



Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Loop scheduling and bank type assignment for heterogeneous multi-bank memory

Meikang Qiu^{a,*}, Minyi Guo^b, Meiqin Liu^c, Chun Jason Xue^d, Laurence T. Yang^e, Edwin H.-M. Sha^f

^a Department of Electrical and Computer Engineering, University of New Orleans, 2000 Lakeshore Dr., New Orleans, LA 70148, USA

^b Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

^c College of Electrical Engineering, Zhejiang University, Yuquan Campus, Hangzhou 310027, China

^d Department of Computer Science, City University of Hong Kong, Hong Kong

^e Department of Computer Science, St. Francis Xavier University, Antigonish, NS, B2G 2W5, Canada

^f Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA

ARTICLE INFO

Article history:

Received 5 January 2008

Received in revised form

23 September 2008

Accepted 8 February 2009

Available online 6 March 2009

Keywords:

Type assignment

Heterogeneous

Low power design

Multi-bank memory

Loop scheduling

ABSTRACT

Many high-performance DSP processors employ multi-bank on-chip memory to improve performance and energy consumption. This architectural feature supports higher memory bandwidth by allowing multiple data memory accesses to be executed in parallel. However, making effective use of multi-bank memory remains difficult, considering the combined effect of performance and energy requirement. This paper studies the scheduling and assignment problem about how to minimize the total energy consumption while satisfying the timing constraint with heterogeneous multi-bank memory for applications with loop. An algorithm, TASL (*Type Assignment and Scheduling for Loops*), is proposed. The algorithm uses bank type assignment with the consideration of variable partition to find the best configuration for both memory and ALU. The experimental results show that the average improvement on energy-saving is significant by using TASL.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Memory access latency and energy consumption are two of the most important design considerations in memory architecture. A number of papers have investigated how to exploit multi-bank memory from a single aspect: improving performance or increasing energy savings. However, the combined effect of both performance and energy requirements is seldom tackled because increased performance often conflicts with energy savings. In high-performance *digital signal processing* (DSP) applications, strict real-time processing is critical [42] since the growing speed gap between CPU and memory becomes a bottleneck for designing such real-time systems. In order to close this speed gap, embedded systems need to utilize multi-bank on-chip memories [35,34]. The high energy consumption of memories makes them target of many energy-conscious optimization techniques [4]. This is especially true for mobile applications, which are typically memory-intensive. This paper focuses on the problem of reducing the total

energy consumption while satisfying performance constraints for loop applications with multi-bank memory architectures.

In many advanced memory architectures, there are heterogeneous memory banks. Different memory banks have different memory access latencies and energy consumptions for same operations [14,27,3,11]. A certain memory bank type may access the data stored slower but with less energy consumption, while another bank type will access the data faster with higher energy consumption. Also, there is a limitation of how many banks can be accessed simultaneously in certain memory architectures. Therefore, an important problem arises: how to assign types to the banks selected and partition variables for an application to minimize the total energy consumption while satisfying timing constraints.

Much research has been conducted in the area of using multi-bank memory to achieve maximum instruction level parallelism, i.e., optimize performance [30,8,19,20,26,38]. These approaches differ in either the models or the heuristics. However, they seldom consider the combined effect of performance and energy requirements. Actually, performance requirement often conflicts with energy saving [9,17,32,10,24,39]. There is a trade off between energy consumption and performance. Usually, improved performance is achieved at the cost of higher energy consumption if the user does not carefully study the intricate relationship between performance and energy of a system. By exploiting

* Corresponding author.

E-mail addresses: mqiu@uno.edu (M. Qiu), guo-my@cs.sjtu.edu.cn (M. Guo), liumeiqin@zju.edu.cn (M. Liu), jasonxue@cityu.edu.hk (C.J. Xue), lyang@stfx.ca (L.T. Yang), edsha@utdallas.edu (E.H.-M. Sha).

heterogeneous multi-bank memory at the instruction level, significant improvement of both energy saving and performance can be obtained. Wang et al. [33] have considered the combined effect and proposed the VPIS algorithm to improve both energy saving and performance, but their algorithm does not fully exploit the heterogeneous multi-bank memory architecture. We also use loop scheduling to further improve energy saving and performance.

Combining both energy and performance considerations for both memory bank and ALU, in this paper, we propose a novel graph model to overcome the weaknesses of previous works. We design an algorithm, TASL (*Type Assignment and Scheduling for Loops*), to minimize the total energy consumption while satisfying performance requirements. The experimental results show that TASL achieves a significant reduction on average in total energy consumption. For example, using the SPAM compiler (Princeton Spam Compiler Project, <http://www.idiom.com/free-compilers/TOOL/SPAMComp-1.html>) with 3 memory types and 3 ALU types, compared with the VPIS algorithm [33], TASL shows an average 16.2% reduction in total energy consumption.

In summary, the main contributions of this paper are the following. First, we study the combined effects of energy saving and performance of memory and ALU in a systematic approach. Second, we exploit the energy saving with type assignment and minimum resource scheduling for both memory and ALU. Third, to the best of our knowledge, our paper is the first to consider the combined effect of both energy and performance with heterogeneous multi-bank memory. Fourth, we obtain the best results by rescheduling nodes repeatedly based on loop scheduling. Fifth, we improve the heterogeneous scheduling and assignment for both ALU and memory simultaneously.

In the next section, we introduce basic concepts and models. An example is shown in Section 3. The algorithm is discussed in Section 4. We show our experimental results in Section 5. Related work and Concluding remarks are provided in Sections 6 and 7, respectively.

2. Basic concepts and models

In this section, we introduce some basic concepts which will be used in the later sections. First, the *data flow graph* (DFG) for modeling heterogeneous multi-bank memory and multi-type ALU architecture is described. Then, the basic concepts of retiming and rotation scheduling are introduced, followed by the concepts of variable partition and *Variable Independence Graph* (VIG). Finally, we provide the formal definition of the heterogeneous multi-bank type assignment problem.

2.1. Data flow graph

Data flow graph (DFG) is used to model many multimedia and DSP applications. We use a *cyclic data flow graph* (CDFG) to denote a loop in our work. The definition is as follows:

Definition 2.1. A CDFG $G = \langle U, ED, d, T, E \rangle$ is a node-weighted and edge-weighted directed cyclic graph, where $U = \langle u_1, \dots, u_i, \dots, u_N \rangle$ is a set of operation nodes; $ED \subseteq U \times U$ is an edge set that defines the precedence relations among nodes in U ; $d(ed)$ is a function to represent the number of delays for any edge $ed \in ED$; The edge without delay represents the intra-iteration data dependency; the edge with delays represents the inter-iteration data dependency and the number of delays represents the number of iterations involved. T is a set of operation time for all nodes in U ; E is a set of energy consumption for all nodes in U .

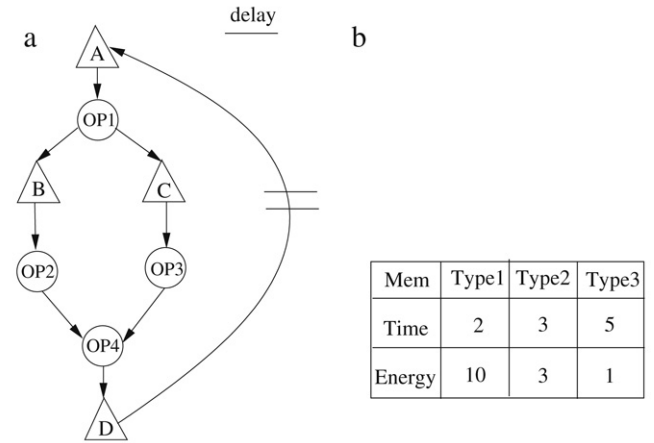


Fig. 1. (a) A DFG with both memory and ALU operations. (b) The types of memory.

CDFG $G = \langle U, ED, d, T, E \rangle$ is a sub case of general DFG $G = \langle U, ED, T, E \rangle$. Fig. 1(a) shows a DFG. The ALU operations are represented by circles, and the memory operations are represented by triangles. Particularly, an edge from a memory operation node to an ALU operation node represents a *Load* operation, whereas the edge from an ALU operation node to a memory operation node represents a *Store* operation. We start from loading value of variable A into the ALU to perform operation 1, i.e., $OP1$. Then variable B is loaded to implement $OP2$, and similarly is for $OP3$. Next, $OP4$ is performed utilizing the inputs of the results of $OP2$ and $OP3$. Finally, the result of $OP4$ is stored into variable D in memory.

In this example, only 2 banks are available, i.e., can be accessed at the same time. There are 3 types of banks to choose from and we can use only 2 types of banks maximum. The memory bank types are shown in Fig. 1(b). Type 1 has memory access time 2 clock cycles with energy consumption 10 nJ; Type 2 has memory access time 3 cycles with energy consumption 3 nJ; The access time is 5 cycles and energy consumption is 1 nJ for type 3. We can represent the memory bank types in Type(Time, Energy) format, such as type 1(2, 10). Regarding the ALU part, we use two homogeneous ALUs with time 1 cycle and energy consumption 0.5 nJ. There is a timing constraint L and it must be satisfied for executing the whole DFG, including both memory access part and ALU part.

2.2. Retiming and rotation scheduling

Static schedule: From the cyclic DFG of an application, we can obtain a static schedule. A static schedule of a cyclic DFG is a repeated pattern of an execution of the corresponding loop. In our work, a schedule implies both assignment and allocation of a control step. A static schedule must obey the dependency relations of the Directed Acyclic Graph (DAG) portion of the DFG. The DAG is obtained by removing all edges with delays in the DFG. Fig. 2 (a) shows the corresponding static schedule to Fig. 1(a).

