Address Assignment Combined with Scheduling in DSP Code Generation

Yoonseo Choi and Taewhan Kim
Department of Electrical Engineering & Computer Science
and Advanced Information Technology Research Center(AITrc)
Korea Advanced Institute of Science and Technology, KOREA
yschoi@jupiter.kaist.ac.kr tkim@cs.kaist.ac.kr

Abstract – One of the important issues in embedded system design is to optimize program code for the microprocessor to be stored in ROM. In this paper, we propose an integrated approach to the DSP address code generation problem for minimizing the number of addressing instructions. Unlike previous works in which code scheduling and offset assignment are performed sequentially without any interaction between them, our work *tightly couples offset assignment problem with code scheduling* to exploit scheduling on minimizing addressing instructions more effectively. We accomplish this by developing a fast but accurate two-phase procedure which, for a sequence of code schedules, finds a sequence of memory layouts with minimum addressing instructions. Experimental results with benchmark DSP programs show improvements of 13%-33% in the address code size over *Solve-SOA/GOA* [7].

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: [Signal processing systems]

General Terms

Algorithms, Performance

Keywords

Offset assignment, Scheduling, Code Generation

1. INTRODUCTION AND RELATED WORK

The increasing complexity of designing embedded VLSI systems has made the traditional DSP compilers unable to meet the very tight constraints of code size for on-chip program and real-time performance. Unfortunately, due to irregular architectures such as non-homogeneous register sets, specialized functional units and registers, even compilers available for commercial DSP processors generate very inefficient code in terms of size and performance.[1, 2].

Storage assignment, i.e., optimization of memory layout for program variables is an important part of code generation. It was shown that, for a set of programs in MediaBench [4] that were compiled for Motorola *DSP56000* family, more than 55% of the instructions involve address registers.[3] Consequently, optimizing address assignment could lead to a significant reduction on code size and program execution time.

DAC 2002 June 10-14, 2002, New Orleans, Louisiana, USA

Many architectures (e.g., the VAX, TI TMS320C2x DSP family, most embedded controllers) provide indirect addressing modes with auto-increment/decrement address arithmetic. These architectures also provide a set of dedicated address generation units (AGUs) that perform fast address computation in parallel to the central data path and contain a separate adder/subtractor for performing next-address computations. Thus, a high instruction-level parallelism can be achieved by generating many auto-increment/ decrement addressing modes to access variables, which requires a careful placement of variables in memory. Consequently, contrary to the traditional compilers, DSP compilers should carefully determine the relative location of variables in memory.

The storage assignment with a single address register in AGU (the *simple offset assignment* problem, SOA) was first studied by Bartley [5] and Liao *et. al* [1, 6]. They modeled the problem as a graph theoretic optimization problem. Liao *et. al* showed that the SOA problem is equivalent to the *maximum-weighted Hamiltonian path cover* (MWPC) problem and proved that it is NP-complete. They proposed a heuristic to solve SOA based on the Kruskal's maximum spanning tree algorithm. Liao *et. al* [1, 6] also extended the approach to the case with multiple address registers in AGU (the *general offset assignment* problem, GOA). Leupers and Marwedel [7] have extended the work done by Liao *et. al* by proposing a *tie-breaking* heuristic and a variable partitioning method to improve the quality of SOA/GOA solution. Leupers and David [8] have solved GOA problem with arbitrary register file sizes and auto-increment ranges. Sudarsanam *et. al* [9] take into account the utilization of auto-increment with increments varying from $-l$ to $+l$. Gebotys [10] modeled the problem of assigning address registers to every variable access in the code given a fixed memory layout as a network flow problem and solved it optimally.

All of the previous approaches [1, 5, 6, 7, 8, 9] have addressed the SOA/GOA problem as a separate code optimization problem for a given exact access sequence of the program variables. However, since the access sequence of the variables is one of the most critical factors which affect the quality of SOA/GOA solutions, the memory layouts produced by the previous approaches will be SOA/GOA solutions that were locally optimized. In this context, we study the problem of integrating the SOA/GOA with code scheduling to exploit the effect of scheduling on minimizing the size of address code (in assembly) more fully and effectively. Our solution is naturally extended to incorporate reordering of input-operands of commutative operations[1] (e.g., $a = b + c$). To our knowledge, the work done by Rao and Pande [11] has addressed the optimization of access sequence of the variables by a (local) reordering of commuta-

[1]The optimization is performed at the level of an intermediate representation like three-address instruction that has two source operands.

