

Exploring Variable Partitioning for Dual Data-Memory Bank Processors

Qingfeng Zhuge
Department of Computer
Science
University of Texas at Dallas
Richardson, Texas 75083
qfzhuge@utdallas.edu

Bin Xiao
Department of Computer
Science
University of Texas at Dallas
Richardson, Texas 75083
bxiao@utdallas.edu

Edwin H.-M. Sha
Department of Computer
Science
University of Texas at Dallas
Richardson, Texas 75083
edsha@utdallas.edu

ABSTRACT

Dual data-memory banks are found in more and more embedded processors for high-performance media or DSP applications. It offers better performance by providing potentially doubled memory bandwidth. However, making effective use of dual memory banks remains difficult. And there's little research work that has been done to study variable partitioning problem model.

In this paper, we present a new graph model for tackling the variable partitioning problem, namely, *variable independence graph*, which shows the most precise information for variable partitioning compared to the previous graph models for this problem. We also present algorithms for variable partitioning. The experiment results show constant improvements on schedule length by using the variable independence graph for variable partitioning.

Categories and Subject Descriptors

C.4 [Computer Systems Organization]: Performance of Systems—*modeling techniques*; C.1.4 [Processor Architecture]: Parallel Architectures; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—*signal processing systems*

General Terms

Performance, Algorithm, Experimentation

1. INTRODUCTION

It's well known that high-performance media and DSP applications require strict real-time processing, while the growing gap of speed between memory and CPU becomes bottleneck for such real-time systems. To reduce this speed gap, embedded systems need to utilize on-chip memories. Since the size of on-chip memory is very limited, the technique of variable mapping and scheduling becomes one of the most important factors in performance optimization.

To improve the overall performance, Harvard architecture provides simultaneous accesses to separate memory modules for instructions and data. Some of the most advanced DSPs are even equipped with on-chip dual data-memory banks accessible in parallel, such as Motorola 56000 [1] and Analog Devices ADSP-2106x series [2]. Since data can be partitioned and allocated to separate data-memory banks, and can be accessed simultaneously, the dual data-memory bank architecture actually offers potentially higher memory bandwidth, and thus potentially higher performance. This architectural feature is very attractive for high-performance media applications. In fact, many DSP routines, such as FIR filters, require the convolution of two data arrays as a kernel operation. Processors with dual memory banks can achieve higher memory bandwidth for this kind of application.

One major problem arises for processors with dual memory banks is how to map the array variables to two memory banks appropriately to use the high memory bandwidth capacity efficiently. For example, for a typical array operation such as

```
for (i=0; i<N; i++) y += A[i] * B[N-i];
```

it must ensure that arrays A and B are placed in different memory banks, so that the entire loop body can be encoded into a single multiply-accumulate instruction. Many existing C compilers cannot work well in a dual memory bank architecture. Instead, all program vari-

ables are assigned to just one bank. It's obvious that this strategy will result in a performance loss, since the potential instruction-level parallelism is not exploited. Consequently, variable partitioning technique plays an important role in this architecture. But there's little research work that has been done to study the models and properties of the variable partitioning problem.

A variable partitioning technique for dual memory bank DSPs has been described in [3]. The problem is transformed to a partition problem of *interference graph* (IG) [4] [5]. Nodes of interference graph represent variables in the program. the graph edges are used to reflect the potential parallelism between variables. The total access times of two variables of an edge is used as priority in partitioning. It's claimed that the best partitioning is achieved when the total weight of the edges between two partitions is maximal. The problem of this method is that the information in interference graph is not sophisticated enough to reveal the potential parallelism exists in the program. For example, interference graph can only be applied to *directed acyclic graph* (DAG). It does not consider the inter-iteration data dependencies in the program. The other problem of interference graph is that it does not identify the fact that some potential parallelism of memory accesses are actually impossible to be implemented in a schedule. So an appropriate priority metric need to be chosen to favor some parallel memory access requests in variable partitioning.

