ADDRESS ASSIGNMENT SENSITIVE VARIABLE PARTITIONING AND SCHEDULING FOR DSPS WITH MULTIPLE MEMORY BANKS

Chun Jason Xue, Tiantian Liu, Zili Shao, Jingtong Hu, Zhiping Jia, Weijia Jia, Edwin H.-M. Sha

ABSTRACT

Multiple memory banks design is employed in many high performance DSP processors. This architectural feature supports higher memory bandwidth by allowing multiple data memory access to be executed in parallel. Dedicated address generation units (AGUs) are commonly presented in DSPs to perform address arithmetic in parallel to the main datapath. Address assignment, optimization of memory layout of program variables to reduce address arithmetic instruction, has been studied extensively on single memory architecture. Make effective use of AGUs on multiple memory banks is a great challenge to compiler design and has not been studied previously. In this paper, we exploit address assignment with variable partitioning for scheduling on DSP architectures with multiple memory banks and AGUs. Our approach is built on novel graph models which capture both parallelism and serialism demands. An efficient scheduling algorithm, *Address Assignment Sensitive Variable Partitioning (AASVP)*, is proposed to best leverage both multiple memory banks and AGUs. Experimental results show significant improvement compare to existing methods.

Index Terms— Scheduling, Memory Management, Design Automation, Program Compilers

1. INTRODUCTION

High-performance DSP applications generally require strict real-time processing. To increase performance, some DSP processors employ multiple memory bank architecture to provide parallel memory access, such as Motorola 56000, Analog Device ADSP2100, and Gepard Core DSPs. Compiler support is essential to harvest the benefits provided by multiple memory bank architecture. A number of papers [3, 8] have investigated the use of multiple memory banks to achieve maximum instruction level parallelism. These approaches differ in either the models or the heuristics. However, none of these works consider the combined effect of an important hardware feature in most DSPs, Address Generation Units (AGUs). AGUs can perform address computations in parallel to the central data path. As a result, when we access data in register-indirect addressing mode, the address stored in the address register (AR) can be auto-incremented or auto-decremented without extra addressing instruction. Contrary to traditional compilers, DSP compilers can carefully determine the relative location of data in memory and achieve compacted object code size and improved performance. A lot of research [4, 2, 7] has been done to optimize the address assignment of variables to minimize the total number of address arithmetic instructions on single memory architecture. However, none of above work consider address assign problem on multiple memory bank architecture. In this paper, we exploit address assignment together with variable partitioning for scheduling on architectures with multiple memory banks and AGUs to maximize performance.

First, we propose graph models that capture both parallelism for variable partitioning purpose and serialism for address assignment purpose. Variable partitioning is done with consideration of address assignment. After variable partitioning, we then construct an address assignment for each memory bank, and perform scheduling based on the address assignment and memory bank configuration. The experimental results show that compared to using variable partitioning technique alone, our proposed method achieves 14.4% reduction in average schedule length and 20.1% reduction in the number of address instructions.

The remainder of this paper is organized as follows. Section 2 provides a motivating example. Section 3 introduces basic concepts and the architecture model. Graph modeling approach used in this paper is presented in Section 4. The algorithm is discussed in Section 5. Experimental results and concluding remarks are provided in Section 6 and 7, respectively.

2. MOTIVATING EXAMPLE

In this section, we provide a motivating example to show that different partitioning of variables can lead to different results in schedule length. The computation code of the example is shown in Figure 1(a). The data flow graph (DFG) for the code is shown in Figure 1(b). In the DFG, the squares represent memory operations (load/store), while the circles represent ALU operations. The edges represent data dependencies between ALUs and memory operations. Figure 1(c) shows a variable partition based on VIG graph proposed in [8].

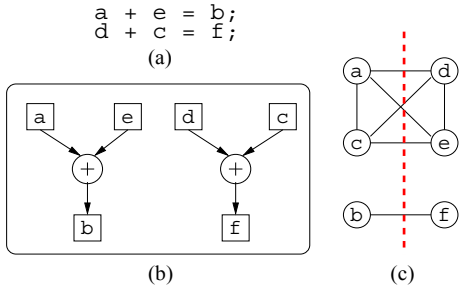

Fig. 1. (a) Example code. (b) Its corresponding DFG. (c) Variable partition.