Retiming: Retiming [18] is an optimal scheduling technique for cyclic DFGs considering inter-iteration dependencies. It can be used to optimize the cycle period of a cyclic DFG by evenly distributing the delays. Retiming generates the optimal schedule for a cyclic DFG when there is no resource constraint. Given a cyclic DFG $G = \langle U, ED, d, T, E \rangle$, retiming r of G is a function from U to integers. For a node $u \in U$, the value of $r(u)$ is the number of delays drawn from each of incoming edges of node u and pushed to all of the outgoing edges. Let $G_r = \langle U, ED, d_r, T, E \rangle$ denote the retimed graph of G with retiming r , then $d_r(ed) = d(ed) + r(u_1) - r(u_2)$ for every edge $e(u_1 \rightarrow u_2) \in ED$.

Rotation scheduling: Rotation Scheduling [5] is a scheduling technique used to optimize a loop schedule with resource

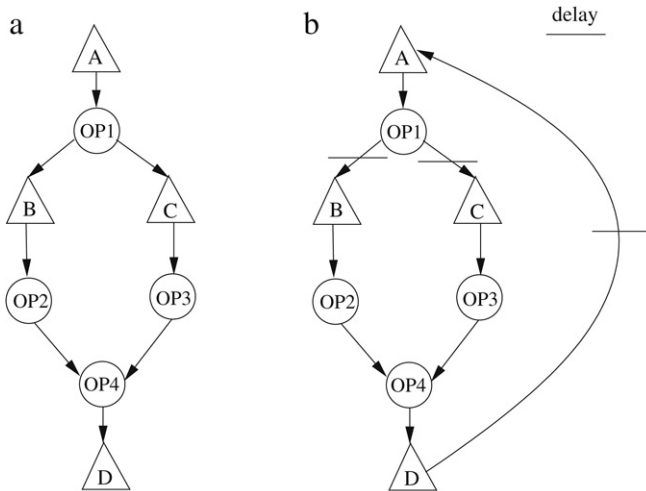


Fig. 2. (a) The static schedule of Fig. 1(a) by removing the edge with delays. (b) The rotated DFG.

constraints. It transforms a schedule to a more compact one iteratively in a DFG. In most cases, the minimal schedule length can be obtained in polynomial time by rotation scheduling. Fig. 2(b) shows an example to explain how to obtain a new schedule via rotation scheduling. We use the schedule generated by list scheduling from Fig. 1(a) as an initial schedule. List scheduling makes an ordered list of processes by setting the priority of a node as the longest path from this node to a leaf node [23], and then repeatedly executing the following two steps until a valid schedule is obtained: (1) select from the list, the process with the highest priority for scheduling, (2) select a resource to accommodate this process. We get a set of nodes at the first row of the schedule (in this case, it is $\{A\}$); and we rotate node A down. The rotated graph is shown in Fig. 2(b).

2.3. Variable partition and variable independence graph (VIG)

Variable partition [42,33] is an important method to improve the data locality. Different variable partitions may significantly affect the schedule length and energy consumption of an application. In order to properly partition variables, we use VIG (Variable Independence Graph) to expose all parallel memory accesses in a DFG [42]. The nodes of the graph represent variables, and the edges in the graph represent potential parallelism existing among the memory accesses for these variables.

Definition 2.2. A VIG is an undirected weighted graph $G_v = \langle U, ED, w \rangle$, where U is a set of nodes representing variables, and $ED \subseteq U \times U$ is a set of edges connecting between nodes in U , whose memory operations can be executed in parallel potentially. Function $w(u_1, u_2)$ maps from ED to a set of real values representing a priority of partitioning nodes u_1 and u_2 to different memory banks of an edge $u_1 \rightarrow u_2 \in ED$, $u_1, u_2 \in U$.

In order to capture the tradeoff between the desire of parallelism and that of serialism, Wang et al. [33] used two lists of weights. One is the list of possibility weights, which is proposed by Zhuge et al. [42] and referred as parallelism weights. The second is the list of weights that is referred as serialism weights. The goal of introducing the serialism weights is to model the possibility of serializing a pair of operations without sacrificing performance. In this paper, we use both parallelism weights and serialism weights to build a VIG.

For example, assume there are two memory banks and two ALUs. Both banks are of type 1 (2, 10). Fig. 3(a) shows the schedule 1

with B and C in different banks. Thus, we can fully take advantage of parallelism by assigning variable B to M1 and variable C to M2 at the same time unit 4 and 5. The schedule length is only 9. When we group B and C into the same bank, then the schedule length changes to be 11. In this case, we must use B and C in serial, and cannot take advantage of the parallelism. The corresponding schedule 2 is shown in Fig. 3(b). Therefore, it is apparent that variable partition will affect the schedule length and the total time and energy consumption for the DFG of an application.

In the following, we introduce some important concepts that will be used in constructing a complete VIG. During the graph construction, we will be particularly interested in some memory operation pairs that help us identify the parallel memory accesses. We call them “independent pairs”. For example, the nodes B and C in Fig. 1(a) are independent pairs.

Definition 2.3. Given DFG $G = \langle U, ED, T, E \rangle$, if nodes $u_1, u_2 \in U$, are not reachable from each other through any path without delay in G , nodes u_1 and u_2 are independent pairs.

Below, we will introduce the concept of the *mobility window*. Given a DFG $G = \langle U, ED, T, E \rangle$, a *mobility window* [23] of node $u \in U$, which is denoted by $MW(u)$ in this paper, is a set of time units in a static schedule by which node u can be placed. The first time unit where the node u can be scheduled is determined by *as soon as possible* (ASAP) scheduling, and the last control step by which node u can be scheduled is determined by *as late as possible* (ALAP) scheduling with the longest path as a time constraint. Mobility window gives the earliest and the latest position in which a node can be scheduled. Note that the overlap of mobility windows of two nodes indicates the possibility that the nodes could be scheduled in the same time unit. The mobility property of a node in a schedule is very important in improving the preciseness in the graph construction.

In the following, we define the priority function of an edge in VIG based on mobility windows of two parallel memory accesses. We use the cardinality of mobility window overlap to denote the possible occurrences of parallel operations and use the multiplication of the cardinalities of two mobility windows to denote all arrangements of two nodes in a schedule. We define the variable partition problem as follows:

Definition 2.4. Given a VIG $G_v = \langle U, ED, w \rangle$, and let n to be the number of partitions required, the variable partitioning problem is to partition U into n disjoint sets P_1, P_2, \dots, P_n , such that the total $w(u, v), \forall u \in P_i, \forall v \in P_j, \forall i, j = 1, \dots, n$, is maximum.

A VIG can be built in various ways, depending on how accurately the graph conveys the potential memory access parallelism in the program. Different graph constructions can lead to different variable partitioning results. For the variable partitioning problem that aims to produce a shorter schedule, the accuracy of the VIG is limited by the unknown positions of the memory operations in the schedule. We build a history table and use profiling to predict unknown positions of the memory operations [25,36]. We would like to provide a complete and accurate view for variable partitioning as much as possible, but on the other hand, we also would like to maintain the flexibility so that the partitioning process can work with different scheduling algorithms. The intricacy of building the graph model for the variable partitioning problem is how to keep certain level of accuracy of the parallelism and still have a graph working for the variable partitioning problem in an effective way. We first give two intuitive ways to build the initial VIG graph.

Construction of VIG-1: Given DFG $G = \langle U, ED, T, E \rangle$, if there exists a pair of memory operation nodes u and v that are independent pairs in G , then there is an edge (u, v) in the VIG.

Time	ALU1	ALU2	M1	M2
1			A	
2			A	
3	OP1			
4			B	C
5			B	C
6	OP2	OP3		
7	OP4			
8			D	
9			D	

Time	ALU1	ALU2	M1	M2
1			A	
2			A	
3	OP1			
4			B	
5			B	
6	OP2		C	
7			C	
8	OP3			
9	OP4			
10			D	
11			D	

Fig. 3. (a) Schedule 1 with B and C in different banks. (b) Schedule 2 with B and C in the same bank.

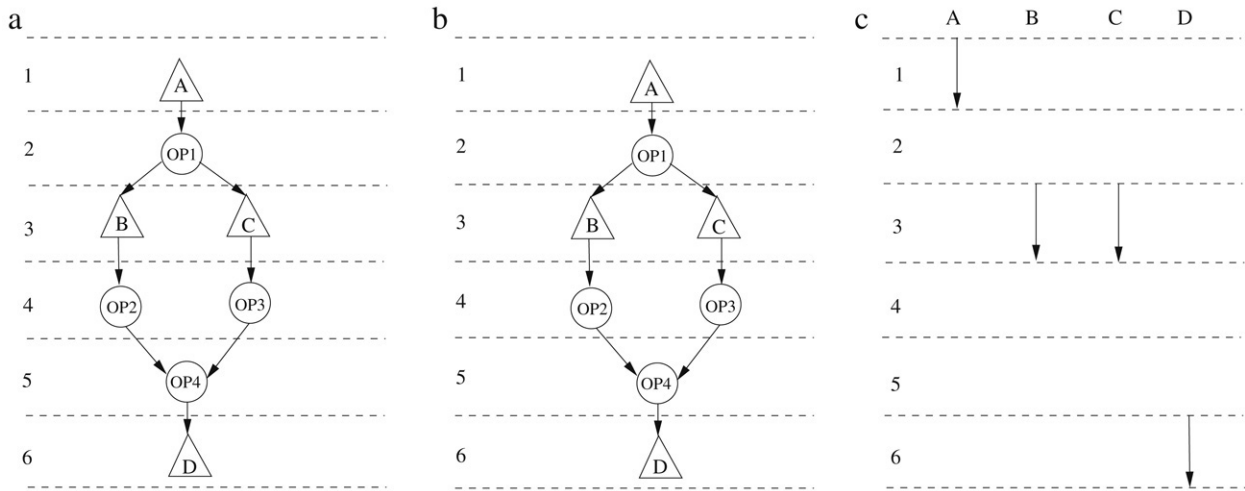


Fig. 4. (a) ASAP schedule. (b) ALAP schedule. (c) Memory operation mobility graph.