We present a new graph model for variable partitioning problem. We name it as *variable independence graph*. This graph reflects all the potentially parallel memory accessed that may actually occur in scheduling. It also gives a priority metric for different kind of parallel access demand. The graph is constructed from *data flow graph* of the program. So it is able to deal with the data flow graph with cycles and delays. It aims to convey the information of potential parallelism as precisely as possible before the actual scheduling starts. A new variable partitioning algorithm and an algorithm for graph construction are also presented. The experiment results on a variety of filter applications show that constant improvements are achieved on schedule lengths.

The structure of this paper is as follows. Section 2 introduces variable partition problem. Section 3 presents an incremental definition of our graph model. The algorithms for graph model construction and variable partitioning are illustrated in Section 4. Section 5 compares the experiment results on different graph models, and also discussed the retiming effect on variable partitioning. And finally, conclusions are given in Section 6.

2. PROBLEM DESCRIPTION

For a processor equipped with multiple ALUs and multiple memory modules, the variable partition problem is to map the variables of a program to different memory modules, so that, memory operations access the different memory modules can be preceded in parallel. It is obvious that some partition result allow more parallel memory accesses to be implemented in scheduling. And it may help a scheduler to produce a shorter schedule.

DEFINITION 2.1. *For a multiple memory modular architecture, Variable Partitioning Problem is to partition the variables into disjoint sets, with respect to multiple memory modules. So that variables in different memory modules can be accessed simultaneously. And maximal memory access parallelism can be actually achieved by a scheduler.*

By taking a further step to look inside the Variable Partition Problem, we can find that there are two aspects of this problem need to be addressed and resolved.

Part 1 – To identify all potential parallelism for variable accesses.

The potential parallelism of memory operations in dual memory bank processors help the scheduler to produce shorter schedules. The more they exist, the more chances that can be taken in scheduling.

Part 2 – To maximize the realizable parallelism for a given program.

The potential parallelism in memory operations are to be implemented by a scheduler. And some memory operations may have more opportunities to be scheduled in the same control step than others. To maximize the realizable parallelism for a given program, we need to choose some appropriate priority besides the number of edges across different partitions. In a graph model, this part of the problem can be translated into a graph problem.

We will see that the variable partition problem is difficult, since variables may present contrary requests for assigning a variable when they appear in different code segments. And it should be noted that the implementation of parallel memory accesses based on a certain variable partition can only be determined in a dedicated scheduler. So an appropriate graph model is very important in providing correct information for variable partitioning. In the next section, we will take an incremental approach to introduce *Variable Independence Graph*. We will also present algorithms of the *Variable Partitioning Problem* for a dual memory bank architecture.

3. DEFINITIONS AND PROPERTIES

Many media and DSP application programs can be represented by a *Data Flow Graph (DFG)*. It reflects the data dependencies of both ALU operations and memory operations inside iteration or between iterations.

DEFINITION 3.1. A *Data Flow Graph (DFG)* is a directed weighted graph $G_D = \langle V, E, d, t, S \rangle$ where V is a set of computation nodes that represent ALU operations and memory operations (load or store), E is the edge set which defines the precedence relations from nodes in V to nodes in V , $d(e)$ represents the number of delays for an edge e , $t(e)$ represents the computation time of a node v , $S(e)$ represents the variable accessed by a node v on edge e ($u \rightarrow v$).

An edge e from u to v with $d(e)$ delays means that the computation of node v at iteration i depends on the computation of node u at iteration $i - d(e)$. An edge without delay represents precedence relation within an iteration. A static schedule must obey the precedence relations defined by the subgraph consisting of edges without delays in a DFG.

In the DFG of our definition, memory operations are represented by pseudo-computation nodes which computation time is 0. An edge ending at or starting from a pseudo-node represents a memory access. Particularly, an edge ending at a pseudo-node represents a Store; an edge starting from a pseudo-node represents a Load. An edge label $S(e)$ indicates the name of a variable that is accessed by edge e . An edge with delays reflects precedence relationship between different iterations. An edge without delay indicates precedence relationship within an iteration. A static schedule must obey the precedence relations defined by the subgraph consisting of edges without delays in a DFG.

The DFG directly shows the input variables for an ALU operation. And it's easy to see that memory operation nodes that are not precedent to each other are candidates for potential parallel accesses. It can also be observed that potential parallelism may come from memory operations in different iterations.

