Scheduling-based Code Size Reduction in Processors with Indirect Addressing Mode

Sungtaek Lim
Dynalith Systems Co., Ltd.
Taejon 305-701, Korea
+82-42-862-6411
stlim@dynalith.com

Jihong Kim
School of EECS
Seoul National University
Seoul 151-742, Korea
+82-2-880-8792
jihong@davinci.snu.ac.kr

Kiyoung Choi
School of EECS
Seoul National University
Seoul 151-742, Korea
+82-2-880-6768
kchoi@azalea.snu.ac.kr

ABSTRACT
DSPs are typically equipped with indirect addressing modes with auto-increment and auto-decrement, which provide efficient address arithmetic calculations. Such an addressing mode is maximally utilized by careful placement of variables in storage, thereby reducing the amount of address arithmetic instructions. Finding proper placement of variables in storage is called *storage assignment problem* and the result highly depends on the access sequence of variables. This paper suggests statement scheduling as a compiler optimization step to generate a better access sequence. Experimental results show 3.6% improvement on the average over naive storage assignment.

Keywords
Code generation, indirect addressing mode, storage assignment, code size reduction

1. INTRODUCTION
In general, DSPs provide two main addressing modes: direct and indirect. The direct addressing mode uses immediate field in the instruction word to form memory addresses, while in the indirect addressing mode, addresses are read from *address registers*. The addressing mode with address register plus an offset of index is not usually provided by DSPs because the address calculation can increase the execution time significantly. Whereas the indirect addressing mode equipped with auto-increment and auto-decrement that are executed in parallel with main datapath can improve both the size and performance of the code.

The placement of variables in memory has a significant impact on the utilization of indirect addressing mode. Optimizing DSP compilers (such as SPAM [2]) usually defer storage allocation of variablesdown to the code generation step where addressing modes are selected, thereby increasing the opportunities of using efficient auto-increment and auto-decrement operations. The deferred storage allocation is formulated as the *storage assignment problem*. Bartley [1] was the first to address the storage assignment problem. Liao et al. [3] formulated it as *simple offset assignment* (SOA), which is a simplified storage assignment with a single address register. They first built an *access graph* from the access sequence of variables in storage. Then they showed that the SOA problem is equivalent to the *maximum weighted path covering* (MWPC) problem on the access graph and proved that it is NP-complete. They also showed that the SOA solution can be used to solve *general offset assignment* (GOA) problem that handles a fixed number of address registers and suggested heuristics to solve the two problems. Leupers and Marwedel [4] extended the work done by Liao et al. by proposing a tie-breaking heuristic and a variable partitioning strategy to reduce the cost of SOA and GOA solutions respectively.

The above approaches do not attempt to optimize the variable access sequence itself, which can significantly affect the result of storage assignment problem. Rao et al. [5] suggested modifying the variable access sequence using expression tree transformations and formulated it as the *least cost access sequence* (LCAS) problem and developed heuristic algorithms to solve it. They used algebraic transformations (such as commutativity) on the expression tree to modify the order of operands of an instruction.

We can further optimize the access sequence by transforming not only the expression tree of an instruction but also the schedule of instructions. We formulate the problem as statement scheduling and propose an algorithm that solves the problem.

2. SIMPLE OFFSET ASSIGNMENT
Address generation unit (AGU) of a processor that supports indirect addressing mode can usually compute the address used by the next instruction in parallel with the currently executing instruction. AGU is comprised of a file of k address registers (ARs), as well as a file of m *modify registers* (MRs). AR and MR indices are provided by AR and MR pointers respectively, which are the values of either special registers or part of instruction words. According to the decoded instruction, AGU generates an address based upon an AR, which is incremented or decremented by a constant or by the value of an MR. The range of the constant is represented by r. Thus k, m, and r determine the configuration of the AGU.

When a program accesses a series of variables in memory, if the stride of the addresses is greater than the range, AR or MR should be reloaded with an immediate value. This additional instruction causes code size overhead of indirect addressing mode. *Offset assignment* (OA) is the problem of finding proper memory layout of variables to reduce the occurrence of strides larger than the

CODES 01 Copenhagen Denmark

range supported by the AGU. The offset assignment problem is classified according to the AGU parameters k, m, and r and represented by (k, m, r)-OA [7].

(1, 0, 1)-OA is offset assignment on a processor with only one address register, no modify register, and auto-increment and auto-decrement range of 1. Figure 1 (a) shows an example of (1, 0, 1)-OA, which results in nine instructions. By re-arranging the variable in memory as shown in Figure 1 (b), we can reduce the number of instructions down to seven.