Construction VIG-2: Given DFG $G = \langle U, ED, T, E \rangle$, if there exists a pair of memory nodes u and v that are independent pairs in G , and $MW(u) \cap MW(v) \neq \emptyset$, then there is an edge (u, v) in the VIG.

The construction of the VIG graph is based on the DFG representation of an application. From the DFG representation of a program, one can readily derive both the ASAP and ALAP schedules, considering the constraints of computation units. Let the control steps of a memory operation, a , be $t_s(a)$ and $t_l(a)$ according to ASAP and ALAP, respectively. The mobility, that is, the scheduling freedom of a , defined as $[t_s, t_l]$, represents the time interval in which a can be scheduled without introducing additional delay. Only when the mobilities of two memory operations have some overlap may parallelizing the two corresponding variables be beneficial, in terms of improving performance. Clearly, the larger the overlap between two mobilities, the higher the potential of the two variables being able to be parallelized. If the mobilities of two operations are both small and their overlap is relatively large, parallelizing the corresponding variables is more likely to improve the schedule length. In other words, if such variables are put in the same bank, accessing the two variables is forced to be sequentialized, which is very likely to increase the overall schedule length. Zhuge et al. [42] assigned a possibility weight defined below to an edge to model this property.

We compute the possibility weight as follows: given two memory operations, a and b , let their mobilities be $[t_s(a), t_l(a)]$ and $[t_s(b), t_l(b)]$, and the maximum overlap between these two mobilities be the interval $[t_1, t_2]$, the possibility weight assigned to

the edge between the two variables accessed in operations a and b is $\frac{t_2 - t_1 + 1}{(t_l(a) - t_s(a) + 1)(t_l(b) - t_s(b) + 1)}$.

For example, in Fig. 4(a), for the DFG in Fig. 1(a), we compute the mobility of each variable by using the ASAP and ALAP schedules of the DFG. Then we depict the ALSP schedule in Fig. 4(b). After computing the mobility of each variable, we draw the mobility graph in Fig. 4(c). For instance, variables B and C have $MW(B) = [3, 3]$ and $MW(C) = [3, 3]$. B and C are independent pairs in VIG, using construction of VIG-1. Base on construction of VIG-2 and the possibility weight computation formula shown above, we get: the weight of the pair (B, C) is 1 and there is an edge (B, C) in VIG graph. There is no edge between the two nodes of pairs (A, B), (A, C), (D, B), and (D, C).

2.4. Heterogeneous multi-bank type assignment problem

An assignment A is a function from domain U to range R , where U is the node set and R is the type set. For a node $u \in U$, $A(u)$ provides the selected mode of node u . In a DFG $G, T_{R_j}(u), 1 \leq j \leq M$, represents the execution times of each node $u \in U$ when running with type R_j ; For each type R_j with respect to node u , there is a set of E_i , which is the energy consumption of each node in DFG. $E_{R_j}(u), 1 \leq j \leq M$, is used to represent the energy consumption of each node $u \in U$ on mode $R_j, E_{R_j}(u) = \sum E_i$, which is a fixed value. Given an assignment A of a DFG G , we define the system total energy consumption under assignment A , denoted as $E_A(G)$, to be the

Table 1
The results of *Type_Assign* for the DFG in Fig. 1(a).

Time	9	10	11	12	14	16	18
Energy	42	35	28	14	12	10	6

summation of energy consumption, $E_{A(u)}(u)$, $u \in U$, of all nodes, that is, $E_A(G) = \sum_{u \in U} E_{A(u)}(u)$. In this paper, we call $E_A(G)$ *total energy consumption* in brief.

Define the (*Heterogeneous Multi-Bank Type Assignment*) problem as follows: Given M different types: R_1, R_2, \dots, R_M , a DFG $G = \langle U, ED, T, E \rangle$ with $T_{R_j}(u)$ and $E_{R_j}(u)$ for each node $u \in U$ executed on each type R_j , a timing constraint L , find a type assignment A using only K types to give the minimum energy consumption E under timing constraint L .

3. Motivational example

In this section, we continue the example in Fig. 1(a) and give the final solution by using the proposed algorithm.

Based on precedence relations in Fig. 1(a) and the variable partition information, we know that B and C should be placed in different banks. Then, we perform type assignment to minimize total energy consumption under different timing constraints L . For instance, under the timing constraint of 11 cycles, the type assignment for memory operations is: A, B in one bank with type 1(2, 10) and C, D in another bank with type 2(3, 3). The total time required for the memory part is 8 cycles, and the total energy consumption is 26 nJ. Adding up the ALU part, which requires a total time of 3 cycles and energy 2 nJ, the total time is 11 cycles and the total energy is 28 nJ. The detail schedule is shown in Fig. 5(a). For the homogeneous situation (i.e. when only one memory type is allowed) we can choose type 1(2, 10), since the total time cannot satisfy the timing constraint of 11 cycles with type 2 or 3. The detail schedule with homogeneous memory (type 1(2, 10)) is shown in Fig. 5 (b). The total energy is 42 nJ. Compared with the heterogeneous memory type solution 28 nJ, the energy saving is 33.3%.

For the example shown in Fig. 1, we obtained different minimum energy consumptions while satisfying different timing constraints by using our algorithm. The results are shown in Table 1. In this example, the detail of inputs for Table 1 is described in Section 2.1. Only 2 banks are available, i.e., can be accessed at the same time. There are 3 types of banks to choose from and we can use only a maximum of 2 bank types. The memory types are shown in Fig. 1(b). In Table 1, “Time” represents total time spent and “Energy” represents total energy consumption of the DFG. There is only a total of seven solutions under different timing constraints.

4. The algorithms

In this section, an algorithm, TASL (*Type Assignment and Scheduling for Loops*), is designed to solve the problem of minimizing total energy without sacrificing performance. We need to overcome several challenges. First, for VIG design, we want to keep a certain level of accuracy of the parallelism and still have a graph working effectively for the variable partition problem. In order to tackle the problem, we build a VIG with both serial and parallel variable partition weights for each node. Second, after obtaining the serial and parallel variable partition weights for each node, we need to find an effective way to place all nodes into different memory banks. This problem is addressed by the algorithm $TASL_\delta$, which is an empirical method, to decide the nodes that need to be placed into different banks. Third, we need to fully exploit the parallelism of the memory architecture. For that purpose, a novel algorithm, *Type_Assign*, is proposed to assign

suitable bank types to different nodes in order to minimize the total energy consumption while satisfying timing constraints. Fourth, we need to compute the configuration of the bank types and minimize required resources. We schedule memory and ALU by using *Minimum Resource Scheduling and Configuration* algorithm. Finally, to further improve the result obtained from all previous steps, we use loop scheduling to improve the scheduling and assignment by rescheduling nodes repeatedly.

4.1. The TASL algorithm

The TASL algorithm is shown in Fig. 6. In this algorithm, we first put all nodes in the first row of S into set U_r . Then we delete the first row of S and shift S up by one control step. After that, we retime each node $u \in U_r$ such that $r(u) \leftarrow r(u) + 1$. Then based on the precedence relation in the retimed graph G_r , we rotate each node $u \in U_r$ by putting u into the earliest location. After all nodes in U_r are scheduled, we build the scheduling graph. Based on Fig. 1(a), we obtained the variable partition weights by building VIG graph. Then we put nodes into different memory banks by algorithm $TASL_\delta$, which will be described in the following section. Next, *Type_Assign* algorithm is used to find the type assignments with at most K types of memory and P types of ALU while satisfying timing constraint L . Finally, we do memory and ALU scheduling using *Minimum Resource Scheduling and Configuration* algorithm. The outputs include minimum energy consumption, corresponding assignment, and minimum resource scheduling.

TASL algorithm has combined several novel techniques to explore the heterogeneous type memory bank and ALU: First, we propose a novel *Type_Assign* algorithm, which use dynamic programming with the consideration of variable partition weights. Second, VIG graph has been built to obtain variable partition weights. Third, loop scheduling has been used to achieve the optimal results. Third, we consider heterogeneous type assignment and minimum resource scheduling and configuration for both memory and ALU.

4.2. The algorithm $TASL_\delta$

In each iteration of TASL algorithm, we first build VIG graph for variable partition and find both serial and parallel variable partition weights. Then, use dynamic programming *Type_Assign* to get assignments with at most K types of memory with the consideration of variable partition weight for memory part. Between these two steps, we need to decide which node in the DFG needs to consider variable partition weights. Assume there are totally N nodes in the DFG. There are $\delta * N$ nodes need to consider variable partition weights, where δ is a constant value decided by experiments through trials and errors. This means that δ is obtained by empirical study. For example, for one kind of test benchmark, we try 10 different δ values, compare the results of performance. Then select the δ value corresponding to the best performance. We do it several times and get the empirical value of δ .

