J. Parallel Distrib. Comput. 69 (2009) 546–558

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), ltyang@stfx.ca (L.T. Yang), edsha@utdallas.edu (E.H.-M. Sha).

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.02.005

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.

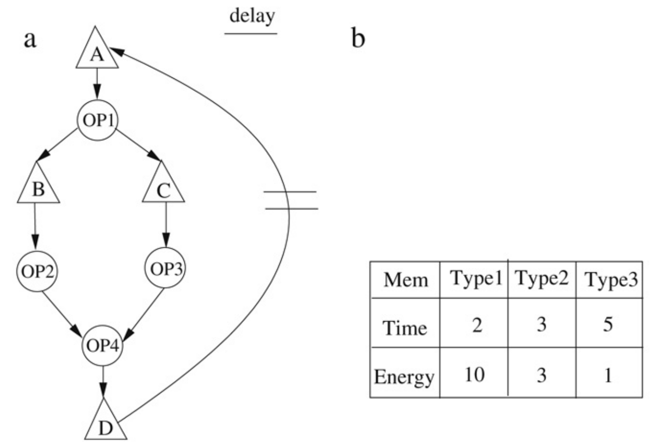
2. Basic concepts and models

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 $U = \langle u_1, \ldots, u_i, \ldots, u_N\rangle$ 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.



Mem	Type1	Type2	Type3
Time	2	3	5
Energy	10	3	1

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 chose 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

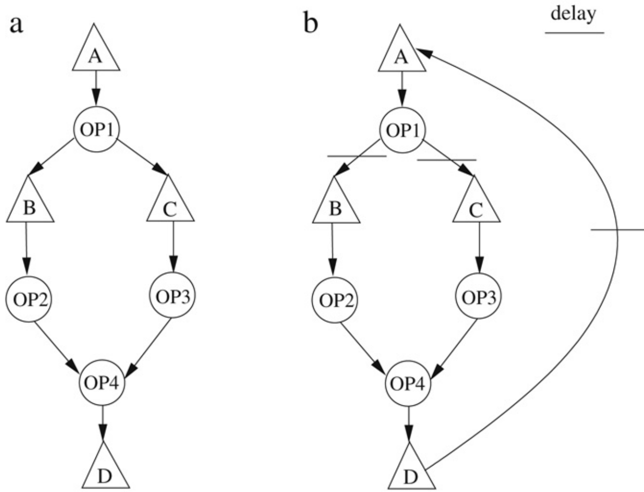


Fig. 2. (a) The static schedule of Fig. 1(a) by removing the edge with delays. (b) The rotated DFG.

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 $u_1 \to u_2 \in ED$, $u_1, u_2 \in U$.

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 $P_1, P_2, \ldots, P_n$, such that the total $w(u, v), \forall u \in P_i, \forall v \in P_j, \forall i, j = 1, \ldots, n$, 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.

a

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	

b

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.

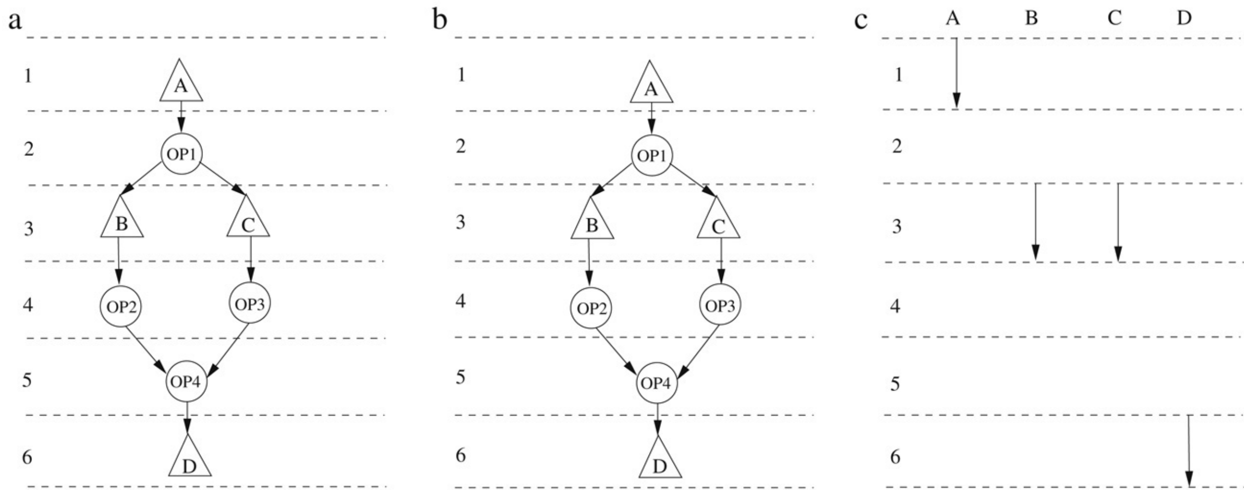


Fig. 4. (a) ASAP schedule. (b) ALAP schedule. (c) Memory operation mobility graph.

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: $R_1, R_2, \cdots, R_M$, 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 chose 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_* δ, 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_* δ, 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_δ

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_* δ 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: $\alpha + \beta = 1$.

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_δ* 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', and 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: $W(a_1) > W(a_2) > \cdots > W(a_N)$
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.