and OBN) A *Critical Path Node* (CPN) is a node on the critical path. An *In-Branch Node* (IBN) is a node, from which there is a path reaching a CPN. An *Out-Branch Node* (OBN) is a node, which is neither a CPN and an IBN.

Compacting the latency of CPNs and IBNs is beneficial to compacting the schedule length. If there is no legal assignments related to CPNs, we will consider the assignments related to IBNs. Similar to CP\_gain, we construct a matrix IBN\_gain which represents the schedule length improvement from the variable assignment related to an IBN. Due to space limitations, we omit the definition IBN\_gain. We try to find the legal variable assignment with the largest value in IBN\_gain to be the result assignment of this round. When the available SPM space of processors related to CPNs and IBNs cannot accommodate any more variables, we will consider all other legal assignment options. In this case, we try to make full use of left over space of SPMs.

**Step 5.** When the variable assignment (var, P) is determined, the variable partition VP and free space of SPM<sub>P</sub> are updated. The weight coefficients  $\lambda_1, \lambda_2$ , are updated as well. When all the variables have been assigned or the left space of SPMs are not enough for any variables, the variable partition VP and compact schedule S<sub>comp</sub> are returned.

The two most time intensive portions of the GVP algorithm are path searching and weight matrix construction. The time complexity of CCP and PCP path

search is  $\Theta(|V|^2)$ . The time complexity of weight matrix construction is  $\Theta(|V||X||M|)$ , where  $|V|$  is number of task nodes,  $|X|$  is number of variable,  $|M|$  is number of processor. Thus, the overall time complexity of GVP algorithm is  $\Theta(|X||V|^2 + |X|^2|V||M|)$ .

### 4.3 Loop Pipeline Scheduling

In the previous section, list scheduling with variable partition for acyclic part of DFG has been introduced. Loops are usually the most time-critical portions of data-intensive applications. In order to improve overall throughput, loop pipeline scheduling—an optimization technique for cyclic DFG [16] needs to be exploited. This section discusses the proposed loop pipeline scheduling algorithm *Rotation Scheduling with Variable Partition* (RSVP) as shown in Algorithm 4.3. RSVP algorithm can produce a compact schedule by repeatedly applying retiming (introduced in Section 2.3) and remapping the rotated down nodes to new position of the schedule table. The novel feature of RSVP algorithm is that it takes variable partitioning into account. During loop pipeline scheduling, variable partitioning is executed only a few times when it is necessary.

**Step 1.** The initial schedule and variable partition are generated in this step. It is shown in lines 1–3 of Algorithm 4.3.  $S_{opt}$  is the reference schedule.

Lines 5–19 of Algorithm 4.3 is one round of rotation. We try to take some nodes from the first row of a schedule table and re-map them to new positions while keeping the remaining nodes in their original schedule spots. The variable partition must be taken into account in the node remapping process.

**Step 2.** We find out the rotatable node set RotSet and rotate these nodes down (lines 5–6). A rotatable node is a node in the first row of current schedule table  $S$  and the delays of its coming edges are larger or equal to one in the DFG. Then, we rotate down the nodes in RotSet by the retiming technique explained in Section 2.3. This operation is equivalent to shifting the iteration boundary down so that the nodes from the next iteration can be explored. After that, we get a new DFG  $G_r$ . For example, Fig. 7a and b illustrate the first round of RSVP for the DFG in Fig. 2. The rotatable nodes set is  $RotSet = \{T1\}$ . After retiming