Below we give the algorithm $TASL_\delta$ to decide the nodes that need to put into different banks, which is shown in Fig. 7. We use a combined weight function W to compute the overall weight. For example, $W(a_i) = \alpha * S_i + \beta * P_i$, where S_i is the serialism weight [33] of node a_i and P_i is the parallelism weight [42] of node a_i . Here, α and β are obtained by empirical study and they satisfy the constraint: $\alpha + \beta = 1$.

Time	ALU1 (1, 0.5)	ALU2 (1, 0.5)	M1 (2, 10)	M2 (3, 3)
1			A	
2			A	
3	OP1			
4			B	C
5			B	C
6	OP2			C
7		OP3		
8	OP4			
9				D
10				D
11				D

Time	ALU1 (1, 0.5)	ALU2 (1, 0.5)	M1 (2, 10)	M2 (2, 10)
1			A	
2			A	
3	OP1			
4			B	C
5			B	C
6	OP2	OP3		
7	OP4			
8				D
9				D
10				
11				

Fig. 5. (a) The best schedule of *Type_Assign* using two memory types with timing constraint 11. (b) The best schedule of using only one memory type with timing constraint 11.

Require: DFG $G = \langle U, ED, T, E \rangle$ with memory and ALU operations, N_1 types of memory banks, N_2 types of ALUs, each type has (energy E , latency T) attributes. K number of memory banks that can be accessed simultaneously, P numbers of the ALUs, the timing constraint L , an initial schedule S of G , rotation number R , the retiming r of G .

Ensure: A schedule S' , retiming r' and the type assignment A' for each bank to minimize energy E while satisfying L .

- 1: **for all** $i = 1$ to R **do**
- 2: $U_r \leftarrow$ All nodes in the first row in S ;
- 3: Delete the first row from S ;
- 4: Shift S up by 1 control step;
- 5: **for all** $u \in U_r$ **do**
- 6: $r(u) \leftarrow r(u) + 1$;
- 7: **end for**
- 8: **for all** $u \in U_r$ **do**
- 9: Put u into the earliest available location of u in S based on the precedence relation in G_r ;
- 10: **end for**
- 11: Build VIG graph for variable partition, find both parallelism [1] and serialism [21] weights;
- 12: Use $TASL_\delta$ to select the nodes that need to put into different Banks;
- 13: Use dynamic programming *Type_Assign* to get assignments with at most K types with the consideration of variable partition weight for memory part; And get assignments with at most P types for ALU part;
- 14: Do ALU and memory scheduling using *Minimum Resource Scheduling and Configuration Algorithm*;
- 15: $E_{min} \leftarrow$ the minimum total energy consumption;
- 16: **end for**
- 17: $E_{min} \leftarrow$ the minimum total energy consumption for all the iterations R ;
- 18: $S', r', A' \leftarrow$ the schedule, retime, and assignment corresponding to E_{min} ;
- 19: Output S', r', A' ;

Fig. 6. *TASL* Algorithm.

Require: DFG $G = \langle U, ED, T, E \rangle$ with N nodes, M types of memory banks with (T, E) pairs. K number of memory banks that can be accessed simultaneously

Ensure: The nodes that need to put into different memory banks

- 1: Compute the combined variable partition weight $W(a_i)$, where W is the combined weight function;
- 2: Sort the nodes according to the descending of combined variable partition weight;
- 3: We obtain a sequence $Q: W(a_1) > W(a_2) > \dots > W(a_N)$
- 4: Let $W(a_i) \geq \delta * N \geq W(a_{i+1})$;
- 5: Fetch the first i nodes in the sequence Q ;
- 6: These i nodes need to put into different banks;

Fig. 7. Algorithm *TASL_δ* to decide the nodes that need to put into different banks.

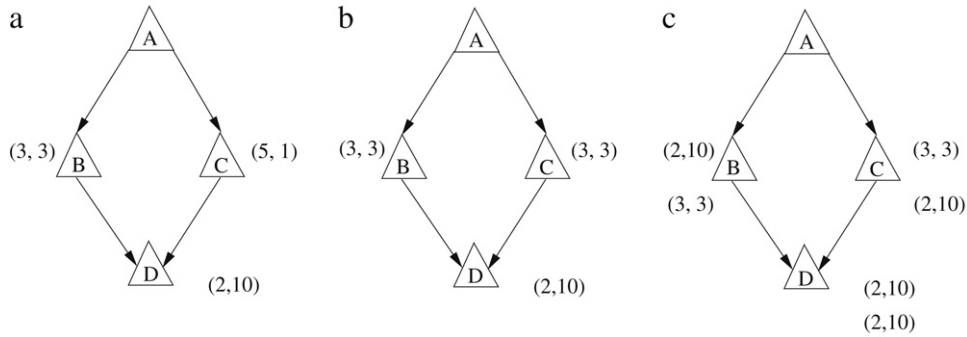


Fig. 8. (a) Example of Rule 1. (b) Example of Rule 2. (c) Example of Rule 3.

4.3. Definitions and lemma of Type_Assign

To solve the type assignment problem for a certain schedule S , we use dynamic programming method traveling the graph in a bottom up or top down fashion. For the ease of explanation, we will index the nodes based on bottom up sequence.

We first give an example to illustrate our approach. Fig. 8 shows a DFG with four nodes, A, B, C and D. The bank types are identical to the bank types shown in Fig. 1(b). Only 2 banks are available, i.e., can be accessed at the same time. There are 3 types of banks to choose from and we can use only a maximum of 2 types of banks. Using the bottom up approach, we start from node D, then move to B and C, and finally we arrive at node A.

If we get a sequence $(2, 10)(3, 3)(5, 1)$ for nodes D, B, and C, we will discard this sequence. In this scenario, three memory types are required, but there are only two memory types available. We summarize this scenario (which is shown in Fig. 8(a)) into a cancellation rule, i.e., Rule 1, which will be shown in later part.

Based on the variable partition, we know that B and C should be assigned to different banks. If B and C are assigned to same bank type i , since B and C must be in different banks, two banks have been used. Then node D must also be in type i because only 2 banks are available. Therefore, B and C are not allowed to be in the same type i unless D is also in the type i . For instance, if we get a sequence $(2, 10)(3, 3)(3, 3)$, it will be deleted. On the other hand, we will keep the sequence $(3, 3)(3, 3)(3, 3)$. The scenario is shown in Fig. 8(b) and we summarize it into Rule 2.

In Fig. 8(c), in order to arrive at node A, we can go through D, B, C, A or D, C, B, A. If we get two sequences $(2, 10)(3, 3)(2, 10)$ and $(2, 10)(2, 10)(3, 3)$, we only need to keep one sequence. Since they are symmetric and there is no difference for the final solution. We summarize this scenario as symmetric rule, i.e., rule 3, which will be shown in later part.

Then we formalize our approach. Given the timing constraint L , a DFG G , and an assignment A , we give several definitions as follows:

- (1) The function from domain of variable to range of bank type is defined as $\text{Bank}()$. For example, “ $\text{Bank}(A) = \text{type } 1$ ” means the bank type of variable A is type 1.
- (2) G^i : The sub-graph rooted at node u_i , containing all the nodes reached by node u_i . In our algorithm, each step will add one node which becomes the root of its sub-graph.
- (3) $E_A(G^i)$ and $T_A(G^i)$: The total energy consumption and total execution time of G^i under the assignment A . In our algorithm, each step will achieve the minimum total energy consumption of G^i under various timing constraints.
- (4) In our algorithm, table $D_{i,j}$ will be built. Here, i represents a node number, and j represents a timing constraint. Each entry of table $D_{i,j}$ will store energy consumption $E_{i,j}$ and its corresponding linked list. Here we define $E_{i,j}$ as follows: $E_{i,j}$ is

the minimum energy consumption of $E_A(G^i)$ computed by all assignments A satisfying $T_A(G^i) \leq j$. The linked list records the type selection of all previous nodes passed, from which we can trace back how $E_{i,j}$ is obtained.

We have the following lemma about energy consumption cancellation with the same timing constraint.

Lemma 4.1. Given $E_{i,j}^1$ and $E_{i,j}^2$ with the same timing constraint, if $E_{i,j}^1 \geq E_{i,j}^2$, then $E_{i,j}^2$ will be kept.

In each step of dynamic programming, we have several rules about cancellation of redundant energy consumption and its corresponding linked list.

- (1) Rule 1: If the number of memory types greater than K or the number of ALU types greater than P , then discard the corresponding energy consumption and linked list.
- (2) Rule 2: If two siblings a and b , i.e., children of same node, are not allowed to be same type, i.e., $\text{Bank}(a) \neq \text{Bank}(b)$, in variable partition, and $\text{Bank}(a) = \text{Bank}(b)$ in assignment, then discard the corresponding E and linked list, except the scenario that all the nodes till now are in the same type, i.e., $\forall u, v \in G, \text{Bank}(u) = \text{Bank}(v)$.
- (3) Rule 3: If two siblings a and b are just exchanged their types and other nodes are same in type assignment for the two corresponding linked lists, i.e., $\text{Bank}(a) = \text{Bank}(b)$ then only keep one E and its corresponding linked list.

