# Storage Assignment to Decrease Code Size

STAN LIAO
Synopsys, Inc.
and
SRINIVAS DEVADAS
Massachusetts Institute of Technology
and
KURT KEUTZER, STEVEN TJIANG, and ALBERT WANG
Synopsys, Inc.

DSP architectures typically provide indirect addressing modes with autoincrement and decrement. In addition, indexing mode is generally not available, and there are usually few, if any, general-purpose registers. Hence, it is necessary to use address registers and perform address arithmetic to access automatic variables. Subsuming the address arithmetic into autoincrement and decrement modes improves the size of the generated code. In this article we present a formulation of the problem of optimal storage assignment such that explicit instructions for address arithmetic are minimized. We prove that for the case of a single address register the decision problem is NP-complete, even for a single basic block. We then generalize the problem to multiple address registers. For both cases heuristic algorithms are given, and experimental results are presented.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Code size, compilation, storage assignment

## 1. INTRODUCTION

Microprocessors such as microcontrollers and fixed-point digital signal processors (DSPs) are increasingly being embedded into many electronic products. In fact, the use of microprocessors in embedded systems outnumbers the use of processors in both the PC and the workstation market combined. Two trends are becoming clear in the design of embedded systems. First, considerations for cost, power, and reliability are forcing

This research was supported in part by the Advanced Research Projects Agency under contract DABT63-94-C-0053 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation. A version of this article was presented at the 1995 ACM/SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, California, June 19–22, 1995.
Authors' addresses: S. Liao, K. Keutzer, S. Tjiang, and A. Wang, Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043-4033; email: {stanliao; keutzer; tjiang; awang}@synopsys.com; S. Devadas, Research Laboratory of Electronics, Massachusetts Institute of Technology, MA 02139-4307; email: devadas @mit.edu.

designers into taking the next step: incorporating all the electronics—microprocessor, program ROM and RAM, and application-specific circuit components—into a single integrated circuit. Second, the amount of software incorporated into embedded systems is growing larger and more complex.

The first trend elevates code density to a new level of importance because program code resides in on-chip ROM, the size of which translates directly into silicon area and cost. Moreover, designers often devote a significant amount of time to reduce code size so that the code will fit into available ROM, as exceeding on-chip ROM size could require expensive redesign of the entire IC [Ganssle 1992, p. 18] and even of the whole system. The second trend—increasing software and system complexity—mandates the use of high-level languages (HLLs) in order to decrease development costs and time-to-market. However, current compilers for microcontrollers and fixed-point DSPs generate code that leaves much room for improvement [Živojnović et al. 1994]—thus programming in an HLL can incur penalties on code size and performance.

While optimizing compilers have proved effective for general-purpose processors, the irregular data-paths and small number of registers found in embedded processors, especially fixed-point DSPs, remain a challenge to compilers. The direct application of conventional code optimization methods (e.g., Aho et al. [1986]) has thus far been unable to generate code that efficiently uses the features of fixed-point DSP architectures.

We believe that generating the best code for embedded processors will require not only traditional optimization techniques, but also new techniques that take advantage of special architectural features and that decrease code size. This article presents one of our efforts at developing such techniques: a data lay-out algorithm that decreases code size.

Many architectures (e.g., the VAX, TI TMS320C25, most embedded controllers) provide indirect addressing modes with autoincrement/autodecrement arithmetic. These features allow for efficient sequential access of memory and increase code density, because they subsume address arithmetic instructions and result in shorter instructions in variable-length instruction architectures. In particular, DSPs and embedded controllers are designed assuming software that runs on them would make heavy use of autoincrement/autodecrement addressing. In many cases, the set of available addressing modes in DSPs and controllers does not include a mode for indexing with a constant offset. Therefore, it is necessary to allocate a register and perform address arithmetic to access variables. Subsuming the address arithmetic into autoincrement/autodecrement modes improves both the size and performance of the generated code.

The placement of variables in storage has a significant impact on the effectiveness of subsumption. Our compiler delays storage allocation of variables, moving it from the front-end to the code generation step that selects addressing modes, thereby increasing opportunities to use efficient autoincrement/autodecrement modes. We formulate this delayed storage allocation as the *offset assignment problem*. Although some modern DSP architectures permit increments and decrements of values other than one (e.g., Motorola DSP56000), it is costly to use this feature if the modifier value varies frequently. This is because extra instructions are required to set the modifier value, which is typically stored in a register rather than encoded in the instruction word. (This feature is usually for traversing arrays with strides greater than one.) Therefore, we will focus on unit increments and decrements.

We first consider a simpler problem that we call *simple offset assignment* (*SOA*). A solution to the SOA problem assigns optimal frame offsets to variables of a procedure, assuming that the target machine has *a single indexing register with only the indirect, autoincrement, and autodecrement addressing modes.* For the SOA problem, we represent a procedure by its sequence of variable accesses, and we summarize the access patterns by a weighted undirected graph called an *access graph*.