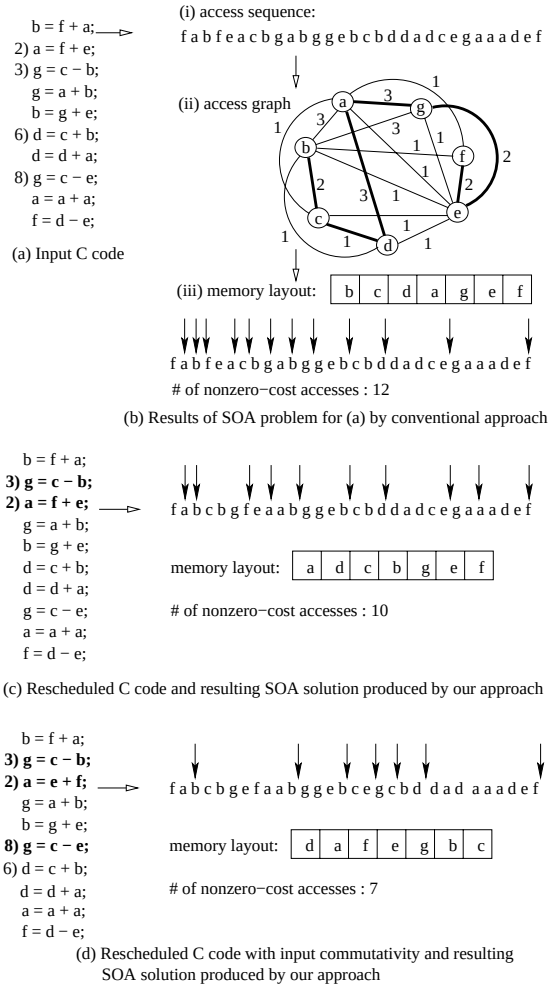


Figure 1: Motivational example illustrating the effectiveness of code scheduling on address assignment.

tive input-operands. However, their solution is confined to the SOA problem only. Lim *et. al* [12] addressed scheduling effect on the SOA problem. Their approach aimed to make graph sparser by an exhaustive search algorithm with pruning techniques. However, it is not true that sparser graphs always lead to cheaper MWPC cost than that of denser graphs.

The key contribution of our work is an effective integration of code scheduling (together with input-operand permutation) which we call Sch-SOA/GOA, into the SOA/GOA heuristics *Solve-SOA/GOA* [1, 6, 7].

2. MOTIVATING EXAMPLE

As an example illustrating the effectiveness of code scheduling on the problem of assigning frame-relative offsets to program variables, consider the C program in Figure 1(a). Suppose we want to solve a simple offset assignment (SOA) problem. That is, we assumed to have a target architecture with a single address register (AR) with only the indirect and auto-increment/decrement address modes. Figure 1(b-i) shows the access sequence of the variables corresponding to the input C code in Figure 1(a).

Liao *et. al* [1] modeled SOA by an undirected weighted graph, called *access graph*, $G(V,E)$ where each node $v \in V$ corresponds to a unique variable and an edge $e \in E$ between nodes x and y exists with weight $w(e)$ if x and y are adjacent to each other $w(e)$ times in the access sequence. The access graph corresponding to the access sequence in Figure 1(b-i) is shown in Figure 1(b-ii).[2] An optimal offset assignment is to find a path cover P in $G(V,E)$ that minimizes the quantity (i.e., total address arithmetic instructions):

$$C(G,P) = \sum_{\forall e \ of \ edges \notin P} w(e) \qquad (1)$$

Finding a maximum weighted path cover (MWPC) minimizes the quantity of C. The heavy lines in Figure 1(b-ii) beginning from b and ending at f form a path cover, The path was obtained by applying Liao *et al*.'s SOA solution [1] with Leupers and Marwedel's *tie-breaking* rule [7]. As a result, the corresponding memory layout is shown in Figure 1(b-iii) where $C = w(a,b) + w(a,c) + w(a,e) + w(a,f) + w(b,d) + \cdots + w(e,g) = 12$.

On the other hand, the left side of Figure 1(c) shows a rescheduled C code where the highlighted statements indicate the change of schedule. The access sequence for the rescheduled code and the resulting memory layout are shown on the right side of Figure 1(c). The total number of nonzero-cost addressing instructions is reduced to 10. Finally, Figure 1(d) shows the further optimized access sequence and memory layout obtained when scheduling is considered together with input permutation for commutative operations. The number of nonzero-cost addressing instructions is then reduced to 7, which is 41% improvement over the schedule in Figure 1(a).

The example in Figure 1 clearly reveals that the offset assignment problem is very sensitive to the access sequence of the variables. Since the existing compilers have designed the offset assignment as a post-pass optimization (after code generation) they failed to globally optimize the access sequence. To overcome this limitation, we propose *a new offset assignment algorithm tightly coupled with code scheduling* to exploit the effect of scheduling on minimizing address arithmetic instructions more fully and effectively. We then extend the scheduling problem to include input permutation for commutative operations to enhance the quality of the assignment further.

