

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 of 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 [12]. The variable partitioning technique for multiple memory banks presented in [13] is aimed to find the mapping of array variables with minimal page misses. All the above techniques consider variable partitioning only as a separate phase before scheduling. Therefore, the quality of these techniques depends on the assumption of scheduling algorithm, such as list scheduling. However, some widely used optimization techniques, such as software pipelining [14], change the dependencies in a loop. As a result, the predefined partition may not work well in real cases.

CS	A0	A1	M0	MI
0	-	-	6	5
1	7	-	1	0
2	2	-	8	-
3	-	-	10	-
4	11	-	9	-
5	3	12	-	-
6	-	-	-	4

Partition 1 = { A, C }
Partition 2 = { B }

(a)

CS	A0	A1	M0	MI
0	-	-	5	6
1	7	-	0	1
2	2	-	8	10
3	11	-	9	-
4	3	12	-	-
5	-	-	4	-

Partition 1 = { A, B }
Partition 2 = { C }

(b)

Fig. 2. (a) Schedule with Partitions of A,C and B of Figure 1. (b) Schedule with Partitions of A,B and C of Figure 1.

In this paper, we address the variable partitioning and scheduling problem on multiple memory module architectures to maximize the benefit of this architectural feature. We first present a new graph model called Variable Independence Graph (VIG) for variable partitioning problem. Unlike the previous graph model, Interference Graph (IG) [10], [11], our graph model can be used to analyze a cyclic data flow graph. We then provides a variable partitioning algorithm that attempts to maximize the parallel memory accesses. Second, we develop a novel scheduling algorithm called Rotation Scheduling with Variable Repartitioning (RSVR), which considers both software-pipelining technique and variable partitioning simultaneously to generate compact schedules on multiple memory module architectures [14]–[17]. This algorithm adjust the variable partition during the software-pipelining process so that the parallel memory accesses can be exploited to produce a compact schedule. Finally, RSVR algorithm is used in our design space exploration approach to achieve a feasible schedule with the minimum number of functional units and memory modules. Our experimental results show the constant improvement on schedule length. The improvement by using RSVR based on VIG is up to 52.9% compared to the approach using interference graph model. The average improvement is

44.8%. Using our design exploration scheme, we can produce a better design solution with a fewer number of functional units and memory modules. Also, we can produce a feasible design solution in many cases while an approach using IG cannot.

This paper is organized as follows: Section II presents definitions and models used in this paper. In Section III, we present the constructions of the VIG. Section IV provides a variable partitioning algorithm based on the VIG. Then a new scheduling algorithm, Rotation Scheduling with Variable Repartitioning, is presented in Section V. In Section VI, a design exploration framework using our algorithm is proposed for finding the minimum memory module and functional unit configuration with schedule length requirement. The example and experimental results are displayed in Section VII. Finally, Section VIII concludes the paper.

II. DEFINITIONS AND GRAPH MODELS

In this section, we present graph models related to the variable partitioning problem. We also introduce important definitions used in our algorithms.

Before we formally present the graph model for variable partitioning problem, let us first introduce a Data Flow Graph model (DFG). Many multimedia and DSP applications can be represented by data flow graphs. In this paper, we use data flow graph as the description of a given program.

Definition 2.1: A *Data Flow Graph (DFG)* $G_D = \langle V, E, X, d, t \rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \subseteq V \times V$ is the edge set that defines the precedence relations over the nodes in V , $X(e)$ represents the variable accessed by an edge e , $d(e)$ represents the number of delays for an edge e , $t(v)$ represents the computation time of node v .

The nodes in the data flow graph can be either ALU operations or memory operations. As shown in Figure 1, 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, while the edge from an ALU operation node to a memory operation node represents a store. An edge label $X(e)$ indicates a variable that is accessed by a memory operation through edge e . Note that edges that do not involve a memory operation will not have the label. d is a function from E to a set of non-negative integers, representing the number of delays between two nodes.

Loops can be modeled by cyclic DFGs. An iteration is the execution of each node in V exactly once. Inter-iteration dependencies are represented by edges with delays. In particular, an edge $e(u \rightarrow v)$ with delay count $d(e) > 0$ means that the computation of node v at j^{th} iteration requires data produced by node u at $(j-d(e))^{\text{th}}$ iteration. The dependencies within the same iteration are represented by edges with zero delay. The delay count of a path is the summation of delay counts of all the edges along the path. Note that if we remove all edges with non-zero delays in a DFG, we will obtain a *directed acyclic graph (DAG)*. A static schedule must obey the precedence relations defined by the DAG part of a DFG.

For a cyclic data flow graph, the delay count of any cycle needs to be positive; otherwise, the corresponding program is illegal. Function t maps from V to a set of positive integers, representing the computation time of each operation node. In this paper, we assume that every node in V takes one time unit to execute.