---

#### Algorithm 4.3 Rotation Scheduling with Variable Partition

---

**Input:** (1)DFG:  $G$ , (2)VAFM:  $F$ , (3)MPSoC:  $M$

**Output:** (1)near-optimal variable partition:  $VP_{opt}$ , (2)near-optimal schedule:  $S_{opt}$

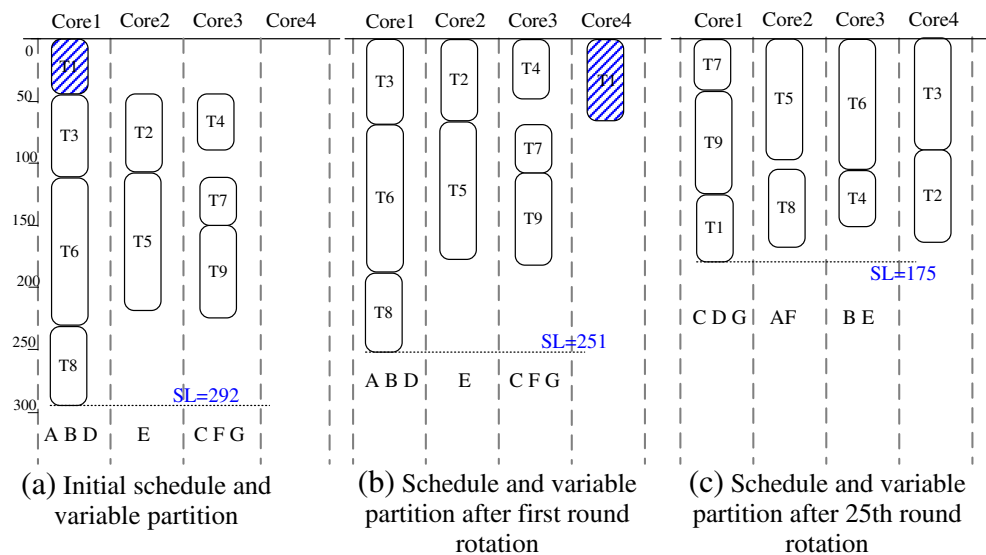
```

1:  $S \leftarrow list\_scheduling(G, M)$ ;
2:  $(S, VP) \leftarrow GVP(S, M, G, F)$ ; /* HAFF can be used as well */
3:  $S_{opt} \leftarrow S$ ;
4: for  $i = 1$  to  $Z$  do
5:    $RotSet \leftarrow find\ rotatable\ nodes$ ;
6:    $G_r \leftarrow retiming(G, RotSet)$ ;
7:   if  $G_r = G$  then
8:     break; /* If  $G_r$  is identical  $G$ , it is not necessary to continue rotation */
9:   end if
10:   $S_{rot} \leftarrow remapping(RotSet, VP, S, F)$ ;
11:  if  $SL(S_{rot}) > SL(S)$  then
12:     $S' \leftarrow list\_scheduling(G_r, M)$ ; /* The upper bound of execution latencies are used */
13:     $(S_{rot}, VP) \leftarrow GVP(S', M, G, F)$ ; /* HAFF or other algorithms can be used for repartitioning */
14:  end if
15:   $S \leftarrow S_{rot}$ ;
16:  if  $SL(S) < SL(S_{opt})$  then
17:     $S_{opt} \leftarrow S$ ;
18:     $VP_{opt} \leftarrow VP$ ;
19:  end if
20: end for
21: Return  $S_{opt}$  and  $VP_{opt}$ ;

```

---

**Figure 7** Rotation scheduling with variable partition (a–c).



$r(T1) = 1$ , the DFG in Fig. 2a is transformed into  $G_r$  as shown in Fig. 3b.

**Step 3.** Remap the nodes in RotSet to appropriate positions of the new schedule table. There are two factors that need to be taken into account in remapping. First is the dependency constraints in the DAG part of  $G_r$ . Second is current variable layout. The pseudo-code of the remapping is shown in Algorithm 4.4.

Since the latency of tasks are different, when the rotatable nodes are rotated down, there may be some different idle time slots in the beginning of some processors. Before the nodes are remapped, we reschedule the nodes in  $\{V - RotSet\}$  to fill up those idle time slots. Note that the processor allocation and relative start order of these nodes are not changed in this rescheduling.