3. A UNIFICATION OF CODE SCHEDULING AND ADDRESS ASSIGNMENT

Based on the motivating example in section 2, we propose an effective two-phase algorithm for address assignment integrated with code scheduling. First, we describe an overview of the proposed algorithm in section 3.1, followed by the detailed description on our two-phase algorithm: (Phase 1) *a generation of initial schedule and address assignment* in section 3.2, and (Phase 2) *a stepwise refinement of the solution* obtained in Phase 1 in section 3.3.

3.1 An overview

The input C program is composed of basic blocks, which are given as a partially ordered list of code operations. Our current work focuses on basic block-level address code optimization The input to our algorithm is *dependency graph* of operations (i.e., instructions), that is an intermediate representation of a basic block. A node in the dependency graph represents an operation, and an arc from node *a* to *b* indicates that *a* must precede *b* in execution.

The objective of our algorithm is to schedule operations in the dependency graph and determines a complete sequence of variable accesses, so that the size of resulting address code is minimized.

Figure 2 summarizes the flow of our offset assignment algorithm combined with scheduling/operand-permutation, called Sch-SOA/GOA. The proposed algorithm is performed in two-steps:

[2]The access sequence involved in operation $x = y \ op \ z$ is '*yzx*' where *op* represents an arithmetic operation. The edges with zero weight are not shown in the access graphs for brevity.

Sch-SOA/GOA: **schedule-driven SOA/GOA** (*Dependency_graph*)
/* Phase 1 */
• Generate access sequence S_{init} and schedule SCH_{init} by calling *Sch-SOA/GOA-init*(*Dependency_graph*); /* sec. 3.2 */
• Generate access graph G from S_{init};
• Apply *Solve-SOA/GOA* to G and determine C in *Eq.*(1).;
/* Phase 2 */
• Set $COST_{min} = C$;
• Set $SCH_{best} = SCH_{init}$;
while ($COST_{min}$ is updated) {
  • Set $COST_{current} = COST_{min}$;
  • Set $SCH_{current} = SCH_{best}$;
  while (there is a 'reschedulable' operation) { /* constrained by data-flow dependencies */
    foreach 'reschedulable' operation (to execution step j) {
      • Reschedule the operation;
      • Update the previous path cover solution to account for the reschedule; /* sec. 3.3 */
      if (operation is commutative) {
        • Update the path cover solution by exploiting input commutativity; /* sec. 3.3 */
      }**endif**
      • Compute C for the path cover of the reschedule;
      • Undo the reschedule;
    }**endforeach**
    • Reschedule the operation with the smallest C;
    if ($C < COST_{min}$) {
      • Update $COST_{min}$ to current C, and update SCH_{best} accordingly;
    }**endif**
    • Lock the operation at the execution step;
  }**endwhile**
  • Unlock all the operations;
}**endwhile**
• **return** (*Solve-SOA/GOA*(G of SCH_{best}));

Figure 2: The overall flow of the proposed schedule-driven SOA/GOA algorithm.

(Phase 1) generating an initial schedule and address assignment from the dependency graph of each basic block derived from the input C program, and (Phase 2) iteratively improving the SOA/GOA solution obtained in Phase 1 at the outer *while-loop*. An operation scheduled at an execution step is called "reschedulable" to another execution step, say j, if scheduling the operation at j does not violate the data-flow dependency constraint. For every reschedulable operation, its C value is computed. Among the operations, the operation with the least value of C is selected, and rescheduled it to the corresponding execution step. Once the operation is rescheduled, it will be locked at that clock step for the rest of the execution at the inner *while-loop*. The outer *while-loop* continues until it is not able to generate a schedule and SOA/GOA solution whose C value is less than that of the minimal C found so far during the previous iterations.

The time complexity of Sch-SOA/GOA depends on the time to generate a path cover solution for a trial of rescheduling at the *for-loop*, which will be discussed in detail in section 3.3.

3.2 Initial Schedule and Address Assignment

It is important to produce a good initial schedule and address assignment because the quality of the final address assignment by reschedule is greatly affected by its initial solution. We focus our discussion on generating an initial schedule and its SOA solution for the reason that our extension to GOA solution is rather straightforward with minor modifications.

We construct an access sequence incrementally, one execution step at a time, from the first execution step to the last. An operation in the dependency graph is said to be a *ready operation* for a certain execution step if all of its predecessors have already been

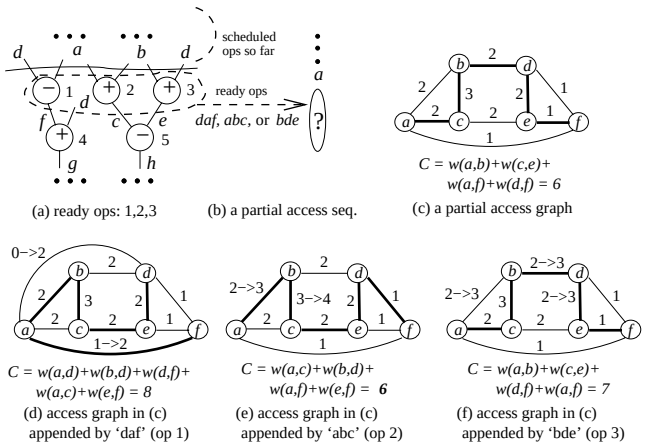


Figure 3: An example of showing the selection of operation, among ready operations, to minimize C in $Eq.(1)$.

scheduled in the previous execution steps. At each iteration, our algorithm selects the most "promising" operation and schedules it at the current execution step, and appends its variable accesses to the end of the current partial access sequence. After the (partial) access sequence is augmented according to the selection of an operation from the ready operations, the algorithm repeats for the next execution step until all the operations in the dependency graph have been scheduled for execution.