The idea here is to extract and translate these information into independence relationship for the nodes in our a graph model. The objective of the graph construction is to expose the independence relationships to the most extent, which means, a large pool of potential parallelisms are conveyed for the variable partitioning problem. And thus a scheduler can get maximal chances to exploit the memory access parallelism.

Before we go into the graph construction details, we define a *Variable Independence Graph (VIG)* to denote

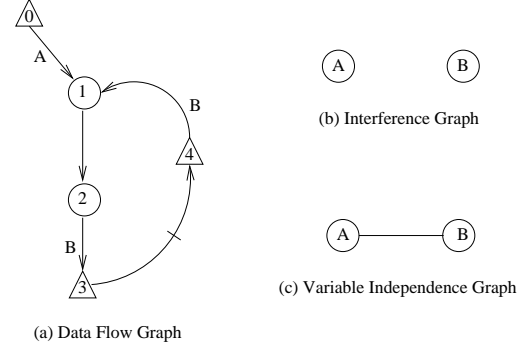


Figure 1: Comparison of Interference Graph and VIG-1.

the independence relationship among the variables for a given DFG.

DEFINITION 3.2. A *Variable Independence Graph (VIG)* is an undirected weighted graph $G_V = \langle V, E, w \rangle$ where V is a node set represents variables, E is an edge set that indicates there is no dependence between the two nodes of a edge $e(u, v)$, i.e., there is potential parallelism between them. $w(e)$ represents the force or extent of independence between the two nodes of edge $e(u, v)$.

For the clarity reason in further definitions, we need to define the term *access set* as follows:

DEFINITION 3.3. Given a DFG $G_D = \langle V, E, d, t, S \rangle$, the *Access Set* of variable X , $ACCESS(X)$, is the set of memory operation nodes that access X .

Now we are ready to build up the first version of variable independence graph. By intuition, if there is a node in the access set of a memory operation is not precedent to another memory operation, these two operations have potential parallelism. So there should be a independence edge between them.

DEFINITION 3.4. The First Build: VIG-1 For any two different variables X_i and X_j , if there exists a pair of sibling nodes from $ACCESS(X_i)$ and $ACCESS(X_j)$, there is an independence edge (X_i, X_j) .

For a DFG with cycles and edge delays, we handle the case by cutting the edge with delays, so transform a DFG with cycles to a DAG. The example in Figure 1 shows the different constructions of a Interference Graph and a VIG-1. Since node 1 consumes

the variable B from the last iteration, A and B can be accessed in parallel. And so maximize the benefit of multiple memory banks bandwidth. In VIG-1, the access sets of the variables are $ACCESS(A) = \{0\}$ and $ACCESS(B) = \{3, 4\}$. Node 0 and node 4 are sibling nodes. So there should be an independence edge (A, B) . It denotes that node 0 and node 4 can be executed in parallel. Compared to the interference graph for this example, the inter-iteration dependencies in DFG are not considered, the variable A and B may be allocated to the same memory module. The benefit of multiple memory modules bandwidth is wasted.

The intuition in this first build of model is to identify memory operations that are not dependent to each other. But not all the independent pairs of variables are useful for variable partitioning. Memory access parallelism is eventually implemented by a scheduler. However, some memory accesses may never be able to appear in the same control step in a schedule. Furthermore, these useless edges may lead to a sub-optimal partitioning, since they also affect the number of edges between the different variable partitions. To exclude these harmful edges in a VIG-1, we consider the possibility that two operations may be scheduled to one control step in a further build of the graph model. And we need to introduce the following definition of *mobility window* (MW) before proceeding to the second version of VIG.

DEFINITION 3.5. Given a DFG $G_D = \langle V, E, d, t, S \rangle$, The *Mobility Window* of some node v in DFG, $MW(v)$, is a series of control steps starting from the position of v scheduled by ASAP scheduling, ending with that scheduled by ALAP scheduling.

PROPERTY 3.1. Two operation nodes are possible to be scheduled to the same control step if and only if their mobility windows overlap.

This property extends our VIG-1 to the second construction of Variable Independence Graph.

DEFINITION 3.6. The *Second Build: VIG-2* For any two different variables X_i and X_j , if there exists a pair of sibling nodes from $ACCESS(X_i)$ and $ACCESS(X_j)$, and $MW(u) \wedge MW(v) \neq \emptyset$, there is an independence edge (X_i, X_j) .