Next, we define a novel graph model which is constructed from a DFG. The objective of using this model is to expose all parallel memory accesses in a DFG so as to properly partition variables. The nodes of the graph represent variables, and the edges in the graph represent potential parallelism existing among the memory accesses for these variables. We call this graph model, *Variable Independence Graph (VIG)*.

Definition 2.2: A *Variable Independence Graph (VIG)* is an undirected weighted graph $G_V = \langle V, E, w \rangle$ where V is a set of nodes representing variables, $E \subseteq V \times V$ is a set of edges connecting between nodes in V , whose memory operations can be executed in parallel potentially. Function w maps from E to a set of real values representing a priority of partitioning node u and v to different memory modules of an edge $(u, v) \in E$.

In the following, we introduce some important concepts that will be used in constructing a complete Variable Independence Graph. 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”.

Definition 2.3: Given DFG $G_D = \langle V, E, X, d, t \rangle$, if nodes $u, v \in V$ are not reachable from each other through any path without delay in G_D , nodes u and v are independent pairs.

For example, the nodes 0 and 1 in Figure 1 are independent pairs.

We define the Access Set as the set of memory operation nodes that access a particular variable.

Definition 2.4: Given DFG $G_D = \langle V, E, X, d, t \rangle$, the Access Set of variable X_i is $ACCESS(X_i) = \{v \mid \forall v, u, w \in V \ X(u, v) = X_i \text{ or } X(v, w) = X_i\}$.

Definition 2.5: Given DFG $G_D = \langle V, E, X, d, t \rangle$, the accessed variable of a memory operation node $v_i \in V$ is $ACCESS^{-1}(v_i) = X_i$, where $X_i \in X$ and $X(u, v_i) = X_i$ or $X(v_i, u) = X_i$, $\exists u \in V$.

For instance, in Figure 1, $ACCESS(A) = \{8\}$, $ACCESS(B) = \{0, 3, 5\}$, $ACCESS(C) = \{1, 6, 10\}$, and $ACCESS^{-1}(3) = B$.

The mobility property of a node in a schedule is very important in improving the preciseness in the graph construction, as we will discuss in details in the next section. Given a DFG $G_D = \langle V, E, X, d, t \rangle$, a Mobility Window [18] of node $v \in V$, denoted by $MW(v)$ in this paper, is a set of control steps in a static schedule that node v can be placed. The first control step the node v can be scheduled is determined by As Soon As Possible scheduling (ASAP), and the last control step that node v 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 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 control step. In Figure 1, the critical path consists of six nodes $5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 4$.

The mobility window of node 8 is $MW(8) = \{3\}$. The mobility window of node 10 is $MW(10) = \{2, 3, 4, 5\}$. Therefore, the parallel operations regarding nodes 8 and 10 can only occur once, that is in control step 3, among all four possible arrangements of these two nodes in the schedule.

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.

Definition 2.6: Given VIG $G_V = \langle V, E, w \rangle$, for $X_i, X_j \in V$ such that $(X_i, X_j) \in E$, the priority function of edge (X_i, X_j) is $w(X_i, X_j) = \sum p(u, v)$, $\forall u \in ACCESS(X_i)$, $\forall v \in ACCESS(X_j)$, and $MW(u) \cap MW(v) \neq \emptyset$, where

$$p(u, v) = \frac{|MW(u) \cap MW(v)|}{|MW(u)| * |MW(v)|}.$$

Based on the definition of the VIG, we are able to define the variable partitioning problem as follows:

Definition 2.7: Given a VIG $G_V = \langle V, E, w \rangle$, and let n be the number of partitions required, the variable partitioning problem is to partition V into n disjoint sets P_1, P_2, \dots, P_n , such that

$$\sum_{\forall i, j=1, \dots, n} \sum_{\forall u \in P_i, v \in P_j} w(u, v), \text{ for } i \neq j$$

is maximum, where $P_1 \vee P_2 \vee \dots \vee P_n = V$, and $w(u, v)$ is the edge weight of $e(u, v)$.

Variable partitioning problem is a NP-Complete. When all the edges have unit edge weights, variable partitioning problem can be reduced to another well known NP-Complete problem – maximum cut problem [19]. In the next section, we will give the algorithms to solve the variable partitioning problem on the VIG.

III. VARIABLE INDEPENDENCE GRAPH CONSTRUCTION

In this section, we discuss the problem of building the variable independence graph for variable partitioning problem, and show the effect of different graph model constructions on the variable partitioning results.