Bartley [1992] was the first to address the SOA problem and presented an approach based on finding a maximum-weight Hamiltonian path on the access graph. However, several aspects of his formulation and implementation can be improved. He considered *complete graphs* which usually contain much information unnecessary for the construction of an optimal solution to the original assignment problem. Also, his procedures for selecting an edge of the Hamiltonian path and detecting whether a cycle is created by a selection are inefficient, of $O(|V|)$. As a result, his algorithm runs in $O(|V|^3 + |L|)$ time, where $|V|$ is the number of variables, and $|L|$ is the number of variable accesses.

In this article we provide a more formal treatment of the offset assignment problem. We show that the SOA problem is equivalent to a path-covering problem of the access graph and that the decision problem for SOA is NP-complete. We then present an $O(|E| \log |E| + |L|)$ algorithm that produces empirically near-optimal solutions, where $|E|$ is the number of edges in the access graph. Our extensive experimental results on larger examples (Section 6) indicate that access graphs are generally quite sparse and, therefore, that our algorithm has a great advantage over Bartley's. There are several similarities between our approach and Bartley's—both are based on access graphs, and both use a greedy strategy in selecting edges in the graph. However, instead of considering complete graphs, we retain only those edges with nonzero weights. This allows for an $O(1)$ procedure for testing whether selecting an edge causes a cycle, which is essential to reducing the overall complexity of the heuristic.

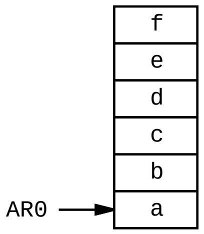
We also extend SOA to the *general offset assignment problem* (*GOA*) that handles multiple index registers. We show how the heuristic used for SOA is used to efficiently solve GOA. Since the SOA heuristic is used as a core procedure for GOA, the reduction in complexity from $O(|V|^3)$ to $O(|E| \log |E|)$ is significant. Although we emphasize code size, our formulation of the offset assignment problem also lends itself naturally to application-specific performance optimization in the presence of trace information from actual applications.

## 2. PROCESSOR MODEL AND NOTATIONS

For the purpose of exposition, we use a simple processor model that reflects the addressing capabilities of most DSPs. The model is an accumulator-based machine. Each operation involves the accumulator and, if any, another operand from the memory. Memory access can occur only indirectly via a set of address registers, `AR0` through `AR`$(k-1)$. Furthermore, if an instruction uses `AR`$i$ for indirect addressing, then in the same instruction `AR`$i$ can be optionally postincremented or postdecremented by 1 at no 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` and `SBAR`. Also, to initialize an address register, the `LDAR` instruction is used. Since `LDAR` involves the address of a variable, its cost is typically higher than either `ADAR` or `SBAR`. Thus if the contents of an address register are known, `ADAR` and `SBAR` are preferred. We use `*(AR`$i$`)`, `*(AR`$i$`)+`, and `*(AR`$i$`)-` to denote indirect addressing through `AR`$i$, indirect addressing with postincrement, and indirect addressing with postdecrement, respectively.

```
c = a + b;
f = d + e;
a = a + d;
c = d + a;
d = d + f + a;
```

(a)



(b)

```
LDAR  AR0,&a    ; a
LOAD  *(AR0)+   ; b
ADD   *(AR0)+   ; c
STOR  *(AR0)+   ; d
LOAD  *(AR0)+   ; e
ADD   *(AR0)+   ; f
STOR  *(AR0)
SBAR  AR0,5     ; a
LOAD  *(AR0)
ADAR  AR0,3     ; d
ADD   *(AR0)
SBAR  AR0,3     ; a
STOR  *(AR0)
ADAR  AR0,3     ; d
LOAD  *(AR0)
SBAR  AR0,3     ; a
ADD   *(AR0)
ADAR  AR0,2     ; c
STOR  *(AR0)+   ; d
LOAD  *(AR0)
ADAR  AR0,2     ; f
ADD   *(AR0)
SBAR  AR0,5     ; a
ADD   *(AR0)
ADAR  AR0,3     ; d
STOR  *(AR0)
```

(c)

Fig. 1. (a) Code sequence; (b) offset assignment; (c) assembly code.

## 3. SIMPLE OFFSET ASSIGNMENT

In this section we assume that only one address register is used to address all variables. We describe the optimization problem of assigning offsets to variables in a frame so as to obtain the most compact code. This implies that we have to minimize the number of instructions whose sole function is setting `AR0` to point to appropriate locations in the frame, i.e., `ADAR` and `SBAR`.