In every step of our algorithm, one more node will be included for consideration. The data of this node is stored in local table $B_{i,j}$, which is similar to table $D_{i,j}$, but with energy consumption only on node u_i . A local table stores only data of energy consumption of a node itself. Table $B_{i,j}$ is the local table only storing the energy consumption of node u_i . $E_{i,j}$ is the energy consumption only for node u_i with timing constraint j . The algorithm to compute $D_{i,j}$ is shown in Fig. 9.

4.4. The Type_Assign algorithm

Algorithm *Type_Assign* is shown in Fig. 10. Without loss of generality, we assume that bottom up approach is used. Algorithm *Type_Assign* gives the near-optimal solution when the given DFG is a DAG. In step 6, $D_{i_1,j} + D_{i_2,j}$ is computed as follows. Let G' be the union of all nodes in the graphs rooted at nodes u_{i_1} and u_{i_2} . Travel all the graphs rooted at nodes u_{i_1} and u_{i_2} . If a node q in G' appears for the first time, we add the energy consumption of q to $D'_{i,j}$. If q appears more than once, that is, q is a common node, we only count it once. That is, the energy consumption is just added once. The final $D_{N,j}$ we get is the table in which each entry has the minimum energy consumption under the timing constraint j .

In the following, we give the Theorem 4.1. about this.

Require: A simple path DFG
Ensure: $D_{i,j}$

- 1: Build a local table $B_{i,j}$ for each node;
- 2: Start from u_1 , $D_{1,j} \leftarrow B_{i,j}$;
- 3: **for all** $u_i, i > 1$, **do**
- 4: **for all** timing constraint j , **do**
- 5: Compute the entry $D_{i,j}$ as follows:
- 6: **for all** k in $B_{i,j}$, **do**
- 7: $D_{i,j} = D_{i-1,j-k} + B_{i,k}$;
- 8: Cancel redundant energy and linked list with the three redundant linked list cancellation rules;
- 9: Insert $D_{i,j-1}$ to $D_{i,j}$ and remove redundant energy value according to Lemma 4.1;
- 10: **end for**
- 11: **end for**
- 12: **end for**

Fig. 9. Algorithm to compute $D_{i,j}$ for a simple path.

Input: DFG $G = \langle U, ED, T, E \rangle$ with N nodes, M types of memory banks with (T, E) pairs. K number of memory banks that can be accessed simultaneously, P numbers of the ALUs, the timing constraint L .
Output: An efficient types assignment to $\text{MIN}(E)$ while satisfying L .
Algorithm:

- (1) $SEQ \leftarrow$ Sequence obtained by topological sorting all the nodes.
- (2) $t_{mp} \leftarrow$ the number of multi-parent nodes; $t_{mc} \leftarrow$ the number of multi-child nodes;
If $t_{mp} < t_{mc}$, **Then** use bottom up approach;
Else, use top down approach.
- (3) **If** bottom up approach, **Then** use the following algorithm;
If top down approach, **Then** just reverse the sequence.
- (4) $SEQ \leftarrow \{u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_N\}$, in bottom up fashion;
 $D_{1,j} \leftarrow B_{1,j}$;
 $D'_{i,j} \leftarrow$ the table that stored $\text{MIN}(E)$ for the sub-graph rooted on u_i except u_i ;
 $u_{i_1}, u_{i_2}, \dots, u_{i_w} \leftarrow$ all child nodes of node u_i ; $w \leftarrow$ the number of child nodes of node u_i .
- (5) **If** $w = 0$, **Then** $D'_{i,j} = (0, 0)$;
If $w = 1$, **Then** $D'_{i,j} = D_{i_1,j}$;
If $w > 1$, **Then** $D'_{i,j} = D_{i_1,j} + \dots + D_{i_w,j}$;
- (6) Computing $D_{i_1,j} + D_{i_2,j}$:
 $G' \leftarrow$ the union of all nodes in the graphs rooted at nodes u_{i_1} and u_{i_2} ;
Travel all the graphs rooted at nodes u_{i_1} and u_{i_2} ;
If a node is a common node, **Then** use a selection function to choose the type of a node.
Cancel redundant energy and linked list with the three redundant linked list cancellation rules.
- (7) For each k in $B_{i,k}$,
 $D_{i,j} = D'_{i,j-k} + B_{i,k}$
- (8) $D_{N,j} \leftarrow$ a table of $\text{MIN}(E)$;
Output $D_{N,L}$.

Fig. 10. Type_Assign Algorithm.

Theorem 4.1. For each $E_{i,j}$ in $D_{i,j}$ ($1 \leq i \leq N$) obtained by algorithm Type_Assign, $E_{i,j}$ is the minimum total cost for the graph G^i under timing constraint j .

Proof. By induction. *Basic Step:* When $i = 1$, there is only one node and $D_{1,j} = B_{1,j}$. Thus, when $i = 1$, Theorem 4.1 is true. *Induction Step:* We need to show that for $i \geq 1$, if for each $E_{i,j}$ in $D_{i,j}$, $E_{i,j}$ is the minimum total energy consumption of the graph G^i , then for each $E_{i+1,j}$ in $D_{i+1,j}$, $E_{i+1,j}$ is the total energy consumption of the graph G^{i+1} under timing constraint j . According to the bottom

up approach (for top down approach, just reverse the sequence), the execution of $D_{i,j}$ for each child node of v_{i+1} has been finished before executing $D_{i+1,j}$. From step 5, $D'_{i+1,j}$ gets the summation of the minimum total energy consumption of all child nodes of u_{i+1} because they can be executed simultaneously within time j . We avoid the repeat counting of the common nodes. Hence, each nodes in the graph rooted by node u_{i+1} was counted only once. From step 7, the minimum total energy consumption is selected from all possible energy consumption caused by adding u_{i+1} into the sub-graph rooted on u_{i+1} . So for each $E_{i+1,j}$ in $D_{i+1,j}$, $E_{i+1,j}$ is the total

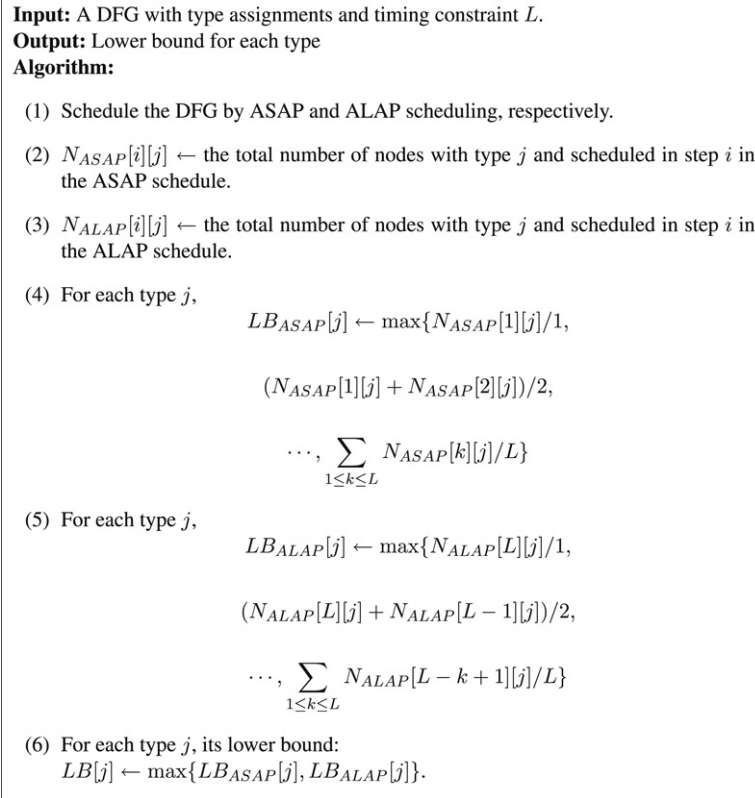


Fig. 11. Algorithm Lower_Bound_RC.

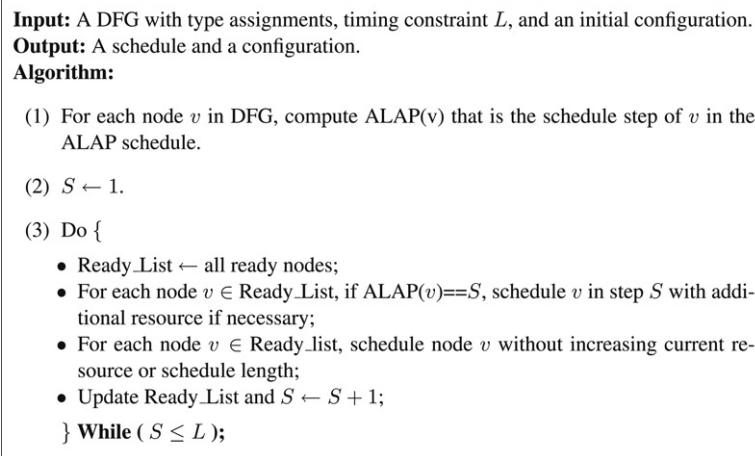


Fig. 12. Algorithm Min_RC_Scheduling.

energy consumption of the graph G^{i+1} under timing constraint j . Therefore, Theorem 4.1 is true for any i ($1 \leq i \leq N$). \square

4.5. The minimum resource scheduling and configuration

We have obtained type assignment with at most K types of memory banks and P types of ALU. Then we will compute the configuration of the types and give the corresponding schedule which consume minimum energy while satisfying timing constraint. We propose minimum resource scheduling algorithms to generate a schedule and a configuration which satisfies our requirements. We first propose *Algorithm Lower_Bound_RC* that produces an initial configuration with low bound resource. Then we propose *Algorithm Min_RC_Scheduling* that refine the initial configuration and generate a schedule to satisfy the timing constraint.