The algorithm selects, among ready operations, the operation with minimum additional cost of C in $Eq.(1)$ for the augmented access graph. For example, suppose we have already generated an access sequence (ended by accessing variable *a*) partially up through execution step $t-1$, as depicted in Figure 3(a). Now, we want to select an operation to be executed in execution step t, and accordingly, append the variable accesses to the current partial access sequence '$\cdots a$' (denoted by S_{curr}), as indicated in Figure 3(b). Suppose we have an access graph (denoted by G_{curr}), shown in Figure 3(c) corresponding to S_{curr}. The path (denoted by P_{curr}), with heavy lines indicates a solution of maximum weighted path cover obtained by applying heuristic *Solve-SOA* [7]. Then, $C(G_{curr}, P_{curr}) = 6$ as shown in Figure 3(c).

Let us now consider all possible schedules of ready operations at the next execution step. Figures 3(d), (e) and (f) show the access graphs with augmented weights and the computation of their C values when operations 1, 2 and 3 in Figures 3(a) are selected for execution at execution step t, respectively. Consequently, we choose operation 2 because it results in the minimum increase of C value. Thus, we attach the access sequence '*abc*' (for operation 2) to S_{curr}, resulting in '$\cdots aabc$'. The ready operations are then updated and the process repeats until there is no more operations in the list.

Figure 4 summarizes the flow of our SOA algorithm combined with code scheduling, called Sch-SOA-init. It is a variant of list scheduling. For each trial of scheduling ready operations, a maximum weighted path cover for the updated access graph is obtained by applying *Solve-SOA* [7], the time complexity of which is $O(m \cdot \log m)$ where m is the number of edges in access graph. Thus, the total time of our algorithm is bounded by $O(n^2 \cdot m \cdot \log m)$ where n is the number of operations since the outer *for-loop* is executed n times and at each time, at most n ready operations are tested using *Solve-SOA*. However, since the number of ready operations is very small in practice, the run time is actually bounded by $O(n \cdot m \cdot \log m)$.

3.3 Iterative Improvement Techniques

The core of our algorithm is to generate, efficiently but accurately,

Sch-SOA-init**: schedule-driven initial SOA** (*Dependency_graph*)
- Set n = the number of operations in *Dependency_graph*;
- Set R = the operations in *Dependency_graph* with no predecessors;
- Set $S = \phi$;

for t = 1 to n {
  foreach $op_i \in R$ {
    - Schedule op_i to execution step t;
    - Update access graph G by augmenting the weights associated with the variable access transitions in op_i;
    - Find a maximum weighted path cover, P, for G;
    - Compute $C(G,P)$ in $Eq.(1)$;
    - Undo the schedule and G;

  } **endforeach**
  - Schedule op_i with the smallest $C(G,P)$ to t;
  - Update access sequence S by appending access sequence for op_i;
  - Update access graph G and ready operation set R accordingly;

} **endfor**
- **return** (*S*, *Schedule*);

Figure 4: The proposed algorithm for generating an initial SOA solution.

the address assignment solution (i.e., path cover which minimizes the quantity of C in *Eq.*(1) when an operation is rescheduled from execution step i to j. Clearly, the rescheduling changes the the access sequence, and thus, changes weights of some edges in the access graph. A path cover solution can be obtained simply by applying *Solve-SOA/GOA* [7]. However, this definitely requires an excessive run time when the problem size is large. However, from the fact that each reschedule changes the execution order locally, causing a small change of edge weights in the access graph, and a path cover solution with a minimum C value for the previous schedule has already been known, it is natural to ask whether there is *a way to find a path cover with a minimum C value for the current schedule rapidly by exploiting the previous path cover solution.* To do this, we develop a fast and comprehensive path cover computation procedure based on the previous solution.