The definition of VIG-2 ensures that only the potential parallelisms that are possible to be implemented in some schedules are considered in variable partitioning. In the example shown in Figure 2, the mobility windows of A and B are $MW(A) = \{3\}$ and $MW(B) =$

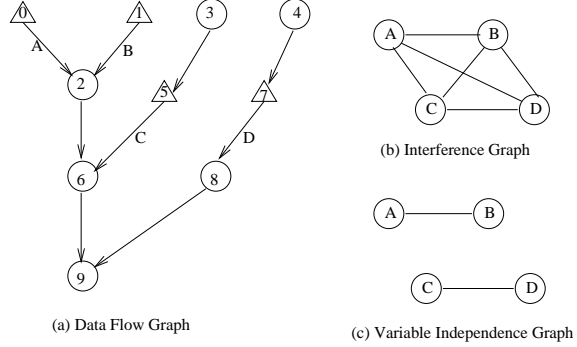


Figure 2: Comparison of Interference Graph and VIG-2.

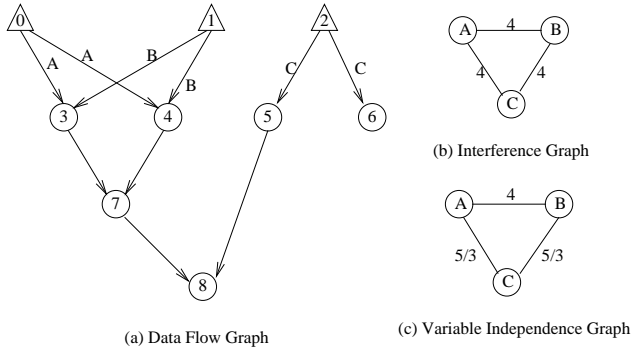


Figure 3: Comparison of Interference Graph and VIG-Best.

$\{3\}$, and those of C and D are $MW(C) = \{2\}$ and $MW(D) = \{2\}$. So, only the edges (A, B) and (C, D) are useful for variable partitioning. All the other edges cannot be actually implemented by a scheduler. Furthermore, these edges may lead to an inferior partitioning result. For example, $Partition_1 = \{A, B\}$ and $Partition_2 = \{C, D\}$ is also a legal partitioning result based on Interference Graph.

VIG-2 gives more detail information for variable partitioning, but a further look into VIG-2 shows that not all the memory operations pairs connected by an independence edge have the same opportunity to be implemented as parallel operations in a schedule. Some of them have much better opportunities than others. So they should enjoy higher priority in variable partitioning. We calculate the probability of each independence edge, and make it a priority metric. This priority information completes the last construction of our Variable Independence Graph.

DEFINITION 3.7. *The Third Build: VIG-Best* For any two different variables X_i and X_j , if there exists a pair of sibling nodes u and v from $ACCESS(X_i)$ and $ACCESS(X_j)$, and $MW(u) \wedge MW(v) \neq \emptyset$, there is an independence edge (X_i, X_j) in Variable Independence Graph. For every pair of nodes u and v , there is a probability that u and v may appear in the same control step within their mobility windows. And the edge weight $w(X_i, X_j)$ is the sum of the probabilities for all pairs of sibling nodes of $ACCESS(X_i)$ and $ACCESS(X_j)$.

The example in Figure 3 shows that independence edge (A, B) have higher priority than edge (A, C) and (B, C) . So variables A and B need to be allocated to different variable partitions. While the interference graph cannot provide this information for partitioning, a sub-optimal partition result may be produced.

Based on the previous definition of Variable Independence Graph, we are able to define the Variable Partition Problem formally.

DEFINITION 3.8. *Variable Partition Problem* is to partition the variables into disjoint sets, and to map the sets to the memory modules. such that the total weight of the edges of a given Variable Independence Graph that cross the different variable partitions is maximal.

DEFINITION 3.9. *A variable partition* is a set of nodes of Variable Independence Graph that are allocated to the same memory module.