Bartley [1] modeled the (1, 0, 1)-OA problem as an undirected edge-weighted *access graph* G(V,E,W), where V models the set of variables and E models the set of transitions between variables. For each edge e=(v1, v2) in E, the weight w(e) is the number of transitions from v1 to v2 or vice versa in the access sequence. Large weight of edge (v1, v2) means v1 and v2 are frequently accessed in sequence so the two variables should be placed into neighboring memory locations because accessing v1 after or before v2 is supported by AGU's auto-increment or auto-decrement. Each offset assignment corresponds to a

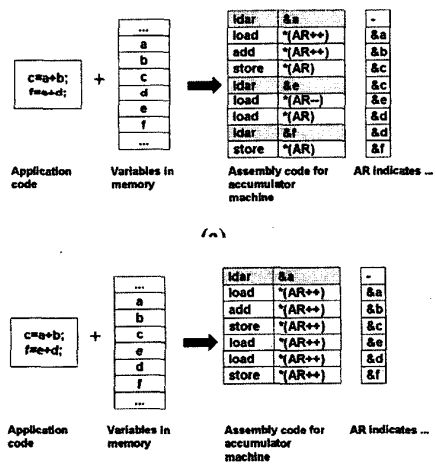

Figure 1. Examples of (1, 0, 1)-OA.

Hamiltonian path in G, i.e. a path that traverses all nodes just once.

It is obvious that an optimum offset assignment corresponds to a maximum weighted Hamiltonian path in G. This problem is called *maximum weighted path covering* (MWPC). Liao et al. [3] showed that the offset assignment problem is NP-complete even for the simple case of (1, 0, 1)-OA and presented a heuristic based on the access graph model to solve it. They generalized it to (k, 0, 1)-OA by partitioning G into k subgraphs each of which is covered by an AR.

3. STATEMENT SCHEDULING

First we assume an accumulator machine with one accumulator register and load/store instructions and consider simple offset assignment problem. Figure 2 (a) shows an application code, the corresponding access sequence, the access graph, and the MWPC solution. Edges that are not covered by the MWPC solution represent transitions requiring additional instructions. The weight of such an edge corresponds to the number of additional instructions. Hence the cost of the MWPC solution is the sum of the uncovered edges' weights, and the cost of the solution in Figure 2 (a) is 3.

Observing that the two statements e = d + 2 and a = b have no data/control dependency to each other, we can change their order without affecting the functionality. Figure 2 (b) shows a modified code and the corresponding access sequence, access graph, and MWPC solution with cost of 1. This way, proper statement scheduling can result in lower MWPC cost and this is the motivation of our work.

We formulate the problem as follows. Given a basic block[1] of an application code, schedule the statements in such a way that the cost of the MWPC solution for the corresponding access graph is minimized. We use the term *statement* to denote a schedulable element of compiler's intermediate representation–*medium level intermediate representation* (MIR) [6]–for a given application code. One statement corresponds to one MIR instruction.

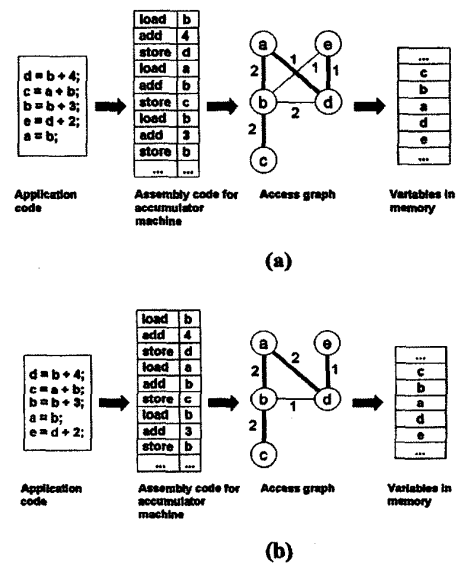

Figure 2. Statement scheduling.

We perform scheduling[2] based on the MIR, which comprises MIR instructions as well as the dependencies between the instructions. *Dependence DAG* is a neat way to show an MIR and we will use it

[1] Currently, we do not allow branches and perform the optimization within a basic block.

[2] We assume all local variables reside not in registers but in memory. Hence instruction scheduling considering register spill is not of our concern.

hereafter. Each node means a statement (or MIR instruction) and each directed edge means dependency between two statements. Figure 3 illustrates the application code in Figure 2 using the dependence DAG.