A variable independence graph 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 is aimed to produce shorter schedule, the accuracy of the variable independence graph is limited by the unknown positions of the memory operations in a schedule. We would like to provide a complete and accurate view for variable partitioning as possible, but on the other hand, we also would like to keep the flexibility so that the partitioning process can work with different scheduling algorithms. The intricacy of building the graph model for variable partitioning problem is how to keep certain level of accuracy of the parallelism and still have a graph working for variable partitioning problem in an effective way.

In the following, we show how to construct the VIG incrementally in three steps. To construct a VIG that provides

proper information of the potential memory access parallelism in a DFG, we need to discuss the underlying properties of the memory operation node pairs that are possible to be executed in parallel. Intuitively, two memory operations could be parallelized if they are not reachable from each other, i.e., there is no zero-delay path between two memory operations in DFG. In other words, if a memory operation v is reachable from another memory operation u via a zero-delay path, then nodes u and v must be executed in sequence. Thus, the memory operations we only need to consider are the node pairs that are not reachable through any zero-delay path in the data flow graph, that is, the memory operations that are independent pairs. As a result, two variables may need to be assigned to different memory modules if their operations are of an independent pair. Accordingly, we can construct a VIG based on the existing independent pairs in the data flow graph. We call this variable independence graph VIG-1.

Construction VIG-1. Given data flow graph $G_D = \langle V, E, X, d, t \rangle$, $\forall X_i, X_j \in X$, and $X_i \neq X_j$, if there exists a pair of nodes $u \in \text{ACCESS}(X_i)$ and $v \in \text{ACCESS}(X_j)$ that are independent pairs in G_D , then there is an edge (X_i, X_j) in the variable independence graph.

Figure 3(a) shows a simple data flow graph with a cycle with one delay. To find all the independent pairs of memory operation nodes, we only need to look at the DAG portion of the DFG after removing the edge with delay, as shown in Figure 3(b). In this example, the access sets of the variables are $\text{ACCESS}(A)=\{0\}$, $\text{ACCESS}(B)=\{1,6\}$ and $\text{ACCESS}(C)=\{3,4\}$. Since only node 0 and node 1 are an independent pair, there is an edge between node A and node B in VIG-1, which is shown in Figure 3(c).

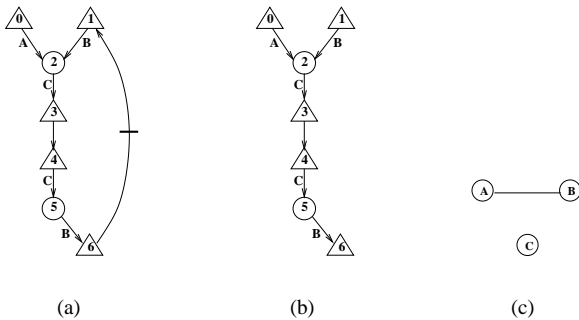


Fig. 3. (a) Cyclic data flow graph. (b) DAG after removing the edge with a delay. (c) Variable independence graph.

We build the edges of VIG-1 by finding out all independent pairs of memory operation nodes in the data flow graph. There can be plenty of memory operation pairs that are not reachable from each other in the DAG portion of a data flow graph. However, not all of them will be scheduled in the same control step and be executed in parallel. The independent pairs that cannot be scheduled in one control step also produce edges in VIG-1. These edges can mislead the decision of variable partitioning. Then, an inferior variable partitioning result may be produced. In the next construction, we use the Mobility Window (MW) concept to eliminate those edges.

If the mobility windows of two memory operations are not overlapped, these two nodes cannot be executed in parallel. Therefore, we should only consider those independent pairs whose mobility windows are overlapped. This improvement is made in the second construction, VIG-2.

Construction VIG-2. Given data flow graph $G_D = \langle V, E, X, d, t \rangle$, $\forall X_i, X_j \in X$, and $X_i \neq X_j$, if there exists a pair of nodes $u \in \text{ACCESS}(X_i)$ and $v \in \text{ACCESS}(X_j)$ that are independent pairs in G_D , and $MW(u) \cap MW(v) \neq \emptyset$, then there is an edge (X_i, X_j) in the variable independence graph.