**Algorithm 4.4** Remapping

**Input:** (1)a rotatable node set: RotSet, (2) variable partition: VP, (3) schedule: S, (4)VAFM:F, (5)retimed DFG:  $G_r$

**Output:** a new schedule:  $S_{rot}$

- 1: reschedule the note in  $\{V - Rotset\}$  to fill up the idle time slots
- 2:  $list_{rp} \leftarrow$  generate a ordered list by descending order of  $t_{up}$  ;
- 3:  $h \leftarrow$  head of  $list_{rp}$  ;
- 4: **while** ( $h \neq NULL$ ) **do**
- 5:      $t_{rp} \leftarrow \infty$ ;
- 6:     **for**  $i = 1$  to  $|M|$  **do**
- 7:          $t_{temp} \leftarrow$  find the earliest start time for  $h$  on  $P_i$  while maintaining the dependencies defined in  $G$  and not overlapping with other nodes already been on  $P_i$ ;
- 8:         **if** ( $(t_{temp} + Latency(h, P_i)) < t_{rp}$ ) **then**
- 9:              $P_{target} \leftarrow P_i$ ;
- 10:         **end if**
- 11:     **end for**
- 12:      $S_{rot} \leftarrow$  Schedule the  $h$  onto  $P_{target}$ ;
- 13:      $h \leftarrow h.next$ ;
- 14: **end while**
- 15: **return**  $S_{rot}$

The basic principle of remapping is to make the finish times of a rotatable task as soon as possible. To obtain shorter schedule lengths, the tasks with longer  $t_{up}$  have higher priority to be remapped first (line 2 of Algorithm 4.4). The processor on which  $h$  can finish first is the target processor for remapping  $h$  (lines 6–11 of Algorithm 4.4). After all the rotatable nodes have been remapped, a new schedule is returned. For example, in Fig. 7b is a new schedule where task1 has been mapped to Core4. Even though the latency of task 1 increases, the schedule length is compacted to 251 control step.

**Step 4.** The pseudo-code of this step is shown in lines 11–14 of Algorithm 4.3. We compare the schedule length of  $S_{rot}$   $SL(S_{rot})$  with the length of last round schedule  $SL(S)$  (line 11). If the  $SL(S_{rot})$  is longer than  $SL(S)$ , we will generate a new schedule  $S'$  for  $G_r$  by list scheduling and use GVP (Algorithm 4.2) to repartition all the variables. Note that the increase of  $SL(S_{rot})$  is not only caused by the variable partition, but also by the length of critical path which is varied by retiming. Even though variable repartitioning and rescheduling are executed, the length of  $S_{rot}$  may be still longer than reference length. RSVP algorithm continues iterating on this new state ( $S_{rot}$ ) instead of returning to the previous schedule state in hopes of reducing the schedule length in future iterations.

**Step 5.** At the end of each iteration, the best schedule and corresponding variable partition among all previous rotation rounds are saved.

The first termination condition is an argument  $\mathcal{Z}$ , which restricts the number of rotation iterations. When the algorithm reaches this value, the shortest schedule and corresponding variable partition are returned.  $\mathcal{Z}$  can be any integer value, but according to experimental result we define  $\mathcal{Z}$  as Eq. 11 to get near-optimal schedule.

$$\mathcal{Z} = \frac{SL(S_{init})}{\min\{t_{up}(task_i)\}} * |M| \quad (11)$$

where  $SL(S_{init})$  is schedule length of initial schedule,  $|M|$  is number of processors in the MPSoC. Using the example in Fig. 2, after the 25th round of rotation, we obtain a very compact pipelined schedule shown in Fig. 7c, which is only 59.9% the length of the unpipelined schedule. Note that after several rounds of rotation,  $G_r$  may be the same as the original input  $G$ . If this happens it is not necessary to do any more rotation rounds. We check for this termination condition in line 7.

## 5 Experiments

To evaluate the effectiveness of the proposed algorithms, we conduct experiments on programs from MiBench [22]. The benchmarks used in our experiments include three kind of representative embedded system applications, multimedia, telecommunication and security. In this paper, the measure of effectiveness of an algorithm is the schedule length of its generated schedules in terms of millions of clock cycles (M cycles). The result of these experiments show that the proposed algorithms can improve the performance of real applications. Table 1 lists the number of tasks and critical variables in each of the eight benchmarks used in our study. Note that even though each benchmark uses many variables, the access frequency of most variables are very low. We only consider the critical variables, variables of which the access count is not less than 1% of total cycle count. Since the layout of non-critical variables has very limited impact on overall performance, we assign them to SPMs or off-chip memory after all the critical variables haven been assigned to SPMs.