In Figure 2, based on the partition of {a,b,c} and {d,e,f}, with default variable layout in memory, we got a schedule length of 5. In Figure 3, after we run address assignment algorithm [4] to reorder the variable's layout in memory, we saved two address arithmetic operations. However, the schedule length is still 5. In Figure 4,

Chun Jason Xue and Weijia Jia is with the Dept of C. S., City University of Hong Kong; Email: {jasonxue, wei.jia}@cityu.edu.hk. Tiantian Liu and Zhiping Jia is with the Dept of C. S. ShangDong University China; Email: {i0453, zhipingj}@sdu.edu.cn. Zili Shao is with the Dept of C., Hong Kong Polytechnic University Email: cszlshao@comp.polyu.edu.hk. Jingtong Hu and Edwin Sha are with the Dept of C. S. University of Texas at Dallas, USA; Email: {jxh068100,edsha}@utdallas.edu. This work is partially supported by HK CERG 9041129, CityU 113906, NSF EIA-0103709, and NSF CCR-0309461, USA.

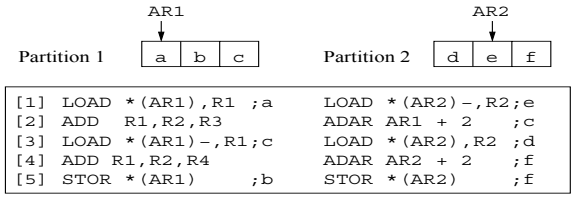

Fig. 2. Schedule with partitions of {a,b,c} and {d,e,f}.

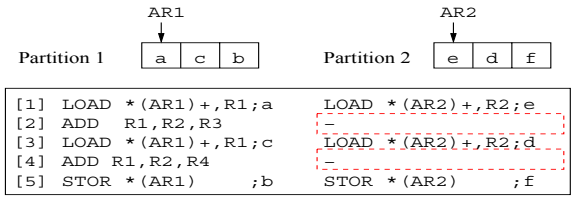

Fig. 3. Schedule with partition of {a,b,c}, {d,e,f} & address assignment.

we partition the variables differently, and with address assignment optimization, we are able to achieve a schedule length of 3. This saving shows that partitioning of variables not only need to consider parallelism, but also need to consider address assignment to maximally leverage register indirect addressing mode provided by AGUs in most DSP processors.

3. BASIC CONCEPTS AND MODELS

The **processor model** we use in this paper is given as follows. There are multiple function units and multiple memory banks in a processor. There is an accumulator in each function unit and an address register associated with each memory bank. Each operation involves the accumulator and, if any, another operand from the memory. Memory access in each memory bank can only occur indirectly via the address register. Furthermore, if an instruction uses an address register AR_j for indirect addressing, then in the same instruction, AR_j can be optionally post-incremented or post-decremented by one without extra cost. If an address register does not point to the desired location, it may be changed by adding or subtracting a constant, using the instructions ADAR or SBAR. In this paper, FN_i is used to denote functional unit i, and AR_j is used to denote the address register for memory bank j. We use *(ARi), *(ARi)+, and *(ARi)- to denote indirect addressing through ARi, indirect addressing with post-increment, and indirect addressing with post-decrement, respectively. This processor model reflects addressing capabilities of most DSPs, and can be easily transformed into other architectures.

Data Flow Graph is used to model loops and is defined as follows. A *Data Flow Graph (DFG)* $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of operation nodes, $E \subseteq V * V$ is the edge set that defines the precedence relations for all nodes in V, $d(e)$ represents the number of delays for an edge e. $t(v)$ represents the execution time for each computation node $v \in V$. Nodes in V can be various operations, such as addition, subtraction, multiplication, logic operation, etc.

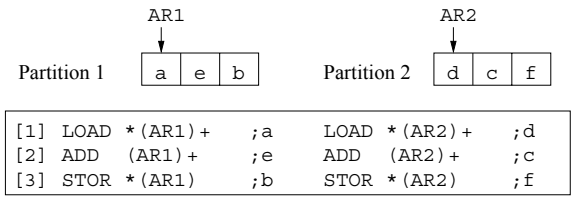

Fig. 4. Schedule with partition of {a,e,b}, {d,c,f} & address assignment.

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 control step assignment, and functional unit allocation. A static schedule must obey the precedence relations of the *directed acyclic graph (DAG)* portion of the respective DFG. The DAG is obtained by removing all edges with delays in the DFG.

4. GRAPH MODELING APPROACH

A graph modeling approach is used in this paper. To explore parallelism for multiple memory bank usage, and to explore serialism for address register usage, two graphs are used to model these two distinct properties. The nodes in both graphs represent all the local variables stored in memory. Partitioning is done by considering information captured in both graphs. Nodes in each final partition correspond to variables assigned to each memory bank.