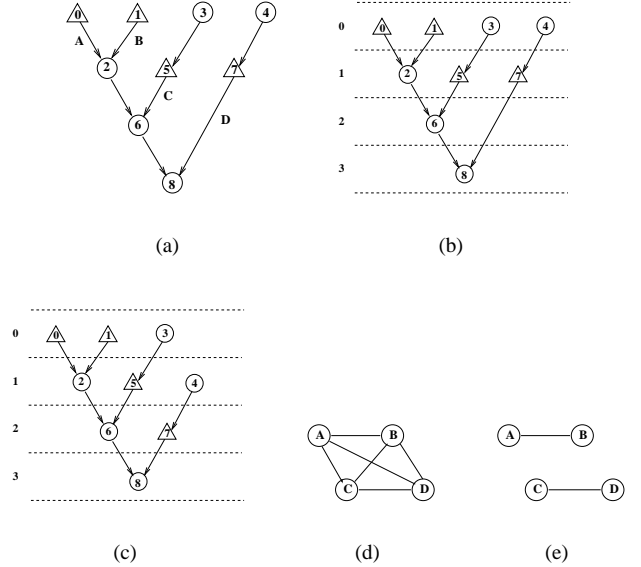


Fig. 4. (a) A data flow graph. (b) ASAP schedule. (c) ALAP schedule. (d) VIG-1. (e) VIG-2.

The example in Figure 4 shows the VIG that considers only the memory operations with overlapped mobility windows in the data flow graph. Figure 4(b) and Figure 4(c) show the ASAP schedule and ALAP schedule of the data flow graph in Figure 4(a). From these schedules, we get the mobility windows of the memory operation nodes as follows: $MW(0) = MW(1) = \{0\}$, $MW(5) = \{1\}$ and $MW(7) = \{1, 2\}$. In Figure 4(d), we have a completely connected graph of VIG-1. However, it does not convey any useful information for variable partitioning. In Figure 4(e), only nodes A and B, as well as nodes C and D, are connected by the definition of VIG-2. It indicates that nodes A and B need to be assigned to different memory modules, as well as nodes C and D. It is clear that the preciseness of memory access parallelism in the VIG is improved in the VIG-2, after excluding the unrealizable parallelism.

VIG-2 improves the preciseness in identifying the potential memory access parallelism existing in a DFG by employing the mobility property of the nodes in a schedule. However, different independent pairs may have different possibilities to occur in one control step. To maximize the potential parallelism among memory accesses, we need to select some appropriate priority metric, so that the variable partitioning

process will favor the partitioning requests with higher possibility to be parallelized. By using the possibility of memory accesses occurring in parallel, that is defined in Definition 2.6, we are able to complete the construction of our variable independence graph.

Construction VIG-Best. Given VIG-2 $G_V = \langle V, E, w \rangle$ and its data flow graph $G_D = \langle V_D, E_D, X, d, t \rangle, \forall X_i, X_j \in V$, and $X_i \neq X_j$, the edge weight $w(X_i, X_j) = \sum p(u, v), \forall u \in \text{ACCESS}(X_i), \forall v \in \text{ACCESS}(X_j)$.

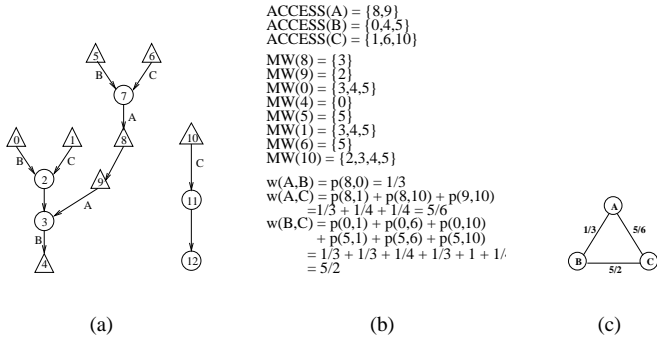


Fig. 5. (a) The DAG of the example in Figure 1. (b) The computation of edge weight. (c) VIG-Best.

We use the example in Figure 1 to illustrate the computation of edge weight. Figure 5(a) shows the DAG portion of the data flow graph in Figure 1. Figure 5(c) is the resulting VIG based on the definition of VIG-Best. It shows that $w(B, C) > w(A, C) > w(A, B)$. Therefore, B and C should be assigned to different memory modules with the highest priority, and then, A should be assigned to a memory module different from that contains array variable C, according to edge (A, C) with the second highest priority. It is also suggested that A and B should be in the same memory module since edge (A, B) has the least priority in parallel memory access. Hence, the first memory module should store {A,B} while the second one stores {C}. With this resulting partition, the schedule will be shortened by 1 control step as shown in Figure 2.

In the algorithm that constructs Variable Independence Graph, VIG-Best, the DFG with cycles is transformed to a DAG by removing the nonzero delay edges. It then computes Access Set (as in Definition 2.4) and Mobility Window for every memory operation node in the DFG. By checking the Access Set, we can identify all independent pairs of memory operations. By checking the Mobility Window, we exclude the edges produced by memory nodes that cannot be scheduled in the same control step. The priority of the edge is then computed by Definition 2.6.