Figure 5(a) shows an example of code rescheduling. According to the reschedule, as depicted in Figure 5(b), the weights of edges $(x1,x2)$, $(x3,x4)$ and $(x6,x7)$ are decremented by 1, and the weights of $(x1,x4)$, $(x3,x7)$ and $(x2,x6)$ are incremented by 1. Thus, we can easily find that the number of edge weights to update by a reschedule of an operation is bounded by a constant (=6), independently of the size of access graph. Further, the amount of change for each edge is in [-3, +3].[3]

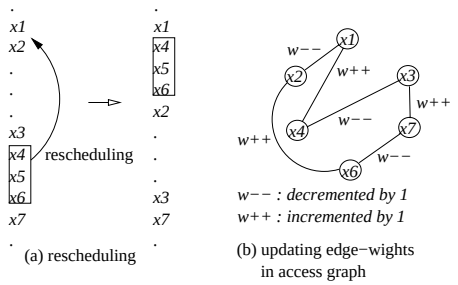


Figure 5: An example of operation rescheduling and update of edge weights.

Schedule-SOA: Let P_{old} denote the path cover (i.e. *solve-SOA* solution) in access graph $G(V,E_{old})$. Suppose some of edge weights in $G(V,E_{old})$ are updated according to a reschedule of an operation. Let $G(V,E_{new})$ denote the updated access graph.

The problem is to find a new path cover, P_{new}, for $G(V,E_{new})$ that minimizes the quantity of C in *Eq.*(1). We determine P_{new} according to the following rules:

- **Case 1**: $C(G(V,E_{old}),P_{old}) > C(G(V,E_{new}),P_{old})$.
  Note that $C(G(V,E_{old}),P_{old})$ is the C value for the access graph before the reschedule using path cover P_{old}, and $C(G(V,E_{new}),P_{old})$ is the C value after the reschedule using the same path cover P_{old}. Case 1 belongs to the case that the C value obtained by using P_{old} for the updated (new) G is smaller than that by P_{old} for the old G. In this situation, we simply use the P_{old} as a new path cover for $G(V,E_{new})$ as long as the C value using P_{old} for $G(V,E_{new})$ decreases. Since the number of edges whose weights are updated is at most 6, computing the C value for $G(V,E_{new})$ using P_{old}, can be done in $O(1)$. In Figure 6, we use P_{old} as a path cover, P_{new}, for the new G, since the C value for the new G is reduced even when P_{old} is used.

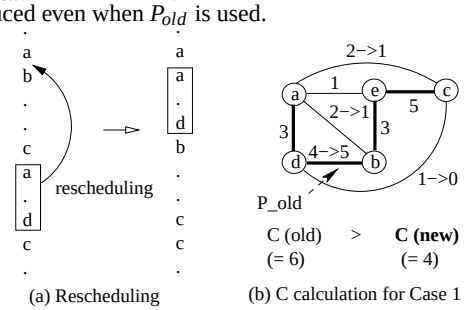


Figure 6: Determination of a path cover for an updated G in Case 1.

The described procedure for determining a path cover for $G(V,E_{new})$ in Case 1 has an optimal property for a special case.

Property 1 Let P_{old} be an optimal path cover for $G(V,E_{old})$, and $w_o(e)$ and $w_n(e)$ denote the weights of edge e in $G(V,E_{old})$ and $G(V,E_{new})$, respectively. We define $\delta(e)$ to be $w_o(e) - w_n(e)$ if $w_n(e) < w_o(e)$, and 0 otherwise. Then, P_{old} is an optimal path cover for $G(V,E_{new})$ if the following conditions are satisfied.

(1) $w_o(e) \geq w_n(e), \forall e \notin P_{old}$ and
(2) $\sum_{e \notin P_{old}} \delta(e) > \sum_{e \in P_{old}} \delta(e)$

proof Let P_i be a path cover for $G(V,E_{new})$ which satisfies $C(G(V,E_{old}),P_i) > C(G(V,E_{old},P_{old})$. We define edge sets

$$S_1 = \{e | e \in E, e \notin P_{old}, e \notin P_i\},$$
$$S_2 = \{e | e \in E, e \in P_{old}, e \notin P_i\},$$
$$S_3 = \{e | e \in E, e \notin P_{old}, e \in P_i\}.$$

We first show the inequality

$$\sum_{e \in S_2} w_n(e) \geq \sum_{e \in S_3} w_n(e). \quad (2)$$

From the assumption that P_{old} is an optimal path cover for $G(V,E_{old})$,

$$\sum_{e \in S_2} w_o(e) > \sum_{e \in S_3} w_o(e).$$

Thus,

$$\sum_{e \in S_2} w_o(e) - 1 \geq \sum_{e \in S_3} w_o(e). \quad (3)$$

According to *condition 2* and the fact that the total sum of the decreases of edge weights (from E_{old} to E_{new}) is at most 3, it is true that $\sum_{e \in P_{old}} \delta(e) \leq 1$, from which

$$\sum_{e \in S_2} w_n(e) \geq \sum_{e \in S_2} w_o(e) - 1. \quad (4)$$

[3] '3' occurs when $x1$=$x2$, $x3$=$x4$, $x6$=$x7$ in Figure 5 whereas '-3' occurs when $x1$=$x4$, $x3$=$x7$, $x2$=$x6$.