To prepare our experiments, we transform C programs of benchmarks to DFGs. We modify a task graph generation tool TGE [14]. We use the modified TGE to divide applications into coarse-grain tasks and statically analyze the programs to extract the critical variables set. In the graph generation process, we guarantee the CCR is no less than 95%, the communication cost can be neglected in scheduling. Second, we use a dynamic analysis tool Valgrind [23] for application profiling. We assume that the on-chip memory is an unlimited sized unified shared memory, an access latency is 10 or 5 clock cycles. The profiling information includes cycle-accurate execution time and memory reference of tasks, from which we can determine the  $t_{up}$  and  $t_{low}$  of each task and the VAFM. All the experiments are run on a 2.8 GHZ Pentium 4 PC running Fedora 9 with 1G of DRAM.

The result of the first experiment shown in Table 2 compares the schedule lengths generated by list scheduling with three different variable partition approaches. The target MPSoC has four processor cores. The latency of remote data access cost 10 clock cycles and each SPM is 64k. The total capability of the four SPM is bigger than total variables size of an application. The three different variable partition approaches are: (1) Integrated data assignment with scheduling (IDAS); (2) HAFF as Algorithm 4.1; (3) GVP as Algorithm 4.2. In IDAS, when a task scheduled onto a processor core, first we try to assigned the related variables of the task to local SPM. If the size of variable exceed capacity of local SPM, it will be assigned to

**Table 1** Characteristics of the benchmarks.

Benchmark	# Tasks	# Vars	Benchmark	# Tasks	# Vars
Blowfish enc	12	11	PGP sign	35	8
CRC32	7	6	FFT	25	9
GSM enc	28	12	JPEG	6	15
Lame	16	10	Mad	20	8

remote SPMs. IDAS is a naive variable assignment schema, no actually variable partitioning executed.

The “SL” columns represent the schedule length of different approaches. The column  $impr_{H:I}$  shows the schedule length improvement of HAFF over IDAS algorithm, and the average improvement is 23.74%. From column  $impr_{G:I}$ , we can see the significant improvement of GVP algorithm compare to IDAS. The average improvement is 31.91%, the largest improvement is 48.97% for benchmark “JPEG”. The last column  $impr_{G:H}$  illustrates that the performance of GVP algorithm is better than HAFF 10.40% in average. The critical factor is that GVP has more effective priority metrics for solving partition in global view than local greedy algorithm HAFF. The results show that the algorithm jointly considering scheduling and variable partitioning outperforms the simply data assignment method.

For the second experiment, all conditions are the same as the first experiment except the remote SPM access latency, which is now 5 clock cycles. Fig. 8 shows the relative performance of HAFF and GVP algorithms normalized to the IDSA. The lower bars on the graph indicate better performance. The last group of bars compare the average schedule length of these three methods.

The results illustrate that GVP and HAFF also perform well under lower access latency conditions. In particular, the GVP is very efficient for all the benchmarks.

The first two experiments show variable partitioning is critical for performance improvements in MPSoC with VS-SPM. Our proposed algorithms are good at reducing the number of remote accesses, particularly we can obtain 21.3% average improvement from GVP

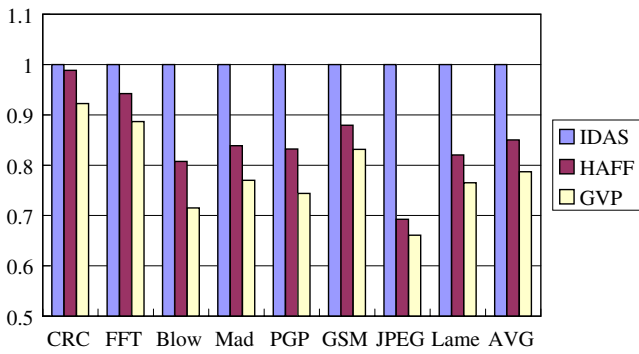
in low access latency condition. However, in above two experiments, the proposed algorithms are only compared with very simple variable assignment schema (IDAS). The next two experiments will compare them with the optimal variable partition.