IV. VARIABLE PARTITIONING ALGORITHM

Since the Variable Independence Graph provides a complete view of potential parallelism of memory operations, we present an algorithm for variable partitioning to ensure a near-optimal partition based on the VIG-Best. In Algorithm IV.1, we always choose to create a partition that has the largest edge weight. The algorithm terminates when all the nodes in the input graph are partitioned.

Algorithm IV.1 Procedure of Variable Partitioning on Multiple Memory Banks

Input: A Variable Independence Graph $G_V = (V, E, w)$ and the number of memory modules n
Output: Variable Partitions P_0, P_1, \dots, P_{n-1}

```

WG ← 0;
P0 ← ∅, P1 ← ∅, ..., Pn-1 ← ∅;
P ← RandomSelect(P0, P1, ..., Pn-1);
Randomly select node u ∈ V;
P ← P ∪ ACCESS-1(u);
Mark node u as an assigned node;
u.partition ← P;
while There are nodes that has not been assigned to a partition do
  for all Unmarked node vi ∈ V do
    for all Partition Pj, for 0 ≤ j < n do
      Pj.weight ← the edge weight among Pj and all the other partitions when ACCESS-1(vi) is assigned to Pj;
    end for
    vi.weight_gain ← maxj(Pj.weight);
    vi.partition ← Pk, such that Pk.weight = maxj(Pj.weight);
  end for
  WG ← maxi(vi.weight_gain);
  P ← vm.partition, such that vm.weight_gain = WG;
  P ← P ∪ ACCESS-1(vm);
  Mark node vm as an assigned node;
end while
return P0, P1, ..., Pn-1;

```

V. ROTATION SCHEDULING WITH VARIABLE REPARTITIONING

In this section, we present a novel scheduling algorithm for multiple memory module architecture, called Rotation Scheduling with Variable Repartitioning algorithm (RSVR). This scheduling algorithm is an extension of rotation scheduling [15], which produces a compact schedule by repeatedly applying implicit retiming operations. Before we introduce this algorithm, let's consider the effect of retiming on the VIG and variable partition.

A. Retiming effect on Variable Partitioning

Retiming moves a delay in the data flow graph in the following way: One delay on all the incoming edges of a node is pushed through the node, and pushed out to all its outgoing edges. For cyclic data flow graphs, the number of delays in any cycle remains constant after retiming [15], [16]. Hence, the retiming operation can change the critical path length as well as inter-iteration dependencies in the graph. Consequently, more parallelism among memory operations may be affected. As a result, variable independence graph may be changed after retiming, and so does the variable partitions. The example in Figure 6 shows the different data flow graphs and VIGs before and after retiming. In this example, retiming increases the memory access parallelism, because memory operation nodes from different iterations are able to be executed in the same iteration of the new graph, as shown in Figure 7. The increase in memory access parallelism after retiming is conveyed by the VIG in Figure 6(b). Our RSVR algorithm utilizes the effect of retiming to pipeline the different iterations and generate a

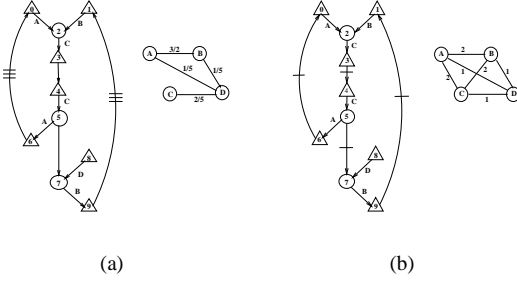


Fig. 6. (a) DFG and VIG before retiming. (b) DFG and VIG after retiming.

compact schedule. Meanwhile, it avoids the re-construction of VIG by adjusting the variable partitions when necessary, and produce a compact schedule where parallel memory accesses are effectively exploited.

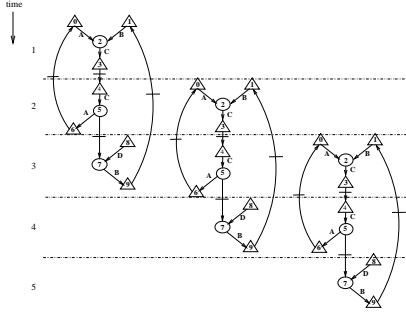


Fig. 7. Reorganized iterations after retiming.

B. Rotation Scheduling with Variable Repartitioning Algorithm

RSVR algorithm modifies the original rotation scheduling by considering multiple memory modules while constructing a schedule. The novelty of this algorithm is that it moves the variables to a new partition during software pipelining when necessary.