Thus, from Eq.(3) and Eq.(4) and $\sum_{e\in S_3} w_o(e) \geq \sum_{e\in S_3} w_n(e)$ according to *condition 1*, we have

$$\sum_{e\in S_2} w_n(e) \geq \sum_{e\in S_3} w_o(e) \geq \sum_{e\in S_3} w_n(e). \quad (5)$$

To show $C(G(V,E_{new}),P_i) \geq C(G(V,E_{new}),P_{old})$, by the definitions of S_1, S_2 and S_3, $S_1 \cup S_2 = \{e|e \in E, e \notin P_i\}$ and $S_1 \cup S_3 = \{e|e \in E, e \notin P_{old}\}$, and using the inequality in *Eq.*(5), we derive

$$\begin{aligned} C(G(V,E_{new}),P_i) &= \textstyle\sum_{e\in S_1} w_n(e) + \sum_{e\in S_2} w_n(e) \\ &\geq \textstyle\sum_{e\in S_1} w_n(e) + \sum_{e\in S_3} w_n(e) \\ &= C(G(V,E_{new}),P_{old}). \end{aligned}$$ □

- **Case 2**: $C(G(V,E_{old}),P_{old}) \leq C(G(V,E_{new}),P_{old})$.
  This is the case where using P_{old} for $G(V,E_{new})$ does not reduce the C value. Thus, we find another path cover by applying *Solve-SOA*. If we maintain a sorted edge-list for $G(V,E_{old})$ according to the edge weights, finding a path cover for $G(V,E_{new})$ using *Solve-SOA*, can be completed in $O(e)$ where e is the number of edges in $G(V,E_{new})$. In Figure 7, since the C value for $G(V,E_{new})$ using P_{old} is not smaller than that for $G(V,E_{old})$ using P_{old}, we compute P_{new} for $G(V,E_{new})$ applying *Solve-SOA*, generating a reduced C value (i.e., from 6 to 5) for $G(V,E_{new})$.

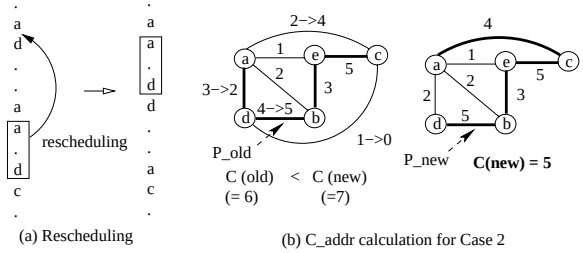


Figure 7: Determination of a path cover for an updated G in Case 2.

Schedule-GOA: GOA is the generalization of SOA towards a number k (> 1) of ARs. A reasonable approach proposed by Liao *et al*. [1] and Leupers and Marwedel [7] is to partition the variables into k disjoint subsets $V_1, \cdots, V_k$, for each of which a distinct AR is used. The objective is to minimize

$$C^{tot} = \sum_{\forall V_i,\ i=1,\cdots k} C_i \quad (6)$$

where C_i is the quantity of C in *Eq.*(1) for the access subgraph containing the variables only in V_i. Then, the problem is reduced to k SOA problems.

At each trial of rescheduling, we selectively apply *Solve-GOA* [7] to the updated G. Let G be the access graph for the previous schedule, and $G_1, \cdots, G_k$ be the access subgraph corresponding to the partitioned vertex sets $V_1, \cdots, V_k$, respectively. Also, let $P_1, \cdots, P_k$ be the path covers for $V_1, \cdots, V_k$, respectively. From the change of weights in G by the current reschedule, we calculate C for each of G_i, $i = 1,\cdots,k$ using path cover P_i. According to the change of the value of C, we partition the variables in $V_1, \cdots, V_k$ into two subsets, V_{good} and V_{bad}: If there is a reduction on the value of C, all the variables in the corresponding V_i are assigned to V_{good}. Otherwise, the variables are assigned to V_{bad}. Then, for each access subgraph whose variables are all in V_{good}, we use the corresponding previous path covers as the path covers for the current reschedule. However, for the variables in V_{bad}, we apply *Solve-GOA* to the access subgraph for V_{bad}, and generate new access subgraphs and their path covers.

Figure 8 shows an example of our selective application of *Solve-GOA*. Figure 8(a) shows an access graph that was partitioned into three subgraphs containing variables $V_1 = \{a,b,c\}$, $V_2 = \{d,e,f,g\}$ and $V_3 = \{h,i,j,k\}$ each for the previous schedule. Then, by the changes of weights in the access graph due to a reschedule, it results in $V_{good} = V_1$ and $V_{bad} = V_2 \bigcup V_3$. Consequently, the path cover for V_1 does not changed, whereas the partitioning and path covers for V_{bad} are updated as shown in Figure 8(b). As a result, the total cost reduction of C^{tot} in *Eq.*(2) by the reschedule is (1+1+2)-(0+1+0) = 3.