In the third experiment, we compare the variable partition generated by our GVP algorithm with the optimal variable partition. Since there is no linear programming model can exactly match with our problem, we use exhaustive search to achieve the optimal solutions. The three different approaches in this experiment are: (1)First, we exhaustively search all the possible variable partitions. Then, we schedule tasks based on each variable partition by list scheduling to get the optimal schedule. The related results of this approach shown in “ES-list” column of Table 1. (2) List scheduling with variable partition generated by GVP, shown in column “GVP-list”. (3) Integrate the GVP into RSVP to generate pipelined schedules, shown in column “RSVP”. In addition to the eight benchmarks in Table 1, we add other 4 synthesized benchmarks to evaluate our algorithms. The name of each benchmark represents the number of tasks in the benchmark. For example, “ben-T23” means the DFG of this benchmark has 23 tasks. The second column of Table 3 represents the number of critical variables in each benchmark.

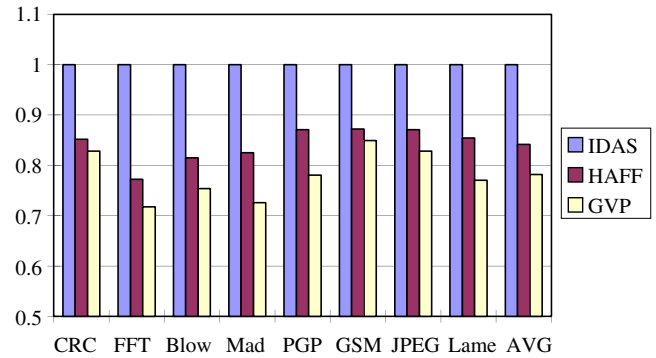
The “Time cost” column represents the running times of ES-list approach for each benchmark. One observation from this column is that running time of exhaustive search increase dramatically when the number of variables increased. For example, obtaining the optimal result of benchmark “JPEG”, which only has 15 critical variables, needs about two and a half hours. When the number of critical variables increases to 18

**Table 2** Comparison of the schedule lengths of IDAS, HAFF and GVP.

Benchmark	IDAS	HAFF	GVP			
	SL(M cycles)	SL(M cycles)	$impr_{H:I}$	SL(M cycles)	$impr_{G:I}$	$impr_{G:H}$
CRC32	89.5	72.8	18.66%	68.2	23.80%	6.32%
FFT	81.8	67.1	17.97%	57.8	29.34%	13.90%
Blowfish enc	66.1	44.6	32.53%	40.8	38.28%	8.52%
Mad	60.1	46.2	23.13%	40.9	31.95%	11.50%
PGP sign	76.5	59.6	22.09%	51.7	32.42%	13.30%
GSM	95.6	84.7	11.40%	74.9	21.65%	11.60%
JPEG	48.4	25.8	46.69%	24.7	48.97%	4.26%
Lame	332	274	17.47%	236	28.92%	13.9%
Average	–	–	23.74%	–	31.91%	10.40%



**Figure 8** Compare IDAS, HAFF and GVP with 5-clock cycle latency.



**Figure 9** Rotation scheduling with IDAS, HAFF and GVP.

as benchmark “lame”, an optimal result can not be obtained in a reasonable amount of time. It is denoted by the symbol “×” in Table 3. Since the maximum solution space for exhaustive search is  $|M|^{|X|}$ , the solution space may be decreased a little when the capacity of SPMs is taken into account. However, from our experimental results, we know that if the number of shared variables exceed 12, the exhaustive search takes a very long time. When the number of variables is more than 17, exhaustive search is not applicable. On the contrary, the running time of GVP for all the benchmarks did not exceed one minute.