Particularly, we define the Variable Partition Problem for dual memory banks DSP as follows:

PROPERTY 3.2. *An optimal variable partitioning for dual memory banks DSP is achieved if the nodes of Variable Independence Graph are divided into two disjoint node sets X and Y , such that the sum of the edge weights between these two node set is maximum.*

4. ALGORITHM OF VARIABLE PARTITIONING

Variable partitioning problem on dual memory banks architecture is NP-completes. Partitioning on the variable independence graph can be reduced to Max Cut problem. In this section, we present two algorithms. The first one is the algorithm for constructing a variable independence graph. The second one is variable partition algorithm. We assume that the list scheduling with longest path length as a weight function is used through this paper.

Algorithm 1 Procedure of Variable Independence Graph Construction – VIG-Best

Input: A DFG G_D with variable names X_i and n vertices $(v_0, v_1, v_2, \dots, v_{n-1})$
Output: A Variable Independence Graph G_V
Construct DAG G_A of DFG G_D ;
 $G_V \leftarrow$ all the memory operation nodes of DFG;
for all Memory operation node v_i that access variable X_i **do**
 Compute $ACCESS(X_i)$;
 Compute $MW(v_i)$;
end for
for all Memory operation node v_i in $ACCESS(X_i)$ **do**
 for all Memory operation node v_j in $ACCESS(X_j)$ **do**
 if v_i and v_j are sibling, and $MW(v_i) \cap MW(v_j) \neq \emptyset$ **then**
 if There's no edge (X_i, X_j) in G_V **then**
 Add edge (X_i, X_j) ;
 end if
 Add edge weight $w(X_i, X_j) \leftarrow w(X_i, X_j) +$
 probability of v_i and v_j appear in the same control step;
 end if
 end for
 end for
return G_V ;

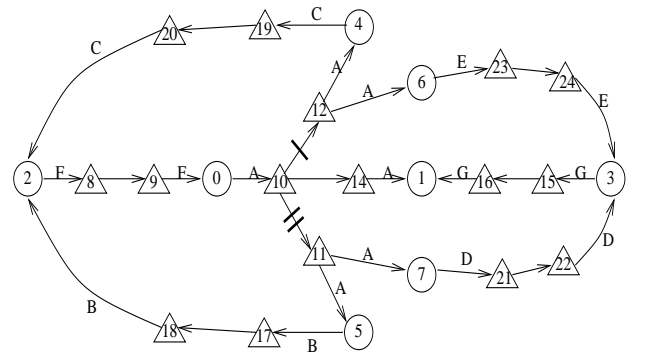


Figure 4: DFG of IIR filter.

Algorithm 2 Procedure of Variable Partitioning on Dual Memory Banks

Input: A Variable Independence Graph G_V with m vertices $(v_0, v_1, v_2, \dots, v_{m-1})$

Output: Variable Partitions P_1 and P_2

```

 $P_1 \leftarrow \emptyset, P_2 \leftarrow \emptyset;$ 
Randomly select a node;
Assign the node to a randomly selected set  $P_1$  or  $P_2$ ;
Mark the partitioned node;
 $Part(v_i) \leftarrow$  partition name;
 $Part(v_i) \leftarrow Part(v_i) + v_i;$ 
while There are nodes that is not partitioned do
  for all Unmarked node  $v_i$  in  $G_V$  do
     $W_1 \leftarrow$  the edge weight gain when  $v_i$  is placed in  $P_1$ ;
     $W_2 \leftarrow$  the edge weight gain when  $v_i$  is placed in  $P_2$ ;
     $W(v_i) \leftarrow \max \{W_1, W_2\};$ 
     $Part(v_i) \leftarrow$  partition name;
    if Edge weight gain  $WG < W(v_i)$  then
       $WG \leftarrow W(v_i);$ 
      Get the next node to be partitioned  $u \leftarrow v_i$ ;
    end if
  end for
  Mark node  $u$ ;
   $WG \leftarrow 0;$ 
   $Part(u) \leftarrow Part(u) + u;$ 
end while
return  $P_1$  and  $P_2$ ;

```

Applications	1 Mem	IG	VIG-Best
IIR Filter	17	17	12
Differential Equation	21	21	11
All-pole filter	29	29	25
4-stage Lattice Filter	44	33	23
Elliptical Filter	35	35	28

Table 1: Schedules based on different models.