Algorithm Lower_Bound_RC is shown in Fig. 11. In the algorithm, it counts the total number of every type in every time unit in the ASAP (As Soon As Possible) and ALAP (As Late As Possible) schedule, respectively. Then the lower bound for each bank type is obtained by the maximum value that is selected from the average resource needed in each time period. For example, for the Fig. 1(a), after using *Lower_Bound_RC*, we find that the lower bound of ALU is 1, and the lower bound of memory is also 1.

Using the lower bound of each type as an initial configuration, we propose an algorithm, *Algorithm Min_RC_Scheduling*, which is shown in Fig. 12, to generate a schedule that satisfies the timing constraint and get the final configuration. In the algorithm, we first compute $ALAP(v)$ for each node v , where $ALAP(v)$ is the schedule step of v in the ALAP schedule. Then we use a revised list scheduling to perform scheduling. In each scheduling step, we first schedule all nodes that have reached the deadline with additional resource if

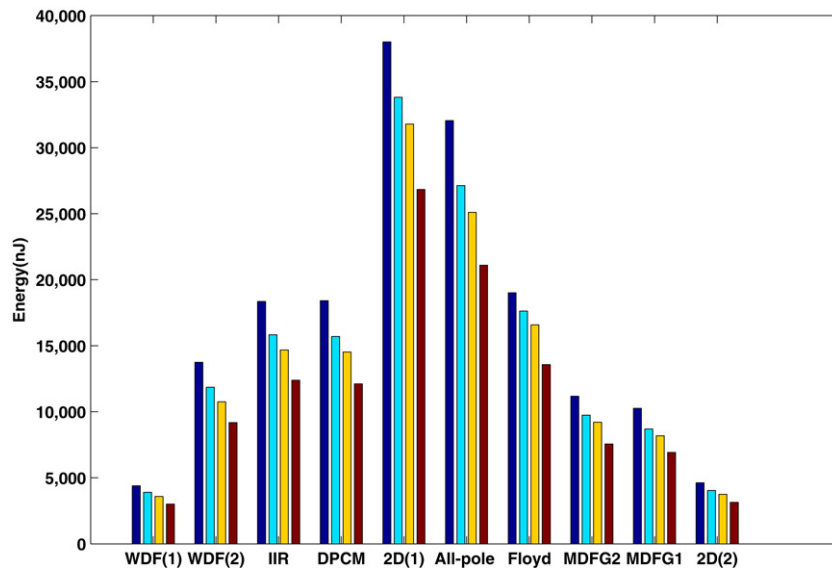


Fig. 13. The comparison of total energy consumption with Method 1, Method 2, VPIS, and TASL while satisfying timing constraint $L = 2000$ for various benchmarks.

necessary and then schedule all other nodes – as many as possible – without increasing resource. For example, for the Fig. 1(a), after using Min_RC_Scheduling, we find that only one ALU and two memory (one is type 1, the other is type 2) are needed.

Algorithms *Lower_Bound_RC* and *Min_RC_Scheduling* both take $O(|U| + |ED|)$ to get results, where $|U|$ is the number of nodes and $|ED|$ is the number of edges for a given DFG. The complexity of algorithm TASL is $O(R * |V| * L * M)$, where R is the rotation time and we set it as $10 * |V|$. $|V|$ is the number of nodes, L is the given timing constraint, M is the number of bank types. When L equals $O(|V|^c)$ (c is a constant) which is the general case in practice, algorithm TASL is polynomial.

5. Experiments

5.1. Experimental setup

In our experiments, we implement our TASL algorithm in the SPAM compiler environment [29] to replace the simulated annealing algorithm [30] originally used by (<http://www.idiom.com/free-compilers/TOOL/SPAMComp-1.html>) the Princeton project. We conduct experiments with our algorithm on a set of benchmarks including Wave Digital filter (WDF), Infinite Impulse filter (IIR), Differential Pulse-Code Modulation device (DPCM), Two dimensional filter (2D), Floyd-Steinberg algorithm (Floyd), and All-pole filter. The proposed runtime system has been implemented and a simulation framework to evaluate its effectiveness has been built. M different memory bank types, B_1, \dots, B_M , are used in the system, in which type B_1 is the quickest with the highest energy consumption and type B_M is the slowest with the lowest energy consumption. We set the energy consumption of memory access operation as 2.20 nJ [16,28], and the ALU energy consumption as 0.65 nJ based on 180 nm process. The leakage energy is negligible due to the small size of memory and ALU operation circuits [22,33]. The memory size of each memory bank is 32 KB.

We conduct experiments on four methods: Method 1: uniform type + list scheduling; Method 2: exhaustive type assignment + list scheduling; Method 3: the VPIS in [33]; Method 4: our TASL algorithm. In Method 1, there is only one memory bank type, the type is the one with minimum energy consumption, i.e., type B_M . Based on homogeneous type, we do list scheduling. In Method 2, there are heterogeneous types. First we fix the type of each memory, then do list scheduling. After that, we assign other

type to each memory and repeat again until we exhaust all type assignment. Then find the type assignment with minimum energy consumption. In the list scheduling, the priority of a node is set as the longest path from this node to a leaf node [23].

In the experiments, we unfold all the benchmarks 20 times, which means the number of nodes increases 20 times. The largest benchmark 2D(1) originally has only 34 nodes. After unfolding 20 times, the benchmark has 680 nodes. The number of nodes of the benchmarks is as the following: 2D(1) [680], WDF(1) [80], WDF(2) [240], IIR [320], DPCM [320], All-pole [380], Floyd [320], MDFG2 [160], MDFG1 [160], 2D(2) [80]. We set the rotation times as $10 * |V|$, where $|V|$ is the number of nodes in the DFG.

5.2. Experimental results

The experimental results for the four methods are shown in Figs. 13–15. In Fig. 15, the number of ALU is 5 and there are 5 memory types and 4 memory banks. The X-axis represents the name of each benchmark. The Y-axis represents the energy consumption “Energy(nJ)”. For each benchmark, we have four data, which represents the the minimum total energy consumption obtained from three different scheduling algorithms: Method 1 (Field “Method 1”), Method 2 (Field “Method 2”), VPIS [33] (Field “VPIS”), and our TASL algorithm (Field “TASL”).

The results show that our algorithm TASL can significantly improve the performance of DSP processors. We can see that with more type-selections, the reduction ratio for the total energy consumption has increased. For example, with 3 types of both memory and ALU, compared with VPIS [33], TASL shows an average 16.2% reduction in total energy consumption. Using 5 types of ALU and memory, the reduction rate changed to be 18.8% for total energy consumption. It is worthwhile pointing out that we obtain this improvement ratio without increasing the code size of applications. Experimental results show that the performance of our algorithm is improved with the increment of number of cores since the degree of parallel has been increased. The experimental results show that when the number of processors increases, the percentage of reduction on total energy increases correspondingly.

The reasons why our algorithm is better than Method 2 are as follows. First, Method 2 does not consider variable partition under each fixed type assignment. Second, minimum resource scheduling and configuration algorithm has been used to improve the final performance. TASL also achieved significantly better results than

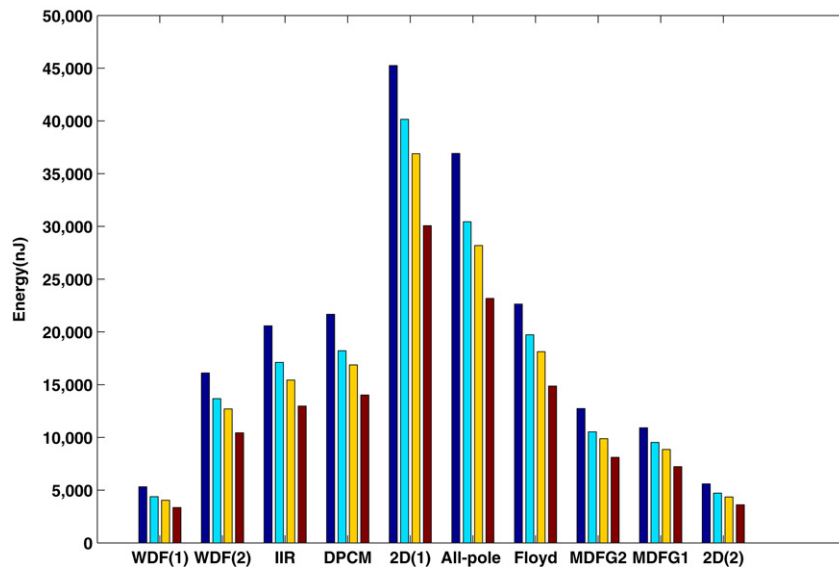


Fig. 14. The comparison of total energy consumption with Method 1, Method 2, VPIS, and TASL while satisfying timing constraint $L = 3000$ for various benchmarks.