Variable repartitioning during scheduling is efficient and crucial in generating software-pipelined schedule in multiple memory module architecture. In RSVR, the resulting partitions obtained by Algorithm IV.1 can be used as the initial assignment of variable partitions. For the clarity of illustration, each rotation phase rotates and reschedules the node set of the first row in the schedule [15]. All the rescheduled nodes have to preserve the dependencies in the DFG. For memory nodes, they have to be rescheduled to the same memory module as the other nodes in its access set. When the schedule length can not be improved in a rotation phase, the algorithm finds out the memory module M that prevents the schedule length to be improved. Then it attempts to find some other available memory module M' where some node set in module M can be moved. If there is no memory module that has enough space, it tries to find some memory module that has some space and find some node set in that module to exchange the memory assignment with the node set in module

Algorithm V.1 Procedure of Rotation Scheduling with Variable Repartitioning

Input: DFG $G_D = \langle V, E, X, d, t \rangle$; initial schedule S

Output: The final schedule S_{best}

```

 $S_{best} \leftarrow S$ 
/* Rotation Phase */
for all  $i = 1$  to  $S.length$  do
   $Q \leftarrow FirstRow(S)$ ;
   $S_{temp} \leftarrow S$ ;
   $S \leftarrow Reschedule(Q, S, G_D)$ ;
  if  $S.length \leq S_{best}.length$  then
     $S_{best} \leftarrow S$ ;
  else
    Find the memory module  $M$  that cause the increase of
     $S.length$ ;
     $S \leftarrow S_{temp}$ ;
    if There is a memory module  $M'$  that has enough space
    to accommodate a node set  $U$  on  $M$ , without violating the
    dependencies in  $G_D$  and memory module assignment in  $S$ 
    then
      Move  $U$  to  $M'$ ;
    else if There is some space on  $M'$ , but not enough for node
    set  $U$  then
      Find a node set  $U'$  on  $M'$ , s.t.  $|U| > |U'|$  and  $U$  and  $U'$ 
      can be swapped without violating the dependencies in  $G_D$ 
      and memory module assignment in  $S$ ;
      Exchange memory module assignment of node sets  $U$  and
       $U'$ ;
       $S \leftarrow Reschedule(U, S, G_D)$ ;
       $S \leftarrow Reschedule(U', S, G_D)$ ;
    else
      Exit the loop;
    end if
  end if
end for
Return  $S_{best}$ ;

```

M . After rescheduling the swapped node sets, the schedule length can be improved, and a new rotation phase can be conducted. The algorithm runs this procedure iteratively and produces the best schedule that contains the adjusted variable partitions when it terminates. Note that for each round of rotation, only a few number of nodes need to be rescheduled. Also, the repartitioning is conducted only when the rotation scheduling can not improve the schedule length anymore. In this algorithm, we implicitly retimed the DFG in each rotation phase to produce a more compact schedule. As we discussed in Section V-A, the VIG is updated according to the retiming operations. However, RSVR algorithm does not reconstruct the variable independence graph. Instead, it produces a good variable partition assignment for a near-optimal schedule by adjusting the partitions when it's necessary in scheduling. Algorithm V.1 shows the procedure of RSVR.

VI. DESIGN SPACE EXPLORATION FRAMEWORK USING RSVR

By given a number of functional units (including ALUs and memory modules), RSVR can produce compact schedule for a multiple memory module architecture effectively. Using the algorithm, we can also explore the minimum number of required memory modules and functional units for a given schedule length requirement. The input of the design space

exploration framework is the data flow graph of an application and a schedule length requirement. The As Soon As Possible (ASAP) scheduling is used to generate both the initial schedule and the upper bound of the number of functional units. If the initial schedule length is less than the schedule length requirement, we iteratively reduce the number of functional units as long as the resulting schedule length still satisfies the schedule length requirement. In each iteration, it reduces the number of functional units by 1, and then the schedule is computed by using RSVR algorithm. If the new schedule length is acceptable, we continue to reduce the number of functional units. On the other hand, if RSVR algorithm cannot generate a schedule within the required schedule length, the program reports the last feasible schedule and exit. If the initial schedule length cannot meet the schedule length requirement, the input data flow graph is transformed using retiming algorithm [16] to generate a new data flow graph with minimum critical path length, till the critical path satisfies the schedule length requirement. Otherwise, it exits with no feasible solution.

VII. EXPERIMENTS

We have experimented with our approaches on several selected benchmarks, including IIR Filter (IIR), Differential Equation Solver (DEQ), All-Pole Filter (All-Pole), Fifth Order Elliptic Filter (Elliptic), 4-stage Lattice Filter (4 Lattice), and Volterra Filter (Volterra). Our approach is divided into stages: First, we construct Variable Independence Graph from DFG using the definition of VIG-Best. Second, we find the initial partition by applying our variable partitioning algorithm on the VIG. Third, we create the DFG schedule based on the initial partition result by using a list scheduling. Finally, in order to improve the schedule, a new scheduling algorithm for multiple memory module architecture, RSVR, is used to iteratively compact the schedule and adjust the variable partitions. The experimental results are very promising.