4.1. Access Graph

The first graph used in this paper is the Access Graph. Access graph was first proposed by Liao et al. in [4]. An access graph $G_{AG} = (V, E, w)$ is an undirected weighted graph, where each node $v \in V$ corresponds to a unique variable and an edge $e \in E$ between node x and y exists with weight $w(e)$ if x and y are adjacent to each other $w(e)$ times in the access sequence. Access sequence is the extracted from a schedule. To build an access graph before scheduling is conducted, we will first build a partial access sequence based on the access sequence within each computation. Basically, a special symbol, "|" is inserted between the access sequences of two neighbor nodes to denote that there is no relation between the two neighbor variables because scheduling is not done yet. For example, " e f d | f a h | b f b " is a partial access sequence, in which "h" and "b" have no relation. With a partial access sequence, we can build the access graph.

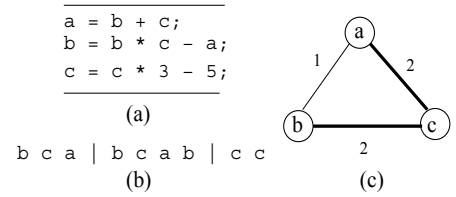

Fig. 5. (a) Example code. (b) Partial Access Sequence. (b) Its corresponding Access Graph.

An example is shown in Figure 5. In this example, Figure 5(a) shows the computation nodes, Figure 5(b) shows the partial access sequence, and Figure 5(c) shows the final access graph, where each node in the access graph represents a variable, and each edge represents the number of times that two variables are accessed next to each other. For example, there is an edge with weight of "2" between "a" and "c" in the access graph because the variable "a" and variable "c" are accessed next to each other two times in the access sequence. The higher the weight of an edge, the more preference we want to give to the variables connected by the edge to be assigned to the same partition. So that more address calculation instructions could be saved by using the address generation unit(AGU).

4.2. VIG Graph

The second graph used in this paper is the VIG graph. A Variable Independence Graph (VIG) is an undirected weighted graph $G_{VIG} = (V, E, w)$ where V is a set of nodes representing variables, $E \subseteq V * V$ 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 $(u, v) \in E$ [8].

Before presenting how a VIG is constructed, we will introduce two important concepts, Mobility Window and Access Set. Given a DFG $G_D = (V, E, d, t)$, a Mobility Window [5] 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 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. The overlap of mobility windows of two nodes indicate the possibility that the nodes could be scheduled in parallel. An example is shown in Figure 6, mobility window for node 10 is $MW(10) = \{0, 1, 2, 3\}$ because node 10 could be scheduled to step 0, 1, 2, or 3. We define the Access Set as the set of memory operation nodes that access a particular variable. For example, ACCESS(a) = {8,9} in Figure 6 because node 8 and 9 access variable a. With Mobility Window and Access Set introduced, we can now introduce the calculation of the priority function for the weight of the edges in the VIG graph.

Given a VIG $G_V = (V, E, w)$, 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 $MW(u) \cap MW(v) \neq 0$, where

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

The weight of each edge in a VIG is a SUM which is related to the possible mobility window overlapping of two variables. The higher the weight of an edge, the more parallelism exist between the two connecting variables. An example of VIG construction is shown in Figure 6.

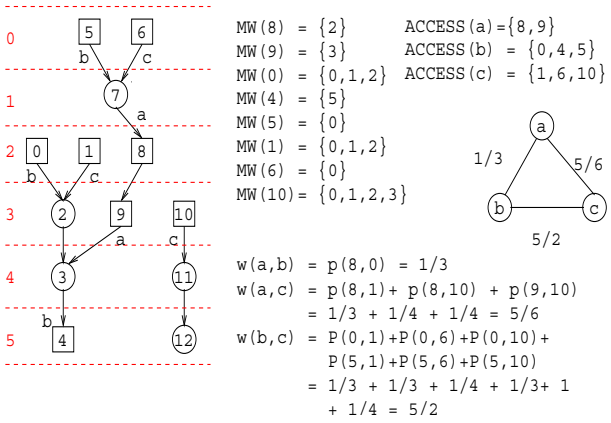

Fig. 6. Example Data Flow Graph. Mobility Window, Access Sets, and Weight Calculation. Final VIG.