The percentage difference between the schedule length of ES-list and GVP-list are shown in “Diff”. We can see the average schedule length of GVP-list is only 8.74% longer than results of optimal schedule. In some case, the results generated by GVP-list are very close to the optimal solutions. For example, the schedule length of benchmark “blow” is only 1.4% longer than optimal schedule. For the loop intensive applications, the GVP-RSVP outperforms non-pipeline approach. The average improvement over ES-list is 25.96%. Al-

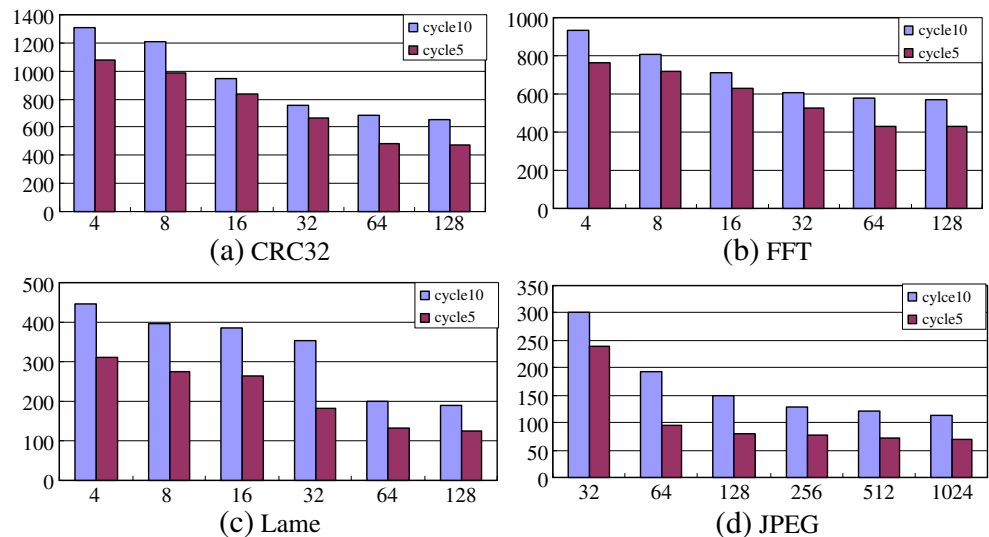
though the performance of loop pipeline is limited by the inherent parallelism of application, the GVP-RSVP can explore the parallelism as far as possible. For example, the improvement of GVP-RSVP for “CRC32” over ES-list is 45.53%. From Algorithm 4.3, we know that rotation is an iterative process, and the variables partitioning may be repeat many times. So exhaustive search variable partitioning cannot be executed in the RVSP algorithm. GVP has very good performance with respect to output schedule length and running time. Particularly, our technique can significantly improve the overall throughput for the applications with high inherent parallelism.

In the fourth experiment, we assume the remote SPM access latency is 5 clock cycles. We compare the schedule length of rotation scheduling with three variable partition algorithms described in experiment one. The results of rotation with HAFF and GVP are normalized to rotation with IDSA shown in Fig. 9. The average improvement of GVP and HAFF are 21.8% and 15.8%. Another fact indicated by Fig. 9 is that, in several cases, such as “CRC32” and “JPEG”, the

**Table 3** Comparison of GVP with exhaustive search.

Benchmark	# Vars	ES-List		GVP-List		GVP-RSVP	
		SL (M cycles)	Time cost (s)	SL (M cycles)	Diff	SL (M cycles)	impr <sub>GR:ES</sub>
CRC32	6	63.7	0.08	68.2	7.06%	34.7	45.53%
FFT	8	53.3	1.16	57.8	8.44%	42.9	19.51 %
blow	11	40.2	71.23	40.8	1.49 %	28.5	29.1 %
mad	12	36.2	269.31	40.9	13.00 %	24.1	33.43 %
pgp	14	49.5	3925.37	51.7	4.44 %	38.1	23.03 %
GSM	14	68.7	4108.44	74.9	9.02 %	51.8	24.6 %
jpeg	15	22.4	8841.02	24.7	10.3 %	19.3	13.84 %
lame	18	×	×	236.1	×	201.4	×
ben-T23	9	122.0	6.31	132.2	8.36%	89.3	26.8%
ben-T35	10	83.5	15.87	94.2	12.8%	66.5	20.36%
ben-T28	11	321.9	69.42	349.5	8.57%	243.8	24.26 %
ben-T40	13	210.2	841.38	236.8	12.7%	157.4	25.12 %
Average	–	–	–	–	8.74%	–	25.96%