Table I compares the schedule lengths generated by three different approaches assuming a processor with two ALUs and two data-memory modules. The schedule lengths generated on a simulated processor with two ALUs and only one memory module is also listed as in the second column to compare with the schedule lengths on multiple memory module architecture. The three different approaches are: 1) list scheduling with variable partitioning using Interference Graph [11] [10], shown in column LS-IG. 2) list scheduling with variable partitioning using Variable Independence Graph, shown in column LS-VIG. 3) Rotation Scheduling with Variable Repartitioning algorithm using VIG, shown in column RSVR-VIG. The percentage of reduction on a schedule length compared to column LS-IG is shown in “Improvement” columns under fields LS-VIG and RSVR-VIG. We can see significant improvements by using the VIG model for list scheduling and more improvements for RSVR-VIG approach. The experiments show that the approach using Interference Graph gives a longer schedule as shown in column LS-IG. The reason is that the Interference Graph does not consider the memory access parallelism of the variables produced from different iterations. Since most applications frequently used in DSPs, such as filters, contain intensive

loop computations with inter-iteration dependencies, RSVR-VIG approach will be very effective. The other drawback of the IG is that it does not have an effective priority metric for solving the variable partitioning problem. The LS-VIG approach gives a schedule length improvement over the LS-IG approach by 28% on average. For our RSVR-VIG approach, the average improvement over LS-IG is 44.8%. The largest improvement on schedule length is up to 52.9%.

Bench	LS-IG	LS-VIG		RSVR-VIG	
		length	Improv	length	Improv
IIR	17	12	29.4%	8	52.9%
DEQ	21	11	52.1%	11	52.4%
All-pole	29	27	6.9%	18	37.9%
4 Lattice	33	23	30.3%	22	50.0%
Elliptic	35	28	20.0%	24	31.4%
Volterra	41	29	29.3%	23	43.9%

TABLE I

COMPARISON OF THE SCHEDULE LENGTHS FOR SELECTED BENCHMARKS.

We also conducted experiments for various types of architectures for the same set of benchmarks to study the impact of the number of data-memory modules on the schedule length. The schedule lengths produced by list scheduling and RSVR are list in Table II with various functional unit configurations, where the letter “A” denotes ALUs, the letter “M” denotes memory modules. A configuration “2A1M”, for example, comprises 2 ALUs and 1 memory module. The experimental results show that the increasing number of memory modules does improve the schedule lengths. Furthermore, our RSVR scheduling algorithm can generate much more compact schedules than list scheduling when the number of memory modules is increased. That is, RSVR algorithm can exploit the parallelism among memory operations for multiple memory module architectures much better than the list scheduling. One observation from the experiments is that, the schedule lengths are improved dramatically when the number of data-memory modules is increased from one to two modules. After that, the improvement on schedule lengths becomes smaller when we continue to increase the number of memory modules. This shows that for these applications, dual memory module architecture is an appropriate solution.

Bench	1A1M		2A1M		1A2M		2A2M		2A3M		2A4M	
	LS	RSVR	LS	RSVR	LS	RSVR	LS	RSVR	LS	RSVR	LS	RSVR
IIR	17	16	17	16	13	9	12	8	12	7	12	6
DEQ	21	21	21	21	13	11	11	11	11	8	11	7
All-pole	29	29	29	29	27	20	27	18	27	11	27	9
4 Lattice	44	44	44	44	27	26	23	22	19	16	19	14
Elliptic	42	41	35	35	40	36	28	24	28	21	28	17
Volterra	41	38	41	37	30	27	29	23	29	17	29	15

TABLE II

SCHEDULE LENGTHS OBTAINED BY LIST SCHEDULING AND ROTATION SCHEDULING WITH VARIABLE REPARTITIONING FOR THE DIFFERENT NUMBER OF FUNCTIONAL UNITS.

In the following set of experiments, we use our design space exploration algorithm presented in Section VI to produce the minimum number of functional units required for

satisfying the given schedule length requirements. We compare the solutions obtained by our method, i.e, RSVR with VIG, and the method of LS with IG. As shown in Table III, our algorithm gives feasible solutions for all the schedule length requirements on all tested benchmarks, while the other cannot give feasible solutions for most cases. Furthermore, for the same schedule length requirement, our algorithm is able to give a solution with fewer or equal number of functional units and memory modules. For example, for a schedule length requirement of 35 cycles or less for elliptical filter, the minimum functional units generated by our algorithm is two ALUs and one memory module, while the solution given by the list scheduling with interference graph uses one more memory module to meet the schedule length requirement. For the target schedule length of 20 cycles, using the IG cannot produce feasible schedules in most cases, while our approach can give feasible solutions. These experimental results show that our design space exploration algorithm can also be used to explore the minimum required number of functional units effectively for a specified performance requirement in the multiple memory module architecture.