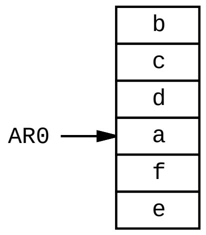
### 3.1 Example

As an example illustrating the offset assignment problem, consider the C program in Figure 1(a). Assume that the offset assignment to the various variables is as shown in Figure 1(b). The assembly code for the C program is shown in Figure 1(c). In the assembly code, the comment following each instruction indicates which variable `AR0` points to, after the instruction is executed. The instructions `ADAR` and `SBAR` are used to change `AR0` to point to the frame location accessed in the next instruction.

Assume that `AR0` initially points to variable `a`. The value of `a` is loaded in the accumulator, and `AR0` is postincremented in the first `LOAD` instruction so that it now points to `b`. In the second `ADD` instruction, the values in `a` and `b` are summed and stored in the accumulator; `AR0` is again postincremented. Next, using the instruction `STOR` the contents of the accumulator are stored in the location corresponding to variable `c`. When the assembly instructions corresponding to `a = a + d` are to be executed, we have to load `a` into the accumulator, but `AR0` now points to `f`. Therefore, we have to subtract 5 from the contents of `AR0` using an explicit instruction `SBAR AR0,5`. In total, nine `SBAR` and `ADAR` instructions are required to execute the code of Figure 1(a), given the offset

```
c = a + b;
f = d + e;
a = a + d;
c = d + a;
d = d + f + a;
```

(a)



(b)

```
LDAR  AR0,&a    ; a
LOAD  *(AR0)
ADAR  AR0,3     ; b
ADD   *(AR0)-   ; c
STOR  *(AR0)-   ; d
LOAD  *(AR0)
SBAR  AR0,3     ; e
ADD   *(AR0)+   ; f
STOR  *(AR0)+   ; a
LOAD  *(AR0)+   ; d
ADD   *(AR0)-   ; a
STOR  *(AR0)+   ; d
LOAD  *(AR0)-   ; a
ADD   *(AR0)
ADAR  AR0,2     ; c
STOR  *(AR0)-   ; d
LOAD  *(AR0)
SBAR  AR0,2     ; f
ADD   *(AR0)+   ; a
ADD   *(AR0)+   ; d
STOR  *(AR0)
```

(c)

Fig. 2. (a) Code sequence; (b) better offset assignment; (c) assembly code.

assignment of Figure 1(b).

Now consider the offset assignment of Figure 2(b) for the same C code. Assume as before that the AR0 register points to variable a initially. This assignment leads to a short assembly code sequence of Figure 2(c). Only four SBAR and ADAR instructions are required to execute the code of Figure 2(a).

We define the cost of an assignment to be the number of SBAR and ADAR instructions required. (When using a single address register, the number of LDAR instructions is a constant and, therefore, may be ignored. For multiple address registers, some LDAR instructions will be needed for every additional address register introduced; this cost is included in the setup cost described in Section 4.)

## 3.2 Assumptions in SOA

The simple offset assignment (SOA) problem involves assigning an offset to each of the local variables to minimize the number of instructions required to perform address arithmetic in a basic block under the following assumptions:

—Every data object has a size of one word.

—A single address register AR0 is used to address all variables.

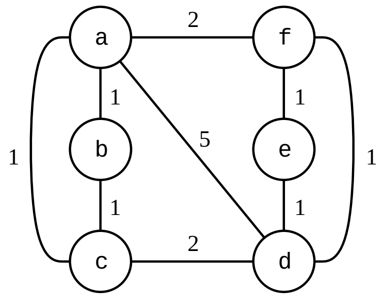
—One-to-one mapping of variables to locations.

—The basic block has a fixed evaluation order (schedule).

## 3.3 Approach to the Problem

Our approach to solving the SOA problem is to formulate it as a well-defined combinatorial problem of graph covering, called *maximum-weight path covering* (MWPC). From a basic block we derive a graph, called an access graph, that summarizes the relative benefits of assigning each pair of variables to adjacent locations. By solving the MWPC problem, we can construct an assignment with minimum cost. We then show `a b c d e f a d a d a c d f a d`

(a)



(b)

Fig. 3. (a) Access sequence; (b) access graph.

how to reduce an instance of the Hamiltonian path problem into an instance of SOA, demonstrating that a fast exact algorithm for SOA is likely to elude us (unless, of course, $P = NP$). At the end of this section, we present a heuristic algorithm for SOA/MWPC.

## 3.4 Access Sequence and Access Graph

Given a C code sequence that represents a basic block, we can uniquely define an *access sequence* for the block. For an operation $z = x$ **op** $y$, the access sequence is $x y z$. The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation in the appropriate order. The access sequence for the basic block of Figure 2(a) is shown in Figure 3(a).