**Figure 10** Schedule lengths vary with capacities of SPMs (a–d).



results of HAFF approximate to GVP. According to the termination condition of RSVP Eq. 11, the number of rotation round related to the problem scale. When the problem scale is very large, the low time complexity algorithm HAFF is an acceptable partition algorithm for RSVP.

We also conducted experiments to study the impact of SPMs capability on the schedule length. Fig. 10 shows the relationship between schedule length and SPMs capability. The schedules are produced by rotation with GVP and HAFF algorithms. The capability of each SPM varies from 4 k to 1 M and 4 processor cores. The experimental results show that the increasing of SPMs capability does improve the schedule lengths. However, when the capability is larger than a threshold, there is no further improvement from increasing SPMs capability. For example, in Fig. 10d, when the SPM size is larger than 64 k, the improvement is very little. These experimental results also show that our GVP algorithm outperforms HAFF algorithm for any SPM budget.

## 6 Conclusion and Future Work

From above sections, we know that the variable partition is very important for MPSoC with VS-SPM. The proposed graph model DFG is efficient for presenting variables accesses information and loop carried dependencies. In GVP algorithm, the novel “Weight” matrix estimates the possible variable assignments with global view, which helps GVP to generate a near-optimal variable partition. The performance of HAFF is also good when the remote access latency is low. Both GVP and HAFF are very efficient for large-scale problems. Our

loop pipeline technique RSVP is an effective heuristic for loop-intensive application. It exploits the loop parallelism sufficiently with only a few variable repartitioning. The experimental results on selected benchmarks show that our proposed algorithms are promising and it can be applied to different remote access latency conditions and different SPMs capacity.

In the future, we plan to extend our work on MPSoC with VS-SPM in two directions. First, we will improve our task graph extraction technique to explore more parallelism and balance the granularity of tasks and the number of variables. Second, our algorithms will be made more adaptive to more complex architecture, such as multi-level on-chip memory hierarchy [24] and heterogenous processor cores systems.

## References

1. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., & Marwedel, P. (2002). Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign* (pp. 73–78).
2. Motorola Corporation (1998). *Mmc2001 reference manual*. [http://www.motorola.com/SPS/MCORE/info\\_documentation.htm](http://www.motorola.com/SPS/MCORE/info_documentation.htm).
3. Texas Instruments (1997). *Tms370cx7x 8-bit microcontroller*. <http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf>.
4. Motorola Corporation (2000). *Cpu12 reference manual*. <http://e-www.motorola.com/brdata/PDFDB/MICROCONTROLLERS/16BIT/68HC12FAMILY/REFMAT/CPU12RM.pdf>.
5. Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., & Parikh, A. (2001). Dynamic management of scratch-pad memory space. In *DAC '01: Proceedings of the 38th conference on Design automation* (pp. 690–695).