5. ALGORITHM

With the Access Graph and the Variable Independence Graph presented in Section 4, we can now model the potential parallelism and potential serialism between program variables. Information modeled in both graphs will be used jointly in partitioning the variables. The problem of variable partitioning of k memory banks is equivalent to the maximum k-cut problem, which is NP-complete [1]. A number of excellent heuristics exist for solving the maximum k-cut problem [1]. To use such heuristics, we need to merge two weights into a single weight for each edge. In this paper, we define the merged weight of an edge $e(i, j)$ as

$$w(i, j) = \alpha * w_{VIG}(i, j) - \beta * w_{AG}(i, j)$$

where α, β are two coefficients representing the trade-offs between parallelism and serialism. $w_{VIG}(i, j)$ is the weight of edge $e(i, j)$ on the VIG graph and $w_{AG}(i, j)$ is the weight of edge $e(i, j)$ on the Access graph. Different coefficient values of α and β explore different trade-offs between parallelism and serialism.

The address assignment sensitive variable partitioning and scheduling algorithm ($AASVP$) is intended to be used in the back end of a DSP compiler to optimize the intermediate code. The $AASVP$ algorithm is shown in Algorithm 5.1.

Algorithm 5.1 Address-Assignment-Sensitive-Variable-Partitioning($AASVP$)

Require: Intermediate codes represented as DFG $G = (V, E, OP, d)$, stepping quantum δ
Ensure: An optimized schedule S and the corresponding α, β.
1. Construct Access Graph G_{AG};
2. Construct Variable Independence Graph G_{VIG} by calculating mobility windows and access sets;
for α = 1 to 0 step by -δ **do**
  for β = 0 to 1 step by δ **do**
    3. Calculate merged weights for each $w(i, j)$ with α, β;
    4. Find the Maximum Cut. Allocate variables to memory banks according to the cut result;
    5. Generate address assignment for each memory bank using mSOA() [7];
    6. Run list scheduling to schedule the input G using address assignment information;
    if $Schedule_length(S) < min_schedule_length$ **then**
      $min_schedule_length = Schedule_length(s)$;
      $S_m \leftarrow S$; $\alpha_m \leftarrow \beta$; $\beta_m \leftarrow \beta$;
    end if
  end for
end for
Output S_m and α_m, β_m;

The basic idea of $AASVP$ algorithm is to first create an Access Graph and a VIG graph based on the input intermediate code, then the merged weight of each edge is calculated based on different α, β values and variables are partitioned using result from max-cut heuristic [6]. Address assignment is generated for each memory bank based on the variables assigned. Finally, a priority based list scheduling is used to schedule the input program based on the variable assignment. During the variable partition phrase, α varies from 1 to 0 and β varies from 0 to 1 by an input quantum δ. At the end, a best schedule that has the minimum schedule length and minimum number of address instruction operations is chosen.

In step 6 of Algorithm 5.1, list scheduling is used with consideration of address assignment obtained. Weighted bipartite matching is used in each step of list scheduling. We repeatedly create a weighted

Bench.	Partition		Parti. with A.A.		AASVP							
	Sch. Len.	Addr. Instr.	Sch. Len.	Addr. Instr.	Sch. Len.	%Sch-P	%Sch-PA	Addr. Instr.	%Addr-P	%Addr-PA	α	β
IIR	25	16	24	13	21	16.0%	12.5%	11	31.3%	15.4%	0.5	0.5
IIR-UF2	47	33	44	29	38	19.1%	13.6%	25	24.2%	13.8%	1	0.3
IIR-UF3	69	48	69	48	64	7.2%	7.2%	47	2.1%	2.1%	1	0.4
4-Latt.	83	62	88	55	71	14.5%	19.3%	49	21.0%	10.9%	0.1	0.9
8-Latt.	132	104	131	98	110	16.7%	16.0%	80	23.1%	18.4%	0.4	0.6
Allpole	68	35	62	27	57	16.2%	8.1%	25	28.6%	7.4%	0.9	0.5
Voltera	91	65	86	56	84	7.7%	2.3%	55	15.4%	1.8%	1	0.8
Ellip	120	79	117	70	99	17.5%	15.4%	67	15.2%	4.3%	1	0.2
Average Reduction (%)						**14.4%**	**11.8%**	–	**20.1%**	**9.3%**		