Bench	Target Schedule Lengths									
	45		35		25		20		15	
	IG	Ours	IG	Ours	IG	Ours	IG	Ours	IG	Ours
IIR	1A1M	1A1M	1A1M	1A1M	1A1M	1A1M	1A1M	1A1M	X	1A2M
DEQ	1A1M	1A1M	1A1M	1A1M	1A1M	1A1M	X	1A2M	X	1A2M
All-pole	1A1M	1A1M	1A1M	1A1M	X	1A2M	X	1A2M	X	2A3M
4 Lattice	1A1M	1A1M	X	1A2M	X	2A2M	X	2A3M	X	2A4M
Elliptic	1A1M	1A1M	2A2M	2A1M	X	2A2M	X	2A4M	X	3A4M
Voltera	1A1M	1A1M	X	1A2M	X	2A2M	X	2A3M	X	2A4M

TABLE III

THE MINIMUM NUMBER OF FUNCTIONAL UNITS FOR REQUIRED SCHEDULE LENGTHS OBTAINED BY RSVR WITH VIG AND LS WITH IG.

VIII. CONCLUSION

The performance improvement of multiple memory module architectures strongly depends on the variable partitioning technique and the schedule of instructions such that the capability of parallel memory accesses is fully exploited. In this paper, we presented a new graph model, Variable Independence Graph, which displays the potential memory access parallelism between variables. The graph model is built incrementally to show the potentially parallel accesses that may actually occur later in scheduling. The graph provides effective information to help decide variable partitions. We present a new scheduling technique, *Rotation Scheduling with Variable Repartitioning* (RSVR), to generate a compact schedule for multiple memory module architectures. Because DFG scheduling and variable partitioning are performed at the same time, it produces a better schedule compared with the existing approach. A new design space exploration approach based on RSVR allows designers to effectively explore feasible schedules with minimum number of fundamental unit and memory modules for a given schedule length requirement. Experimental results on selected benchmarks show that our approach gives a compact schedule for the architecture with

multiple memory modules, while the other approach produces a longer schedule on the same architecture. In our design exploration scheme, we gives a feasible schedule using fewer number of functional units and memory modules compared to the approach based on the interference graph under the same schedule length requirement.

REFERENCES

- [1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang, "Challenges in code generation for embedded processors," in *Code Generation For Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publishers, 1995, ch. 1, pp. 4–17.
- [2] S. Y. H. Liao, "Code generation and optimization for embedded digital signal processors," Ph.D. dissertation, MIT, 1996.
- [3] D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: Retargetable code generation for embedded processors," in *Code Generation For Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publishers, 1995, ch. 5, pp. 85–296.
- [4] Z. Wang, T. W. O'Neil, and E. H.-M. Sha, "Minimizing average schedule length under memory constraints by optimal partitioning and prefetching," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 27, pp. 215–233, Jan. 2001.
- [5] Z. Wang, M. Kirkpatrick, and E. H.-M. Sha, "Optimal two level partitioning and loop scheduling for hiding memory latency for DSP applications," in *Proc. 37th ACM/IEEE Design Automation Conf. (DAC)*, Jun. 2000, pp. 540–545.
- [6] *DSP56000 24-Bit Digital Signal Processor Family Manual*, Motorola, Inc., 1996.
- [7] *ADSP-21000 Family Application Handbook Volume 1*, Analog Devices, Inc., 1994.
- [8] *TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instruments, Inc., Oct. 2000, (literature number SPRU189F).
- [9] *GEPARD Family of Embedded Software Programmable DSP Core*, Austria Micro System, <http://asic.amsint.com/databooks/digital/gepard.htm>.
- [10] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank DSPs," in *Proc. IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, vol. 2, May 2001, pp. 1121–1124.
- [11] M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting dual data-memory banks in digital signal processors," in *Proc. 7th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 234–243.
- [12] A. Sudarsanam and S. Malik, "Simultaneous reference allocation in code generation for dual data memory bank ASIPs," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 2, pp. 242–264, Apr. 2000.
- [13] P. R. Panda, "Memory bank customization and assignment in behavioral synthesis," in *Proc. ACM/IEEE Int'l Conf. on Computer Aided Design*, Nov. 1999, pp. 477–481.
- [14] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. SIGPLAN'88 ACM Conf. on Programming Language Design and Implementation*, June 1988, pp. 318–328.
- [15] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 3, pp. 229–239, Mar. 1997.
- [16] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, Aug. 1991.
- [17] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc., 1995.
- [18] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.
- [19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.