6. Xue, C., Shao, Z., Liu, M., Qiu, M., & Sha E. H. M. (2006). Loop scheduling with complete memory latency hiding on multi-core architecture. In *ICPADS '06: Proceedings of the 12th international conference on parallel and distributed systems* (pp. 375–382).
7. Chen, T.-F., & Baer, J.-L. (1998). A performance study of software and hardware data prefetching schemes. *International Symposium on Computer Architecture*, 223–232.
8. Chen, F., ONeil, T. W., & Sha, E. H.-M. (2000). Optimizing overall loop schedules using prefetching and partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 11(6), 604–614.
9. Wang, Z., Sha, E. H.-M., & Wang, Y. (2002). Partitioning and scheduling dsp applications with maximal memory access hiding. *EURASIP Journal on Applied Signal Processing*, 9, 926–935.
10. Kandemir, M., Ramanujam, J., & Choudhury, A. (2002). Exploring shared scratch pad memory space in embedded multiprocessor system. In *DAC '02: Proceedings of the 39th conference on design automation* (pp. 219–224).
11. Terechko, A., Le Thénaff, E., & Corporaal, H. (2003). Cluster assignment of global values for clustered vliw processors. In *CASES '03: Proceedings of the 2003 international conference on compilers, architecture and synthesis for embedded systems* (pp. 32–40).
12. Suhendra, V., Raghavan, C., & Mitra, T. (2006). Integrated scratchpad memory optimization and task scheduling for mp-oc architectures. In *CASES '06: Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems* (pp. 401–410).
13. Ozturk, O., Chen, G., Kandemir, M., & Karakoy, M. (2006). An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *ISVLSI '06: Proceedings of the IEEE computer society annual symposium on emerging VLSI technologies and architectures* (p. 50).
14. Vallerio, K. S., & Jha, N. K. (2003). Task graph extraction for embedded system synthesis. In *VLSID '03: Proceedings of the 16th international conference on VLSI design* (p. 480).
15. Chao, L.-F., LaPaugh, A. S., & Sha, E. H.-M. (1997). Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer-Aided Design*, 16(3), 229–239.
16. Aiken, A., & Nicolau, A. (1988). Optimal loop parallelization. *SIGPLAN Notices*, 23(7).
17. Chao, L.-F., & Sha, E. H.-M. (1997). Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), 1259–1267.
18. Ozturk, O., Kandemir, M., Chen, G., Irwin, M. J., & Karakoy, M. (2005). Customized on-chip memories for embedded chip multiprocessors. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation* (pp. 743–748).
19. Meftali, S., Gharsalli, F., Rousseau, F., & Jerraya, A. A. (2001). An optimal memory allocation for application-specific multiprocessor system-on-chip. In *ISSS '01: Proceedings of the 14th international symposium on systems synthesis* (pp. 19–24).
20. Panda, P. R., Dutt, N. D., & Nicolau, A. (2000). On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3), 682–704.
21. Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., & Shippy, D. (2005). Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 589–604.
22. Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the workload characterization, 2001. WWC-4. 2001 IEEE international workshop* (pp. 3–14).
23. Valgrind (2009). Valgrind homepage. <http://www.valgrind.org>.
24. Chen G., Ozturk, O., Kandemir, M., & Irwin, M. J. (2006). Multi-level on-chip memory hierachy design for embedded chip multiprocessor. In *ICPADS '06: Proceedings of the 12th international conference on parallel and distributed system* (pp. 383–390).



**Lei Zhang** received M.S. degree in computer science from University of Electronic Science and Technology of China, where he is currently a computer science Ph.D. candidate. He had been awarded a fellowship from the China Scholarship Council to conduct research at University of Texas at Dallas as a visiting scholar from 2007 to 2009. His research interests include embedded systems, compiler optimization and hardware/software co-design.



**Meikang Qiu** received M.S. and Ph.D. degrees of Computer Science from University of Texas at Dallas. He is an assistant professor of Electrical and Computer Engineering at University of New Orleans. He is an IEEE Senior member. His research interests include embedded systems, and computer security.



**Wei-Che Tseng** received B.S degree in Electrical Engineering from the University of Texas at Dallas. He is currently a Computer Science Ph.D. candidate at the University of Texas at Dallas. His research interests include computer architecture and high level synthesis.



**Edwin Hsing-Mean Sha** received Ph.D. degree from the Department of Computer Science, Princeton University, Princeton, NJ, in 1992. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas. He has published more than 250 research papers in refereed conferences and journals. He has served as editors for many journals, and as program committee and Chairs for numerous international conferences.

He received Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award and NSFC Overseas Distinguished Young Scholar (B) Award. His web page can be found in <http://www.utdallas.edu/~edsha>.