Algorithm 1 constructs an *Variable Independence Graph* from a DFG. The DFG with cycles and delays is transformed to DAG by breaking the edges with delay. It stores the *access set* and *mobility window* information for every memory operation nodes in DFG. By checking the access set condition, we can identify all pairs of variables that may be accessed in parallel. By checking the mobility windows, we exclude the edges of the nodes that cannot be scheduled in the same control step. The priority of an independence edge is the summation of all the parallel probabilities of the memory operations that access the same pair of variables. So the potential parallelism used for variable partitioning can never be lost or under-estimated. Our construction of Variable Independence Graph can handle cycles in data dependence graph, while these cycles result in information loss in Interference Graph.

Algorithm 2 uses greedy strategy to partition the nodes of the input Variable Independence Graph into two disjoint sets P_1 and P_2 . The first node is assigned to either of the node sets. We always choose the next partitioned node as a node that creates the largest edge weight gains between two node sets. We can always break a tie by choosing randomly in any case.

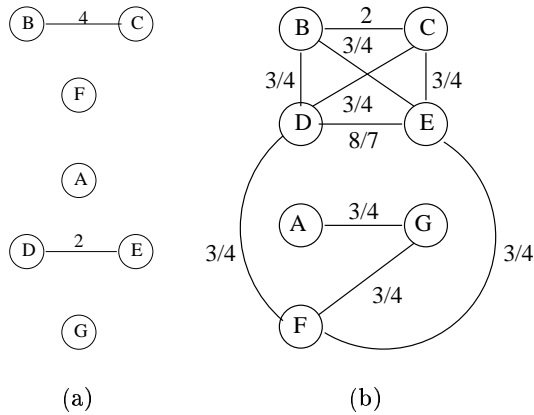


Figure 5: (a) Interference Graph of IIR filter; (b) Variable Independence Graph of IIR filter.

5. DISCUSSION

5.1 Comparison of Graph Models

In this section, we show by experiments that variable partitioning based on *Variable Independence Graph* model indeed helps a scheduler to produce a shorter schedule. We use *list scheduling* with the longest path length as priority function. And the schedules are produced in a simulated architecture with two ALU operation units and two memory banks.

The experiments are done on a variety of filter applications usually used in DSP. For every filter application, we build two different graph models for the variable partitioning, the Interference Graph and the Variable Independence Graph. Because of the space limitation, we only show the DFG and graph models for IIR filter in Figure 4 and Figure 5. Variable partitioning will produce different results from the two graph models. List scheduling is used to produce schedules based on different partition results. A schedule produced without variable partitioning is also shown to be compared

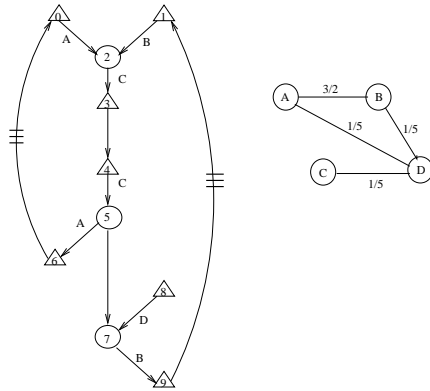


Figure 6: DFG and Variable Independence Graph before retiming.

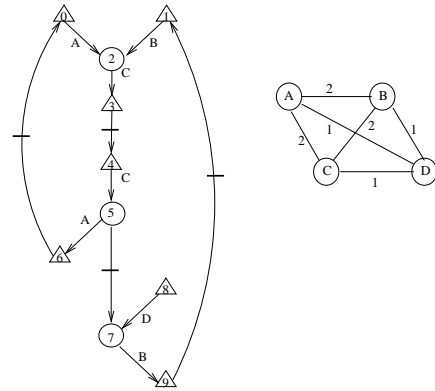


Figure 7: DFG and Variable Independence Graph after retiming.

with the schedule lengths with variable partitioning. The schedule results are listed in Table 1.