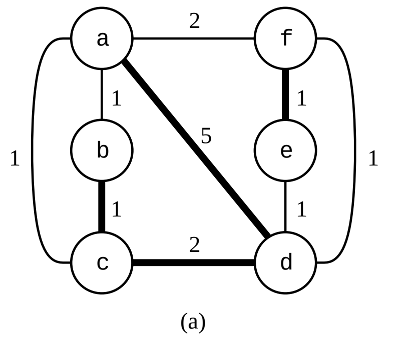
With the notion of the access sequence, it is easily seen that the cost of an assignment is equal to the number of adjacent accesses of variables that are not assigned to adjacent locations. For instance, four address arithmetic instructions are required for the offset assignment in Figure 2, since the following two-symbol substrings of the access sequence refer to variables assigned to nonadjacent locations: `a b`, `d e`, `a c`, and `d f`.

The *access graph* $G\langle V, E\rangle$ is derived from an access sequence, as follows. Each node $v \in V$ in the graph corresponds to a unique variable. An edge $e\langle u, v\rangle \in E$ exists with weight $w(e)$ if variables $u$ and $v$ are adjacent to each other $w(e)$ times in the access sequence. Note that it makes no difference whether $u$ is before or after $v$, since we can autoincrement or autodecrement `AR0` during any load, store, or arithmetic instruction. The access graph for the basic block of Figure 2(a) is shown in Figure 3(b).

Thus, in term of the access graph, the cost of an assignment is equal to the sum of the weights of all edges that do not connect variables assigned to adjacent locations. For the example in Figure 2, the edges $\langle \mathtt{a}, \mathtt{b}\rangle$, $\langle \mathtt{a}, \mathtt{c}\rangle$, $\langle \mathtt{d}, \mathtt{e}\rangle$, and $\langle \mathtt{d}, \mathtt{f}\rangle$ are such edges, and these edges have a total weight of 4.

## 3.5 SOA and Maximum Weight Path Covering

*Definition* 3.5.1. A path $P$ in $G$ is an alternating sequence of nodes and edges $[v_1, e_1, v_2, e_2, \ldots, e_{m-1}, v_m]$, where $e_i = \langle v_i, v_{i+1}\rangle \in E$, and no $v_i$ appears more than once on the path.



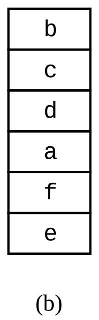


Fig. 4. (a) A disjoint path cover with a cost of 6; (b) an implied assignment with a cost of 4.

*Definition* 3.5.2. Two paths are said to be *disjoint* if they do not share any nodes.

*Definition* 3.5.3. A *disjoint path cover* (henceforth *cover*) of a weighted graph $G$ is a subgraph $C\langle V, E'\rangle$ of $G$ such that:

—For every node $v$ in $C$, $\deg(v) \leq 2$;
—There are no cycles in $C$.

Note that the edges in $C$ form a set of disjoint paths (some of which may contain no edges), hence the name.

*Definition* 3.5.4. The *weight* of a graph $G$ is the sum of the weights of the edges in $G$. The *cost* of a cover $C$ of $G$ is the sum of the weights of the edges in $G$ but *not* in $C$:

$$\text{cost}(C) = \text{weight}(G) - \text{weight}(C).$$

*Definition* 3.5.5. An offset assignment $A$ is said to be *implied* by a cover $C$ if edge $e\langle u, v\rangle \in C$ implies variables $u$ and $v$ are adjacent in $A$.

*Definition* 3.5.6 (MWPC). Given an access graph $G$, find a cover $C$ with maximum weight.

Note that a cover with maximum weight is also one with minimum cost. We now show that solving the MWPC problem is equivalent to solving the simple offset assignment problem.

LEMMA 3.5.7. *Given a cover $C$ of $G$, the cost of every offset assignment implied by $C$ is less than or equal to the cost of the cover.*

PROOF. Let $A$ be any assignment implied by $C$. As seen in Section 3.4, the cost of the assignment is equal to the sum of the weights of all edges $\langle u, v\rangle$ such that $|A(u) - A(v)| > 1$, where $A(u)$ denotes the offset of variable $u$ under assignment $A$. By Definition 3.5.5, these edges are a subset of edges in $G$ but not in $C$. (There may well exist nodes $u$ and $v$ such that $|A(u) - A(v)| = 1$, but $\langle u, v\rangle$ is not in $C$.) Thus the cost of this assignment is at most equal to that of $C$. □

Figure 4 gives an example of a cover and an implied assignment with cost less than that of the cover. The edge $\langle \texttt{a}, \texttt{f}\rangle$ is not in the cover; but it does connect two variables

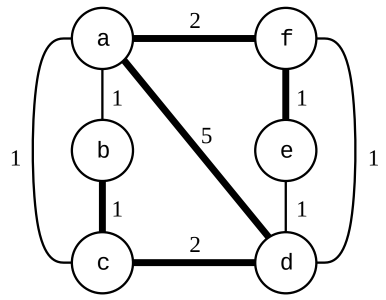


Fig. 5. Covering an access graph.

assigned to adjacent locations. Thus, the cost of the cover is 6, whereas the cost of this particular implied assignment is 4. Comparing with the cover in Figure 5, it is evident that this cover is not optimal.