Table 1. The comparison of schedule length and address operation for VIG partitioning only, partitioning with address assignment, and AASVP scheduling

bipartite graph G_{BM} between the set of available functional units and the set of ready nodes in L_{RD}, and assign nodes based on the min-cost maximum bipartite matching M. In each scheduling step, the weighted bipartite graph, $G_{BM} =< V_{BM}, E_{BM}, W >$, is constructed as follows: $V_{BM} = FU_{ready} \cup L_{RD}$ where $FU_{ready} \subseteq \langle F_1, F_2, \cdots, F_N \rangle$ is the set of currently available FUs and L_{RD} is the set of ready nodes ; for each FU $F_i \in FU_{ready}$ and each node $u \in L_{RD}$, an edge $e(F_i, u)$ is added into E_{BM} and the edge weight is $W(F_i, u) = WCF(AList(F_i), First(u), Priority(u))$, where $AList(F_i)$ is the list of variables last accessed by each FU, $First(u)$ is the first variable that will be accessed by node u. $Priority(u)$ is the longest path from node u to a leaf node. $WCF(AL, y, Z)$ is a function defined as follows:(AL is a list of variables; y is a variable in the address assignment; Z is the priority)

$$WCF(AL, y, Z) = \begin{cases} Z - 2 & \text{y=x, x} \in \text{AL} \\ Z - 1 & \text{y is a neighbor of x, x} \in \text{AL} \\ Z & \text{Otherwise} \end{cases}$$

In this way, the ready nodes with higher priority are considered first. Given the same priority, nodes with address operation savings have more advantage. After all the nodes are scheduled, the schedule S and α, β are recorded. The main body of $AASVP$ algorithm is repeated for different values of α, β. A best schedule is selected which has the minimum schedule length and the minimum number of address instructions.

6. EXPERIMENTS

In this section, we conduct experiments with the $AASVP$ scheduling algorithm on a set of DSPstone benchmarks programs including 4-stage lattice filter, 8-stage lattice filter, differential equation solver, elliptic filter and voltera filter. The experiments are performed on a simulator with the similar architecture as Motorola 56000 DSP. We compare our results with those from VIG partitioning only and partitioning with address assignment directly. The experiments are performed on a PC with a P4 2.1 G processor and 512 MB memory running Red Hat Linux 9.0. In the experiments, the running time of $AASVP$ on each benchmark is less than one minute.

The experimental results for VIG partitioning, partitioning with address assignment and $AASVP$ algorithm, are shown in Table 1. Column "Addr. Instr." presents the number of address instructions and Column "Sch. Len" presents the schedule length obtained from three different scheduling algorithms: scheduling with VIG partitioning only (Field "Partition"), VIG partitioning with address assignment applied afterward (Field "Partition with A.A."), and $AASVP$ algorithm (Field "$AASVP$"). Column "%Sch-P" and "%Sch-PA" under "$AASVP$" represent the percentage of reduction in schedule length compared with partition scheduling and partition with address assignment respectively. Column "%Addr-P" and "%Addr-PA" under "$AASVP$" represent the percentage of reduction in the number of address instructions compared with partition scheduling and partition with address assignment respectively.

Compared to direct application of VIG partitioning scheduling, the reduction in schedule length is 14.4% and the reduction in address instructions is 20.1%. Compared to the partitioning scheduling with address assignment applied afterward, the reduction in schedule length is 11.8% and the reduction in address instructions is 9.3%. We can see that if address assignment technique is applied directly after VIG partitioning and scheduling is applied, we do achieve some reduction in the number of address instructions. However, total schedule length is barely reduced. In $AASVP$, when we consider address assignment together with partitioning and scheduling, we gain substantial reduction in schedule length.

7. CONCLUSION

In this paper, we proposed an algorithm, $AASVP$, that incorporate address assignment into variable partitioning and scheduling, so we can maximize the usage of both address generation units and multiple memory banks. $AASVP$ algorithm can significantly reduce schedule length and address instructions.

8. REFERENCES

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. Complexity and approximation. *Springer Verlag*, 1999.

[2] Y. Choi and T. Kim. Address assignment combined with scheduling in dsp code generation. In *ACM IEEE Design Automation Conference*, pages 225–230, June 2002.

[3] R. Leupers and D. Kotte. Variable partitioning for dual memory bank dsps. *IEEE Intl Conf. on Acoustics, Speech, and Signal Processing*, page 1121.

[4] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18:235–253, May 1996.

[5] G. Micheli. Synthesis and optimization of digital circuits. *McGraw-Hill, Inc.*, 1994.

[6] R. Prim. Shortest connection networks and some generalizatoins. *Bell Systems Technical Journal*, (36).

[7] C. Xue, Z. Shao, Q. Zhuge, B. Xiao, M. Liu, and E. Sha. Optimizing address assignment and scheduling for dsps with multiple functional units. *IEEE Transaction on Circuits and Systems II*, 53(9):976–980, 2006.

[8] Q. Zhuge, E. Sha, B. Xiao, and C. Chantrapornchai. Efficient variable partitioning and scheduling for dsp processors with multiple memory modules. *IEEE Transaction on Signal Processing*, 52:1090–1099, 2003.