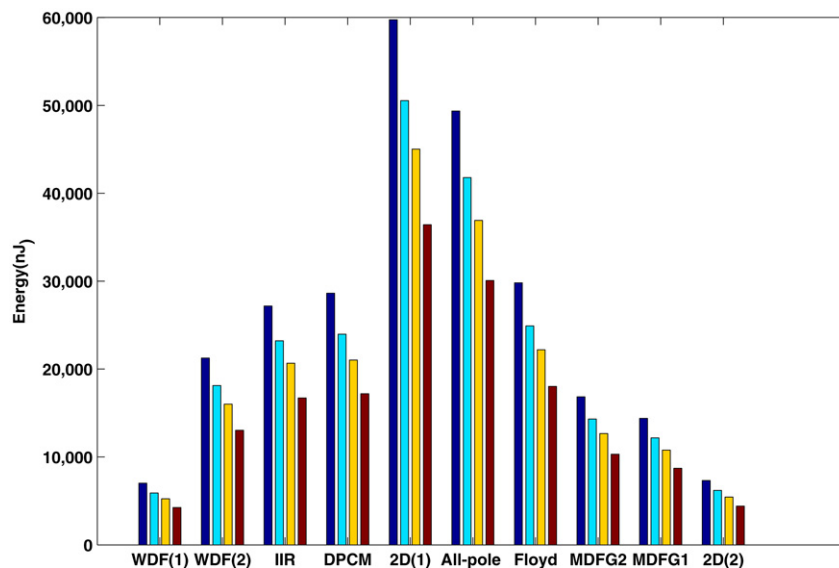


Fig. 15. The comparison of total energy consumption with Method 1, Method 2, VPIS, and TASL while satisfying timing constraint $L = 4000$ for various benchmarks.

that of VPIS [33], because it fully exploits the heterogeneous multi-bank memory architecture and uses loop scheduling to further optimize the performance and energy saving.

The results show that the rotation times to generate the best schedule is less than $1 \cdot |V|$ and close to the times when all nodes have been rotated one time. The complexity of algorithm TASL is $O(R \cdot |V| \cdot L \cdot M)$, where R is the rotation time and we set it as $10 \cdot |V|$. L is the given timing constraint, M is the number of bank types. In the experiments, the running time of TASL on each benchmark is less than 30 min. We compare TASL with the algorithm *DFG_Assign_CP* in Shao et al.'s paper [27], which always pick the node with the minimal ratio (between increased cost and reduced time) among all nodes with all possible unmarked types in the critical path. The experimental results show that our scheduling time is similar to *DFG_Assign_CP* but the result is significantly better than it. Algorithm *DFG_Assign_CP* needs about 20 min in average, but the performance of it is 25.6% lower than that of our approach in average.

In conclusion, our algorithm has several main pros: First, we study the combined effects of energy saving and performance of

memory and ALU in a systematic approach. Second, we exploit the energy saving with type assignment and minimum resource scheduling for both memory and ALU. Third, to the best of our knowledge, our paper is the first to consider the combined effect of both energy and performance with heterogeneous multi-bank memory. Fourth, we obtain the best results by rescheduling nodes repeatedly based on loop scheduling. Fifth, we improve the heterogeneous scheduling and assignment for both ALU and memory simultaneously.

6. Related work

Heterogeneous multi-bank memory: To improve the overall performance, many DSPs employ Harvard architecture, which provides simultaneous accesses to separate on-chip memory banks for instructions and data [12,1,31,13]. Some DSP processors are further equipped with multi-bank memory that are accessible in parallel, such as Analog Device ADSP2100, Motorola DSP56000, NEC uPd77016, and Gepard Core DSPs [12,13,19,33]. Harvesting the benefits provided by the multi-bank memory architecture

hinges on sufficient compiler support. Parallel operations afforded by multi-bank memory give rise to the problem of how to maximally utilize the instruction level parallelism. To the best of our knowledge, we are the first to consider the combined effect of both energy and performance with heterogeneous multi-bank memory model.

Variable Partition: Some previous work on the variable partitioning problem introduced in [19,26] tries to solve the variable partitioning problem on dual memory banks by using an interference graph. For most benchmarks used in our experiments, the variable partitioning results based on the interference graph model gives a longer schedule length. One of the limitations of the interference graph model is that it can only be applied to a directed acyclic graph (DAG), where parallelism across the loop body from different iterations is not explored. The second problem with the interference graph is that it does not incorporate sufficient information for a schedule to exploit the potential parallel memory accesses. Other dual-bank variable partitioning techniques in previous work are restricted to some specific architecture such as Motorola DSP processors [30].

Previous related work on operation parallelism can be roughly divided into two main categories: those that use compacted intermediate code as the starting point [26,20,8,30], and those that start with uncompact intermediate code [19,42]. *Compacted intermediate code* refers to the intermediate code that is compacted or scheduled by some heuristics such as list scheduling, to increase the instruction level parallelism without considering the data dependency. Since scheduling is done prior to exploring memory bank assignments, it is obvious that some memory-operation-pair combinations may be left out of consideration, no matter which heuristic is used to compact the code. Thus, the approaches in the first category often fail to exploit many optimization opportunities. Techniques in the second category overcome this problem by using the uncompact code to explore all possible pairs of memory operations as long as there are no dependencies between them. Therefore, we adopt the same philosophy as these techniques, that is, starting with the uncompact code.

Given a program represented by a data flow graph (DFG), an undirected graph can be constructed to model the relationship among the variables in the program. The nodes in the graph represent all the local variables stored in memory. Partitioning the nodes in the graph into different groups then leads to partitioning the corresponding variables to different memory banks.

The effectiveness of such an approach relies on modeling edge weights properly to capture all relevant information. A straightforward way of assigning edge weights is to connect two nodes with an edge of weight 1 if the two corresponding variables do not have data dependencies and the memory operations involving the variables can potentially overlap [19]. The reason is that accessing such two variables in parallel may decrease the schedule length. However, such potential parallelism may not always be realizable due to certain timing constraints on the associated memory operations.

Zhuge et al. [42] introduced the concept of possibility weight to capture the likelihood of parallelizing pairs of instructions. The model does improve on the simple graph model above, but it still has some deficiencies. One deficiency of the possibility weight model is associated with simple summation of the possibility weights mentioned earlier. Another problem with the possibility weight model is that it does not distinguish mobility overlaps within a single mobility range from those in different mobility ranges. Wang et al. [33] exploit serialism in instruction execution to trade off performance for energy savings. They use two lists to describe the edge weight in the graph model. By introducing one more dimension to the graph edge weight, they can capture

the serialism information among operations and overcome the deficiencies of previous models.

Heterogeneous scheduling and assignment: Heterogeneous assignment of special purpose architectures for real-time DSP applications has become a common and critical step in the design flow in order to satisfy the requirements of stringent timing constraint [6,15,37,5]. DSP applications need special high-speed functional units (FUs) like adders and multipliers to perform addition and multiplication operations [27]. Energy-saving task scheduling in multi-FU DSP systems has been mostly on homogeneous multiprocessors [7,2,41], few results considered heterogeneous systems in energy-saving real-time task scheduling [40,21,27]. Among the work for heterogeneous multi-FU DSP systems, Yu and Prasanna [40] considered the minimization of energy consumption for systems. The proposed algorithm is based on the *Integer Linear Programming* (ILP) without guarantees on the final solution. Luo and Jha [21] proposed list scheduling based heuristics for the scheduling of real-time tasks in heterogeneous distributed systems. However, little existing work for energy-saving scheduling in heterogeneous multi-bank memory provides guarantees on the energy consumption.

7. Conclusion

In this paper, we studied the scheduling and assignment problem that minimizes the total energy without sacrificing performance on heterogeneous multi-bank memory and multi-type ALU. We proposed a highly efficient algorithm, namely TASL (*Type Assignment and Scheduling for Loops*), to minimize energy consumption with heterogeneous multi-bank memory for applications with loops in particular. TASL achieved a significant energy-saving using a novel type assignment and scheduling approach with the consideration of variable partition, and rescheduling nodes repeatedly based on loop scheduling. A wide range of benchmarks have been tested and the experimental results showed that our algorithm significantly improved both the energy-saving and performance for applications on heterogeneous multi-bank memory.

Acknowledgments

This work is partially supported by NSF CCR-0309461, NSF IIS-0513669, HK CERG 526007 (HK PolyU B-Q06B), City U of HK [Project No. 7200106], and NSFC 60728206, 60811130528, 60725208, 60533040, 60504024 and 60874050; National High-Tech Research and Development Plan of China (863 Plan) under Grants 2008AA01Z106 and 2006AA01Z202, Shanghai Pujiang Plan No. 07pj14049; Zhejiang Provincial NSFC Y106010.

References

- [1] ADSP-21 000 Family Application Handbook, vol. 1, Analog Devices, Inc., Norwood, MA, 1994.
- [2] H. Aydin, R. Melhem, D. Mosse, P. Alvarez, Dynamic and aggressive scheduling techniques for power aware real-time systems, in: IEEE RTSS, London, UK, Dec. 2001.
- [3] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, Y. Robert, Scheduling strategies for master-slave tasking on heterogeneous processor platforms, IEEE Transactions on Parallel and Distributed Systems 15 (4) (2004) 319–330.
- [4] F. Catthoor, S. Wuytack, E.D. Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design, Kluwer Academic Publishers, 1998.
- [5] L.-F. Chao, A. LaPaugh, E.H.-M. Sha, Rotation scheduling: A loop pipelining algorithm, IEEE Transactions on Computer-Aided Design 16 (1997) 229–239.
- [6] L.-F. Chao, E.H.-M. Sha, Static scheduling for synthesis of DSP algorithms on various models, Journal of VLSI Signal Processing 10 (1995) 207–223.
- [7] J.-J. Chen, T.-W. Kuo, Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics, in: IEEE ICPP, Oslo, Norway, Jun. 2005.
- [8] J. Cho, Y. Paek, D. Whalley, Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms, in: ACM Joint Conf. LCTES-SCOPES, Berlin, Germany, Jun. 2002, pp. 130–138.