LEMMA 3.5.8. *Given any offset assignment A and an access graph G, there exists a disjoint path cover C which implies A and which has the same cost as A.*

PROOF. Given an assignment $A$, we construct a cover $C$ as follows: for each pair of nodes $(u, v)$ such that $A(u) = A(v) + 1$, we pick the edge $\langle u, v\rangle$, if it exists in $G$, to be included in $C$. $C$ is a disjoint path cover because no node in $C$ has a degree greater than 2 (a variable can have at most two neighbors), and there are no cycles (we are not considering memory wraparound). Furthermore, $C$ implies $A$ by construction. The edges in $G$ but not in $C$ are exactly those which connect two nodes with nonadjacent assignments, and thus the cost of $C$ is exactly equal to that of $A$. □

THEOREM 3.5.9. *Every offset assignment implied by an optimal cover is optimal.*

PROOF. Let $C$ be an optimal cover with cost $c$. Suppose there is an assignment (not necessarily implied by $C$) with cost $c' < c$. Since an offset assignment implies the existence of a disjoint path cover with the same cost (Lemma 3.5.8), there is a disjoint path cover with cost $c'$ which is less than $c$. This contradicts our assumption that $C$ is an optimal cover. Hence, no assignment has a cost strictly less than $c$, and all assignments implied by $C$ have cost $c$ (Lemma 3.5.7). □

Theorem 3.5.9 allows us to arrive at an optimal simple offset assignment by solving the corresponding MWPC problem. Intuitively, an edge denotes the number of times two variables are accessed immediately one after another and hence the number of address arithmetic instructions necessary if these two variables are not assigned to adjacent locations. Therefore, by selecting a cover with the maximum weight we minimize the number of address arithmetic instructions required.

Consider the access graph of Figure 5. The dark edges beginning from variable `e` and ending at variable `b` form a maximum-weight path cover (using a single path). This path corresponds to the offset assignment of Figure 2(b). The unselected edges in Figure 5 have a weight of 4. This means that the number of instructions required to explicitly manipulate `AR0` is 4. This is indeed true as seen in Figure 2(c).

Our approach to solving the SOA problem is in essence to *reduce* it to the MWPC problem, solve the latter, and then construct a solution to the former. It is trivial to prove that the MWPC problem is NP-complete. This, however, does not mean that SOA is itself an NP-complete problem, since we might have reduced a problem in complexity class P to one in NPC. Just as one had to prove that the register allocation problem is as hard as the coloring problem to which it is usually reduced [Chaitin et al. 1981], we need to show that the SOA problem is indeed an NP-hard problem. We will do so by constructing an access sequence from an instance of the (unweighted) Hamiltonian path problem, such that optimally solving the offset assignment problem on the access sequence will yield a decision to the Hamiltonian path problem.

LEMMA 3.5.10. *Given an undirected graph $G$, there exists an access sequence such that the corresponding access graph $G'$ is isomorphic to $G$, and each edge of $G'$ has a weight of 2.*

PROOF. Select any node $r$ in $G$ as the root node, and perform a depth-first search on $G$. During the depth-first search each edge $\langle u, v\rangle$ is traversed exactly twice, once forward and once backward. Consider the sequence $T$ in which the nodes are visited (including backtracks). This sequence is an access sequence that gives rise to an access graph that is isomorphic to $G$. In addition, since each edge $\langle u, v\rangle$ is traversed twice, nodes $u$ and $v$ are adjacent to each other in $T$ exactly twice as well. □

THEOREM 3.5.11. *Given an access sequence $T$ and an integer $k$, the problem of deciding whether there exists an assignment for $T$ with cost less than or equal to $k$ is NP-hard.*

PROOF. We prove this by reduction from the Hamiltonian path problem. Let $G\langle V, E\rangle$ be an undirected graph. We obtain an access sequence $T$ as in Lemma 3.5.10, with access graph $G'$ isomorphic to $G$. Each edge of $G'$ has a weight of 2. The *weight* of any cover of $G'$ is at most $2\cdot(|V|-1)$, since every edge has the same weight of 2 and since a cover can have at most $(|V|-1)$ edges. This means the *cost* of any cover is at least $2\cdot(|E|-|V|+1)$. Now let $k = 2\cdot(|E|-|V|+1)$, and suppose there is an assignment $A$ for $T$ whose cost is less than or equal to $k$. By Lemma 3.5.8 there is a cover $C$ that has the same cost as $A$. This implies that the cost of $C$ is exactly $2\cdot(|E|-|V|+1)$ and, in turn, that $C$ has $(|V|-1)$ edges. On the other hand, if $C$ has $(|V|-1)$ edges, it must be a Hamiltonian path.