The first column is the schedule length of one iteration without variable partitioning. That is, all the variables are just assigned to the same memory module. The schedules in this column do not take the advantage of higher memory bandwidth at all. The data in this table show that these schedules are always the longest ones. The second column in this table represents the schedule lengths based on partitioning result of Interference Graph. The experiments show that Interference Graph cannot help to improve the schedule length in most of our applications. The reason is that when there're data dependency cycles, Interference Graph is not able to identify the potential parallelism relationships within the cycles. Cycles with delays are frequently seen in many multimedia and DSP applications. In some cases, the Interference Graph cannot partition the variables at all, since all the variables in the program are included in some connected cycles. So the result is the same as that without variable partitioning.

In all the experimental cases, the partitioning results based on Variable Independence Graph model help the scheduler produce much shorter schedules than those produced from Interference Graph. This result confirms the efficiency of exploiting the dual memory bank bandwidth by using Variable Independence Graph model in variable partitioning.

5.2 Retiming effect on Variable Partitioning

The retiming moves the delay on DFG edges, and changes the inter-iteration dependencies [6] [7]. It may affect the independence relationship for variables, and so the variable partitioning result. A new Variable Independ-

dence Graph may need to be constructed after retiming. The DFG and Variable Independence Graph of an example before retiming is shown in Figure 6. The retiming effect on this example is illustrated in Figure 7. Figure 8 shows the reorganized iterations after retiming. It's interesting to see that in the third iteration retiming all the *loads* are possible to be executed in parallel if there are enough number of memory modules. This new arrangement actually increases the potential memory access parallelism.

6. CONCLUSION

There are several successful embedded processors explore higher memory access bandwidth by implementing dual memory banks. Some research works have been done on variable partitioning. But no one shows that it can exploit the advantage of this architecture efficiently. The graph model usually used on this problem is Interference Graph. We show that there is imperfection in this graph model. To tackle the variable partitioning problem more effectively, we present a new graph model, namely Variable Independence Graph, to describe the potential memory access parallelism between variables. And a Variable Independence Graph model is incrementally built to expose all the potentially parallel accesses may actually occur later in scheduling. The experiment results show that the variable independence graph is superior to the previous graph models for variable partitioning problem.

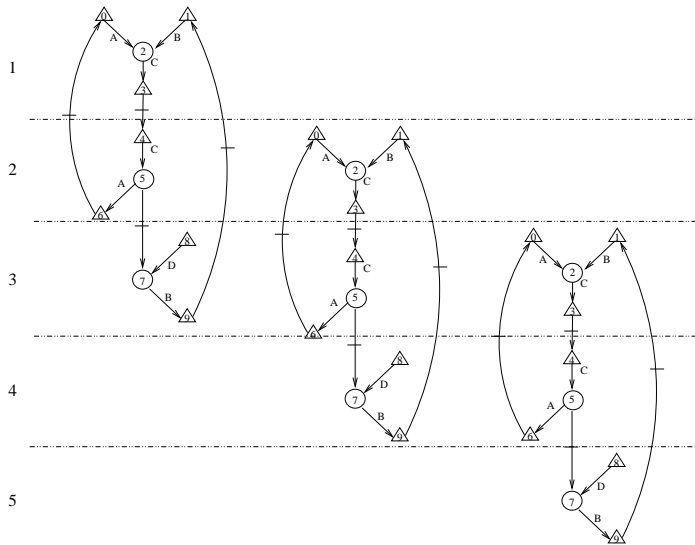


Figure 8: Reorganized iterations after retiming.

7. REFERENCES

- [1] Motorola, Inc., *DSP56000 Digital Signal Processor Family Manual*, 1995.
- [2] Analog Devices, Inc., *ADSP-21000 Family Application Handbook Volume 1*, 1994.
- [3] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank dsp's," in *Proceedings of the 38th International Conference on Acoustics, Speech, and Signal Processing*, pp. 1121–1124, ACM, 2001.
- [4] M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting dual data-memory banks in digital signal processors," in *Proceedings of the 7th International Conference on Acoustics, Speech, and Signal Processing*, pp. 234–243, ACM, 1996.
- [5] A. Sudarsanam and S. Malik, "Simultaneous reference allocation in code generation for dual data memory bank asips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, pp. 242–264, Apr. 2000.
- [6] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.
- [7] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 229–239, Mar. 1997.