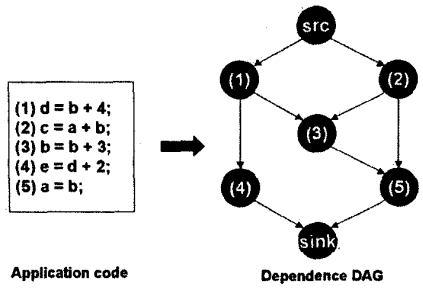

Figure 3. Dependence DAG.

4. ALGORITHMS

We need to find a statement schedule that gives the minimum cost. However, to compute the cost for a given schedule, we need to solve the MWPC problem which itself is NP-complete. We device a heuristics based on the observation that the access graph of Figure 2 (b) has fewer edges than that of Figure 2 (a). When an access graph becomes sparser, maximum weighted path covering tends to result in less cost because the number of edges to be covered are reduced and the edge weights tend to be concentrated on some small set of edges. So our algorithm aims to generate the sparsest access graph.

4.1 List Scheduling

The proposed list scheduling algorithm constructs an access graph by selecting a statement to be scheduled from the dependence DAG and updating the access graph with the new transitions. The update may add new edges, add new vertices and/or increase edge weights. It implements a greedy heuristic that selects a statement that adds least new edges at that schedule step.

Figure 4 (a) shows a statement schedule of the example in Figure 3. Let's assume that statement (1), (2), and (3) are already scheduled and (4) and (5) are not scheduled yet. If we schedule statement (4) first, two new edges will be added to the access graph as shown in Figure 4 (b). Then statement (5) is scheduled and the cost is 3 as shown in Figure 2 (a). If we schedule statement (5) first, one new edge will be added to the access graph as shown in Figure 4 (c). Then statement (4) is scheduled and the cost is 1 as shown in Figure 2 (b). This is the case where list scheduling leads to an optimum solution.

4.2 Exhaustive Search

Exhaustive search method examines all possible statement schedules and finds an optimal schedule, which generates sparsest access graph. Branch pruning can be used to accelerate the exhaustive search. At each schedule step, it selects a statement for the next schedule, updates the edge count, and compares it to the optimal cost found up to that time.

4.3 Hybrid Algorithm

List scheduling executes fast but as is usual for a greedy algorithm, can lead to a local optimum. And it is not easy to set good criteria for tie break. On the contrary, exhaustive search guarantees an

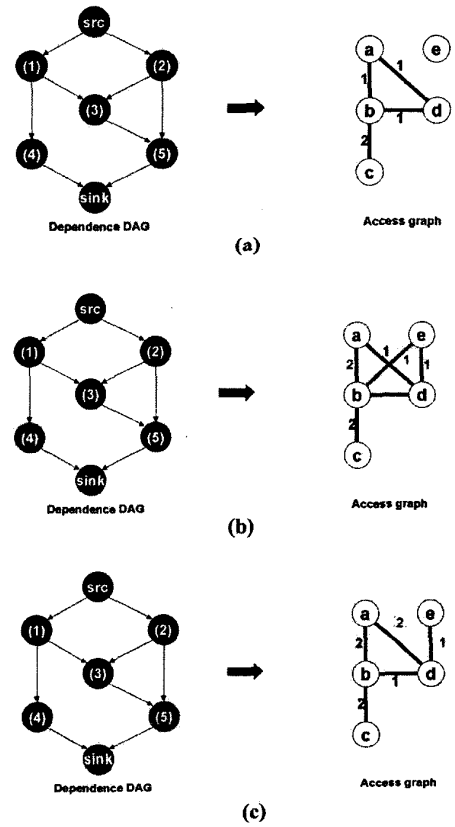

Figure 4. List scheduling.

optimal solution but runs in time exponential to the size of the problem. Even the pruning method does not guarantee to improve the execution time. More aggressive pruning is needed to improve the execution time. The hybrid algorithm confines the exhaustive search to the successors of the statement that is scheduled most recently. Other statements are excluded because they tend to generate new edges in the access graph. For example, consider the dependence DAGs shown in Figure 5. Figure 5 (a) illustrates that the list scheduling method selects one statement from the candidates for the next schedule. Figure 5 (b) illustrates that the exhaustive search considers all candidate statements for the next schedule. However, the hybrid algorithm confines the search to the reduced number of candidates as shown in Figure 5 (c).