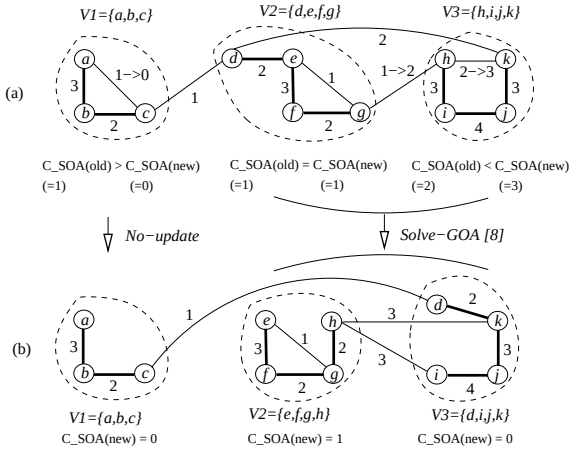


Figure 8: Determination of path covers for Sch-GOA. C_SOA represents C value for the access graph corresponding to the partition.

Exploiting operand-input commutativity: For a commutative operation such as $f = a + b$, the access order can be either abf or baf. For a commutative operation to be rescheduled, our algorithm generates the access graph for each alternative of variable access sequence, and compute the new path cover from the access graph according to the procedure described in Sch-SOA/GOA. Then, we choose the access sequence that has the smaller value of C.

4. EXPERIMENTAL RESULTS

The proposed address code generation algorithm was implemented in C++ and executed on an Intel Pentium3 computer. We tested a set of benchmark programs and randomly generated designs. We considered both SOA and GOA. We assumed that the architectures have a sufficiently large number of temporary registers to store all temporary variables in C-code to avoid the memory accesses due to memory spill values. Since our algorithm is performed at the operation code-level, and not directly at the assembly code-level of target machine, the architecture specific pointer operations with increment/decrement (*p++) was assumed to be achieved with a single variable access operation in code, like LAR p, LACL *, +.

Table 1 shows comparisons of the results in terms of address code size produced by OFU (the order of the first use) offset assignment, Solve-SOA with tie-breaking [7], and the proposed Sch-SOA for the randomly generated C programs. $|V|$ and $|S|$ represent the number of variables and the access sequence length, respectively. The last column shows the run time of Sch-SOA. Sch-SOA reduces the address code size up to 74% and 27% over those by OFU and Solve-SOA, respectively.

Table 2 shows the results produced by Solve-GOA [7], and the proposed Sch-GOA for the randomly generated C programs. $|AR|$ represents the number of address registers (ARs). The comparison of the results indicates that Sch-GOA performs well compared to Solve-GOA, reducing the address code size up to 36% more than that of Solve-GOA. Moreover, the reductions are consistent.

Table 3 shows comparisons of results for benchmark C programs when $|AR|$=1 (i.e., SOA). FIR and BIQUAD (biquad_one_section) are taken from DSPstone benchmark suite. COMP (complex multiplier) and ELLIP (elliptical wave filter) are taken from high-level

|V|/|S|	# addr_instr			gain over(%)	time
	OFU	Sol-SOA[7]	Sch-SOA	OFU/[7]	(sec.)
5/10	3.8	1.8	1.4	63.1/22.2	0
5/20	11.1	6.9	5.8	47.7/15.9	0
15/20	13	5.4	4.5	65.3/16.6	0.01
10/50	33.2	25.1	18.9	43.0/24.7	0.02
40/50	35.8	13.3	10.2	71.5/23.3	1.4
10/100	68.2	54.2	42.5	37.6/21.5	0.1
50/100	80.7	48.2	36.9	54.2/23.4	17
80/100	71.3	25.3	18.4	74.1/27.2	45
100/200	167.3	101.4	80.29	52/20.8	1125

Table 1: SOA results for randomly generated codes.

|V|/|S|	|AR|	# addr_instr		gain over	time
		Solve-GOA[7]	Sch-GOA	[7] (%)	(sec.)
10/50	2	15.6	11.2	28.2	0.08
10/50	4	6.8	5.4	20.5	0.03
40/50	4	7.4	4.7	36.4	3.50
40/50	8	8.8	8.1	7.9	1.08
10/100	4	11.3	8.1	28.3	0.04
10/100	8	5	5	0.0	0.02
50/100	8	20.3	14.1	30.5	1.80
80/100	8	10.9	8.6	21.1	6.86
100/200	8	54.9	46	16.2	144

Table 2: GOA results for randomly generated codes.

synthesis benchmark designs. GAULEG (Gauss-Legendre weights and abscissas), GAUHER (Gauss-Laguerre weights and abscissas) and GAUJAC (Gauss-Jacobi weights and abscissas) are taken from [13]. Overall, the improvements by Sch-SOA are 53% and 34% on the average compared to OFU and Solve-SOA [7], respectively.