Conversely, if there does not exist an assignment $A$ with cost less than or equal to $k$, then by Lemma 3.5.7 there does not exist a cover with cost equal to $k$. This means every cover has fewer than $(|V|-1)$ edges and therefore $G$ has no Hamiltonian path. □

In light of this theorem, we will need to develop efficient heuristic algorithms to solve SOA and MWPC for large problems. For small problems, a branch-and-bound procedure is feasible.

## 3.6 A Heuristic Algorithm for SOA

We describe a heuristic algorithm for MWPC that is similar to Kruskal's maximum spanning tree algorithm [Aho et al. 1974]. The algorithm is greedy in that at each step the edge with the largest weight is selected that does not yield a cycle and does not increase the degree of a node to more than two. The heuristic algorithm is shown in Figure 6.

With careful implementation, we can obtain a running time of $O(|E|\log|E|+|L|)$ for the heuristic procedure described in Section 3.6, where $|E|$ is the number of edges in

```
1   Solve-SOA($L$)
2   {
3       /* $L$ = access sequence for basic block */
4       $G\langle V, E\rangle \leftarrow$ Access-Graph($L$);
5       $F \leftarrow$ sorted list of edges in $E$ in descending order of weight;
6       $C\langle V', E'\rangle$ : $V' \leftarrow V$, $E' \leftarrow \{\ \}$;
7       while ( $|E'| < |V| - 1$ and $F$ not empty ) {
8           choose $e \leftarrow$ first edge in $F$;
9           $F \leftarrow F - \{e\}$;
10          if (($e$ does not cause any node in $V'$ to have degree $> 2$) and
11              ($e$ does not cause a cycle in $C$))
12               add $e$ to $E'$;
13          else
14               discard $e$;
15      }
16      /* Construct an assignment from $E'$ */
17      return Construct-Assignment($E'$);
18  }
```

Fig. 6. Heuristic algorithm for SOA.

the access graph, and $|L|$ is the length of the access sequence. Constructing the access sequence requires $O(|L|)$ time, and $O(|E| \log |E|)$ is due to the need to sort the edges in descending order of weight. The main loop of the algorithm (lines 7–15) runs in $O(|E|)$ time, provided that the test on lines 10–11 takes $O(1)$ time.

Testing whether an edge causes a vertex to have degree greater than 2 is trivial: we simply keep for each vertex a counter that is incremented whenever an incident edge is selected. Testing for cycles in $O(1)$ can be accomplished as follows. At any iteration of the main loop, the cover $C$ consists of a disjoint set of paths. We represent a path by a *path element* consisting of two pointers that point to the end nodes of the path. The two end nodes of a path, on the other hand, have back-pointers to the path element. Consider the selection of an edge $\langle u, v\rangle$. If $u$ and $v$ are each an end node of a path, testing whether selecting $\langle u, v\rangle$ creates a cycle amounts to testing whether they belong to the same path. If, on the other hand, $u$ (or $v$) is not an end node of a path, then the test for the degree of $u$ (or $v$) would fail in the first place. When an edge passes both tests and is selected, two paths are joined to form a new one, and a new path element is created accordingly.

As an example of applying the heuristic algorithm consider the access graph of Figure 3(c). We first pick edges $\langle \mathtt{a}, \mathtt{d}\rangle$, $\langle \mathtt{a}, \mathtt{f}\rangle$, $\langle \mathtt{c}, \mathtt{d}\rangle$. We reject $\langle \mathtt{a}, \mathtt{b}\rangle$ and $\langle \mathtt{a}, \mathtt{c}\rangle$ because each causes a cycle. Next, we pick $\langle \mathtt{b}, \mathtt{c}\rangle$. We reject $\langle \mathtt{d}, \mathtt{e}\rangle$ and $\langle \mathtt{d}, \mathtt{f}\rangle$ and finally pick $\langle \mathtt{f}, \mathtt{e}\rangle$. This results in the selection of the dark path of Figure 5 which is an optimal offset assignment.

## 4. GENERAL OFFSET ASSIGNMENT PROBLEM

We describe a generalization of the offset assignment problem to the case where there are $k$ address registers, AR0 through AR$(k-1)$.