5. EXPERIMENTAL RESULTS

We implemented the three algorithms on SPAM compiler middle- and back-end targeting Texas Instruments' TMS320C25 DSP. The

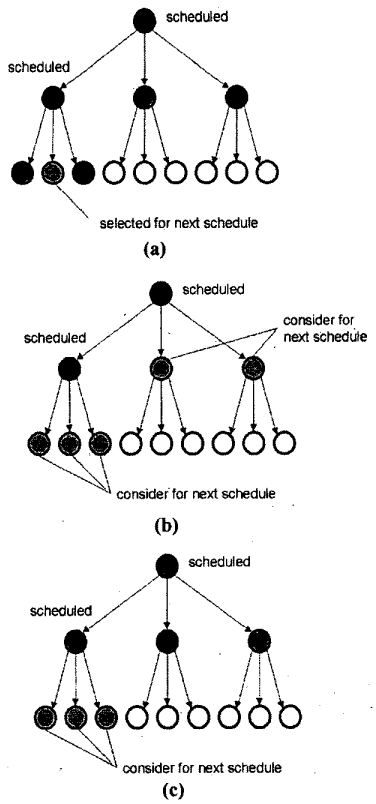

Figure 5. Pruning the search space for statement scheduling.

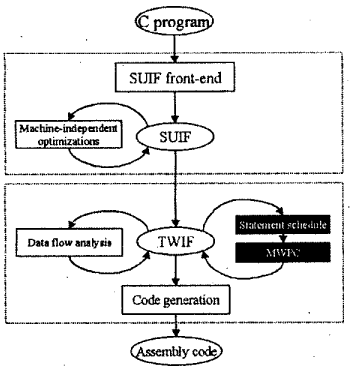

Figure 6. Overall flow of the optimizing compiler.

overall flow of the optimizing compiler is shown in Figure 6.

Table 1 shows the code size reduction in number of words. SOA means the size of code obtained by MWPC but without statement scheduling. The gain of SS gives the code size reduction obtained by the hybrid method with respect to the original code size. It shows the average gain of 3.6%. We could not obtain the result of the exhaustive search on biquad_N_sections due to the enormous amount of running time but the hybrid method found a solution. Execution times of the proposed three algorithms are shown in table 2.

To compare the effect of statement scheduling with that of expression tree transformation, we quoted the gain from [5]. Sometimes expression tree transformation shows better result. But the two approaches are not totally exclusive. We expect additional gain by extending our approach from MIR- to LIR-based scheduling to subsume the effect of expression tree transformation and we set aside it for future work.

Table 1: Code Size Reduction

	SOA	list schedule	exhaustive	hybrid	% gain of SS	% gain of ETT
complex_multiply	34	32	32	32	5.882	2.3
convolution	92	90	86	87	5.435	5.81
dot_product	75	71	71	71	5.333	0
fir	138	134	129	132	4.348	7.03
biquad_N_sections	222	218	N/A	214	3.604	N/A
matrix2	287	278	275	277	3.484	0.74
matrix1x3	71	69	69	69	2.817	N/A
fir2dim	365	361	361	361	1.096	2.4
biquad_one_section	75	75	75	75	0	1.74
average					3.6	2.9

SS : statement scheduling with hybrid method

ETT : expression tree transformation [5]

Table 2: Execution Time (seconds)

	list scheduled	exhaustive	hybrid
complex_multiply	0	0.74	0.67
convolution	0	0.61	0.61
dot_product	0	0.85	0.85
fir	0	28.91	8.71
biquad_N_sections	0.01	N/A	171.1
matrix2	0	6.66	0.83
matrix1x3	0	0.86	0.82
fir2dim	0.01	3.06	2.74
biquad_one_section	0	0.01	0.01

We calculated the effect of code size reduction on the performance for the case of complex_multiply and dot_product, which resulted in 5.88% and 4.94% improvement in cycle count respectively.

6. CONCLUSION

We showed that statement scheduling can further improve the result of simple offset assignment. Among the three scheduling algorithms proposed in this paper, the hybrid method results in the cost generally lower than the list scheduling method and runs faster than the exhaustive search.

Generalization of statement scheduling to solve GOA does not seem to be difficult. Using variable partitioning method, we can also partition dependence DAG into many subgraphs and execute the proposed scheduling algorithm assuming one address register for each subgraph.

The proposed scheduling is done at medium-level intermediate representation (MIR) in SPAM compiler. But if we target low-level intermediate representation, we may be able to obtain further improvement.

REFERENCES

[1] D.H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Software Practice and Experience*, vol. 22 (2), 1992

[2] *SPAM Compiler Users Manual*, SPAM Research group, Princeton University, 1997

[3] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage assignment to decrease code size," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995

[4] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," *Int. Conference on Computer-Aided Design (ICCAD)*, 1996

[5] A. Rao and S. Pande, "Storage assignment using expression tree transformations to generate compact and efficient DSP code," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999

[6] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997

[7] R. Leupers, *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000