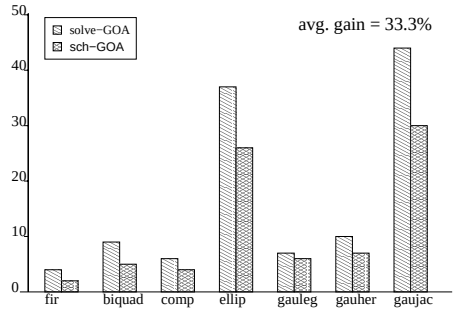
design ($\|V\|/\|S\|$)	# addr_instr			gain over
	OFU	Sol-SOA[7]	Sch-SOA	OFU/[7] (%)
FIR (5/20)	7	4	2	71.4/50
BIQUAD (10/38)	20	16	11	45/31.2
COMP (10/18)	13	8	6	53.8/25
ELLIP (45/100)	84	49	36	57.1/26.5
GAULEG (23/73)	25	20	12	52/40
GAUHER (11/59)	23	21	13	43.4/38.0
GAUJAC (23/148)	111	76	50	54.9/34.2
avg.				53.9/34.9

Table 3: SOA results for benchmark designs.

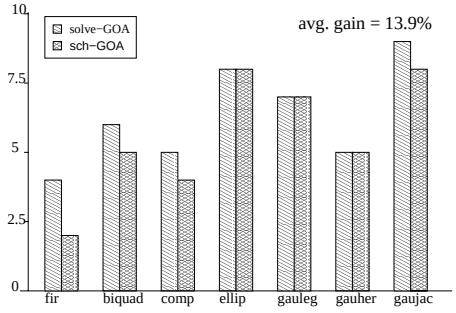
Figure 9 graphically summarizes the comparisons of the address code sizes produced by Solve-GOA [7] and Sch-GOA with the changes of the number of ARs. The numbers in X-axis indicates the number of address instructions. As the number of ARs increases the improvement by Sch-GOA is less. This is mainly due to the fact that the number of ARs is relatively large enough to fully utilize the auto-increment/decrement addressing modes without the exploitation of scheduling. However, overall our improvements are 13%-33% of address code reduction over Solve-GOA and it took within 16 seconds when $|AR| = 2$ and 4 seconds when $|AR|=8$.

5. CONCLUSIONS

In this paper, we proposed a new technique for optimizing address instructions for DSP code generation by utilizing code scheduling. Specifically, we solve the SOA/GOA (offset assignment) problem coupled with scheduling in two steps: (1) generating an initial schedule and address assignment and (2) iteratively improving the SOA/GOA solution. To support a large number of iterations, we



(a) Results when |AR|=2



(b) Results when |AR| = 8

Figure 9: GOA results for benchmark designs.

devised *a set of techniques for fast evaluation of the SOA/GOA solution for a reschedule based on the solution for the previous schedule.* Based on iterative improvement technique, the proposed algorithm may be slow for a large code. However, by limiting the number of iterations or reschedulable operations, it can have moderate speed. Since our approach is oriented towards reducing addressing code size, it may not be suitable for superscalar or VLIW DSPs and DSP codes with ralatively few addressing codes.

Acknowledgment : This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).

6. REFERENCES

[1] S. Liao, *et. al*, "Storage Assignment to Decrease Code Size," *Proc. SIGPLAN on PLDI*, 1995.

[2] P. Marwedel, G. Goossens (Editors), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[3] S. Udayanarayanan and C. Chakrabarti, "Address Code Generation for Digital Signal Processors," *Proc. DAC*, 2001.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. Int. Sym. on Microarchitecture*, 1997.

[5] D. H. Bartley, "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," *Software - Practice and Experience*, Vol. 22, No. 2, 1992.

[6] S. Liao, *et. al*, "Storage Assignment to Decrease Code Size," *ACM TOPLAS*, Vol. 18, No. 3, 1996.

[7] R. Leupers and P. Marwedel, "Algorithm for Address Assignment in DSP Code Generation," *Proc. ICCAD*, 1996.

[8] R. Leupers and F. David, "A Uniform Optimization Technique for Offset Assignment Problem," *Proc. ISSS*, 1998.

[9] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," *Proc. DAC*, 1997.

[10] C. Gebotys, "DSP Address Optimization using a Minimum Cost Circulation Technique," *Proc. ICCAD*, 1997.

[11] A. Rao and S. Pande, "Storage Assignment using Expression Tree Transformation to Generate Compact and Efficient DSP Code," *Proc. ACM SIGPLAN on PLDI*, 1999.

[12] S. Lim, J. Kim and K. Choi, "Scheduling-based Code Size Reduction in Processors with Indirect Addressing Mode," *Proc. CODES*, 2001.

[13] W. H. Press, *et. al* (Editors), *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1993.