In this generalization, we make the following additional assumptions:

(1) There is a fixed setup cost of introducing an additional address register. This setup

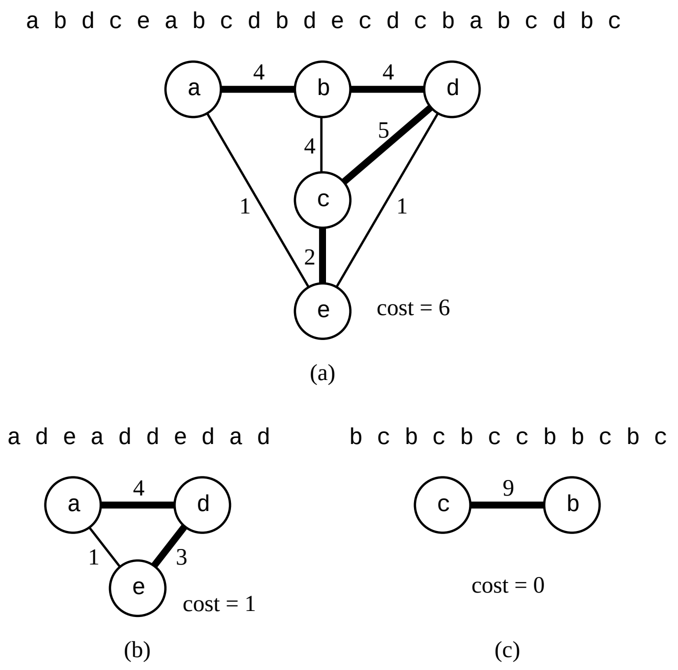


Fig. 7. (a) Access sequence and graph; (b) access subsequence and graph generated by $\{\mathsf{a}, \mathsf{d}, \mathsf{e}\}$; (c) access subsequence and graph generated by $\{\mathsf{b}, \mathsf{c}\}$.

cost reflects the cost associated with initialization upon entry to the procedure and reinitialization after return from a callee.

(2) Each address register is used to point to a disjoint subset of variables.

*Definition* 4.1. Let $L$ be the access sequence of the basic block, and $V$ be the set of variables in $L$. The *access subsequence* generated by $W \subseteq V$ is the subsequence of $L$ consisting of variables in $W$.

*Definition* 4.2 (GOA). Given an access sequence $L$, the set of variables $V$, and the number of address registers $k$, find a partition of $V$, $\Pi = \{P_1, P_2, ..., P_m\}$, where $m \leq k$, such that the total cost of the optimal SOA of the corresponding access subsequences plus the setup costs for using $m$ registers is minimum.

## 4.1 Example of GOA

Consider the access sequence and graph shown in Figure 7(a). The optimal cover is also shown, with a cost of 6. Now consider allocating a second address register for the variables b and c. The access subsequences and graphs induced by this partition are shown in Figures 7(b) and (c). Assuming a setup cost of 2, the cost of using two address registers on this partition is 3. In this case, there is an advantage in introducing a second address register.

```
1   Solve-GOA($L, k$)
2   {
3       /* $L$ = access sequence of basic block */
4       /* $k$ = number of address registers */
5       $H \leftarrow$ Solve-SOA($L$);
6       if ($k == 1$)
7           return $\{H\}$;
8       $P \leftarrow$ Select-Variables($L$);
9       $L_1 \leftarrow$ Subseq($L, P$);
10      $L_2 \leftarrow$ Subseq($L, L - P$);
11      $H_1 \leftarrow$ Solve-SOA($L_1$);
12      $H_2 \leftarrow$ Solve-SOA($L_2$);
13      if (setup-cost + cost($H_1$) + cost($H_2$) > cost($H$))
14          return $\{H\}$;
15      else
16          return $\{H_1\} \cup$ Solve-GOA($L_2, k - 1$);
17  }
```

Fig. 8. Heuristic algorithm for GOA.

## 4.2 A Heuristic Algorithm for GOA

Clearly, an exact solution to this problem is too expensive to compute. Figure 8 gives a heuristic algorithm for solving GOA. Subseq($L, P$) denotes the access subsequence of $L$ generated by $P$. Our heuristic is to build up the partition blocks incrementally by repeatedly selecting a subset of nodes as a new partition block.

The function Solve-GOA returns a collection of disjoint ordered sets of variables which forms a partition of the set of all variables. The order of each subset gives an offset assignment. Given an access sequence $L$, Solve-GOA first computes the SOA of $L$. If there is only one address register, the solution is simply the SOA. Otherwise, Solve-GOA calls Select-Variables to choose a subset of the variables in $L$ and solves SOA on the derived subsequences $L_1$ and $L_2$. If the cost of this split along with the setup cost is more expensive than that of $H$, there is no benefit in introducing the new partition block, and the current solution $H$ is returned. Otherwise, it is advantageous to introduce a new address register for this subset of variables, and Solve-GOA is recursively called for the remaining variables.