- [9] V. Delaluz, M. Kandemir, I. Kolcu, Automatic data migration for reducing energy consumption in multi-bank memory systems, in: DAC, New Orleans, LA, 2002, pp. 213–218.
- [10] V. Delaluz, M. Kandemir, A. Sivasubramaniam, M.J. Irwin, Hardware and software techniques for controlling DRAM power modes, *IEEE Transactions on Computers* 50 (11) (2001).
- [11] A. Dogan, F. Özgüner, Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing, *IEEE Transactions on Parallel and Distributed Systems* 13 (2002) 308–323.
- [12] Dsp56000 24-bit digital signal processor family manual, Motorola, Schaumburg, IL, 1996.
- [13] GEPARD family of embedded software programmable DSP core, <http://asic.amsint.com/databooks/digital/gepard.htm>.
- [14] K. Ito, L. Lucke, K. Parhi, ILP-based cost-optimal DSP synthesis with module selection and data format conversion, *IEEE Transactions on VLSI Systems* 6 (1998) 582–594.
- [15] K. Ito, K. Parhi, Register minimization in cost-optimal synthesis of DSP architecture, in: Proc. of the IEEE VLSI Signal Processing Workshop, Sakai, Japan, Oct. 1995.
- [16] S. Kim, Reducing ALU and register file energy by dynamic zero detection, in: IEEE IPCCC, New Orleans, LA, 2007, pp. 365–371.
- [17] A.R. Lebeck, X. Fan, H. Zeng, C.S. Ellis, Power aware page allocation, in: ACM ASPLOS, Cambridge, MA, Nov. 2000.
- [18] C.E. Leiserson, J.B. Saxe, Retiming synchronous circuitry, *Algorithmica* 6 (1991) 5–35.
- [19] R. Leupers, D. Kotte, Variable partitioning for dual memory bank DSPs, in: IEEE ICASSP, Salt Lake City, May 2001, pp. 1121–1124.
- [20] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, P. Marwedel, Optimized address assignment for DSPs with SIMD memory accesses, in: IEEE/ACM ASP-DAC, Yokohama, Japan, Jan. 2001, pp. 415–420.
- [21] J. Luo, N. Jha, Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems, in: IEEE VLSID, Bangalore, India, Jan. 2002.
- [22] C.-G. Lyuh, T. Kim, Memory access scheduling and binding considering energy minimization in multi-bank memory systems, in: IEEE/ACM DAC, San Diego, CA, 2004, pp. 81–86.
- [23] G.D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
- [24] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, Instruction scheduling for low power, *Journal of VLSI Signal Processing* 37 (2004) 129–149.
- [25] M. Qiu, Z. Jia, C. Xue, Z. Shao, E.H.-M. Sha, Loop scheduling to minimize cost with data mining and prefetching for heterogeneous DSP, in: PDCS, Dallas, Nov. 2006.
- [26] M. Saghir, P. Chow, C. Lee, Exploiting dual datamemory banks in digital signal processors, in: ACM ASPLOS, Cambridge, MA, Oct. 1996, pp. 234–243.
- [27] Z. Shao, Q. Zhuge, C. Xue, E.H.-M. Sha, Efficient assignment and scheduling for heterogeneous DSP systems, *IEEE Transactions on Parallel and Distributed Systems* 16 (2005) 516–525.
- [28] Y. Shimazaki, R. Zlatanovici, B. Nikolic, A shared well dual-supply-voltage 64-bit ALU, *IEEE Journal of Solid-State Circuits* 39 (3) (2004).
- [29] A. Sudarsanam, Code generation libraries for retargetable compilation for embedded digital signal processors, Ph.D. Dissertation, Princeton Univ., Princeton, NJ, 1998.
- [30] A. Sudarsanam, S. Malik, Simultaneous reference allocation in code generation for dual data memory bank ASIPs, *ACM TODAES* 5 (2) (2000).
- [31] TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instruments, Inc., Dallas, TX, 2000.
- [32] V. Venkatachalam, M. Franz, Power reduction techniques for microprocessor systems, *ACM Computing Surveys (CSUR)* 37 (3) (2005) 195–237.
- [33] Z. Wang, X. Hu, Energy-aware variable partitioning and instruction scheduling for multibank memory architectures, *ACM TODAES* 10 (2) (2005) 369–388.
- [34] Z. Wang, M. Kirkpatrick, E.H.-M. Sha, Optimal two level partitioning and loop scheduling for hiding memory latency for DSP applications, in: IEEE/ACM DAC, Los Angeles, Jun. 2000, pp. 540–545.
- [35] Z. Wang, T.W. O’Neil, E.H.-M. Sha, Minimizing average schedule length under memory constraints by optimal partitioning and prefetching, *Journal of VLSI Signal Processing* 27 (2001) 215–233.
- [36] Z. Wang, T.W. O’Neil, E.H.-M. Sha, Minimizing average schedule length under memory constraints by optimal partitioning and prefetching, *Journal of VLSI Signal Processing* 27 (2001) 215–233.
- [37] C.-Y. Wang, K. Parhi, Resource constrained loop list scheduler for DSP algorithms, *Journal of VLSI Signal Processing* 11 (1995) 75–96.
- [38] S. Wuytack, F. Catthoor, G.D. Jong, H.D. Man, Minimizing the required memory bandwidth in VLSI system realizations, *IEEE Transactions on VLSI Systems* 7 (4) (1999).
- [39] W. Ye, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, The design and use of simplepower: A cycle-accurate energy estimation tool, in: IEEE/ACM DAC, Los Angeles, Jun. 2000, pp. 340–345.
- [40] Y. Yu, V.K. Prasanna, Power-aware resource allocation for independent tasks in heterogeneous real-time systems, in: IEEE ICPADS, Taiwan, Dec. 2002.
- [41] Y. Zhang, X. Hu, D.Z. Chen, Task scheduling and voltage selection for energy minimization, in: IEEE/ACM DAC, Anaheim, CA, Jun. 2002, pp. 183–188.
- [42] Q. Zhuge, E.H.-M. Sha, B. Xiao, C. Chantrapornchai, Efficient variable partitioning and scheduling for DSP processors with multiple memory modules, *IEEE Transactions on Signal Processing* 52 (4) (2004) 1090–1099.

Meikang Qiu received the B.E. and M.E. degrees from Shanghai Jiao Tong University, China. He received the M.S. and Ph.D. degrees of Computer Science from University of Texas at Dallas in 2003 and 2007, respectively. From August 2007, he has been an assistant professor of Electrical and Computer Engineering at University of New Orleans. He had worked at Chinese Helicopter R&D Institute, and IBM. He is an IEEE Senior member and has published 60 papers. He now serves as the Program Chair of IEEE EmbeddedCom’09. He has been on various chairs and TPC members for many international conferences, such as IEEE SEC’08 IEEE CSE’08, IEEE ESO’08, IEEE GlobeCom’08, and IEEE RTCSA’09. He received Air Force Summer Faculty Award 2009. His research interests include embedded systems, computer security, and wireless sensor networks.

Minyi Guo received his Ph.D. degree in computer science from University of Tsukuba, Japan. Before 2008, Dr. Guo had been a research scientist of NEC Corp., Japan and a professor at the School of Computer Science and Engineering, The University of Aizu, Japan. Currently Dr. Guo is a distinguished chair professor of Dept. of Computer Science and Engineering, Shanghai Jiao Tong University, China and an adjunct professor of University of Aizu. Dr. Guo has published more than 160 research papers in international journals and conferences. He has served as general chair, program committee or organizing committee chair for many international conferences. His research interests include parallel and distributed processing, parallelizing compilers, pervasive computing, embedded systems software optimization, and software engineering. He is a senior member of IEEE, and member of the ACM, IPSJ, CCF, and IEICE.

Meiqin Liu received the B.S. and Ph.D. degrees in control theory and control engineering from Central South University, Changsha, China, in 1994 and 1999, respectively. From 1999 to 2001, she was a postdoctoral research fellow in Huazhong University of Science and Technology, Wuhan, China. From 2008 to 2009, she was a visiting scholar in the University of New Orleans, New Orleans, LA, USA. She is now a professor of College of Electrical Engineering, Zhejiang University, Hangzhou, China. Her research fields are neural network, robust control, and information fusion.

Chun Jason Xue received B.S. degree in Computer Science and Engineering from University of Texas at Arlington in May 1997, and M.S. and Ph.D. degree in computer Science from University of Texas at Dallas, in Dec 2002 and May 2007, respectively. He is now an Assistant Professor in the Department of Computer Science at the City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware codesign for parallel systems and computer security.

Laurence T. Yang is a professor at Department of Computer Science of St Francis Xavier University, Canada. His research includes high performance computing and networking, embedded systems, ubiquitous/pervasive computing and intelligence. He has published around 300 papers in refereed journals, conference proceedings and book chapters in these areas. He has been involved in more than 100 conferences and workshops as a program/general/steering conference chair. In addition, he is the editor in chief of several international journals and few book series. He has been acting as an author/coauthor or an editor/coeditor of 25 books. He has won 5 Best Paper Awards and 1 Best Paper Nomination in 2007; as well as a Distinguished Achievement Award, 2005; Canada Foundation for Innovation Award, 2003.

Edwin H.-M. 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 an editor 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 at <http://www.utdallas.edu/~edsha>.