The procedure Select-Variables selects a subset of variables for which a new partition block may be created. It is important to note that on line 13 of the algorithm in Figure 8 we are making the assumption that, if allocating a new address register for the subset $L_1$ returned by Select-Variables does not reduce the cost, then further partitioning will not improve either. In other words, we assume that if there is a "good" subset of variables, Select-Variables will find it at the first opportunity.

To develop good heuristics for this procedure, we make the following observations:

(1) If an access subsequence consists of two variables, then the cost for this access subsequence is just the setup cost. No switching cost is incurred. It is also possible to select more variables (typically between two and six); provided the graph for the access subsequence is very sparse, the cost will be kept low.

(2) If a node in an access graph has more than two edges, the associated minimum

penalty for retaining the node in the graph is the sum of the weights on all edges except the two with the largest weights. Hence, if a variable has a high penalty, then it may be beneficial to move it to another partition block. This is analogous to allocating registers for variables that are relatively busy to allow for faster access.

A simple heuristic for Select-Variables is to choose a small subset of variables with the largest penalty and allocate a new address register for this subset. Our first experiments were based on selecting a *fixed* number $p$ of variables for every iteration. Although the computational requirement for this heuristic is small, it is not always clear what $p$ should be. As our experimental results in Section 6 show, the "best" $p$ varies among examples. We may also try more aggressive strategies by varying $p$ between iterations, i.e., to choose a different number of variables for each call to Select-Variables. This will require much more computation, because accurate estimation of the effect of partitioning requires several calls to Solve-SOA. We are currently implementing and evaluating these strategies. Our initial results of GOA already show encouraging improvements.

## 5. OFFSET ASSIGNMENT FOR A PROCEDURE

As the offset assignment for one basic block affects the other, we must tackle the problem for entire procedures. It is relatively straightforward to extend the formulation to take into account the presence of control-flow. Because our formulation of GOA breaks down the problem into several instances of SOA, in this section we will, for the sake of clarity, focus on using only one address register, `AR0`.

As in the basic SOA formulation, we wish to capture the patterns in which the variables are accessed throughout the procedure by counting the number of times each pair of variables is accessed consecutively. Let $V$ be the set of variables the address register may point to, and let $\textbf{first}(n)$ and $\textbf{last}(n)$ denote the first variable and last variable accessed in block $n$. In addition, let $\textbf{count}(n)$ and $\textbf{count}(f)$ denote the expected execution frequency of basic block $n$ and control-flow edge $f$. (If code size is our only objective, then we let $\textbf{count}(n) = \textbf{count}(f) = 1$ for all $n$ and $f$.) We begin by building the access graph for each basic block $n$, with the edges properly weighted by the execution count $\textbf{count}(n)$. These are merged to form the access graph $G$ for the entire procedure. Then, for each control-flow edge $f = \langle n, m \rangle$ ($m$ a successor of $n$), we increase the weight of the edge $\langle \textbf{last}(n), \textbf{first}(m) \rangle$ (in $G$) by $\textbf{count}(f)$, or create such an edge with weight $\textbf{count}(f)$ if it does not already exist.

This access graph $G$ is then covered by using either the heuristic described in Sections 3.6 or a branch-and-bound procedure. Once a solution is found, we will determine the contents of the address register (`AR`) at the exit of each basic block and will place autoincrement, autodecrement, address-arithmetic instructions (`ADAR` and `SBAR`), and address-register initialization instructions (`LDAR`) at the appropriate locations so that the number of instructions is minimized.

For example, consider the fragment of a control-flow graph shown in Figure 9. Suppose in the final assignment the variables appear in this order: `j i x`. We may, at the end of both basic blocks `L3` and `L6`, perform postincrement after accessing `j`, so that, on exit of either, `AR0` points to `i`. Thus, on entering `L1`, `AR0` points to the desired variable. In addition, due to the postincrement in `L6`, on entering `L5` `AR0` points to a *known* location, and we can use the `ADAR AR0,1` instruction at the beginning of `L5` to set it to point to `x`, instead of `LDAR`. Since `LDAR` typically has a higher cost than `ADAR` or

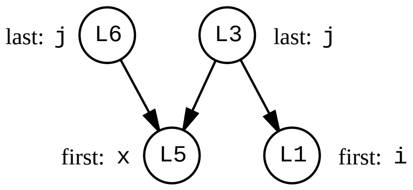


Fig. 9. Fragment of a control-flow graph.

SBAR, it is preferable that upon entering a basic block that the contents of the address register are known, as in this example. A detailed description of placing appropriate addressing modes and instructions can be found in Liao [1996].

## 6. EXPERIMENTS AND RESULTS

We have implemented the heuristic algorithms of Sections 3 and 4 and a branch-and-bound procedure in order to evaluate the heuristic SOA algorithm. All the implementations handle not only basic blocks, but entire procedures, with the formulation described in Section 5. Our initial goal is to minimize static code size; hence, we weigh each basic block equally.

Table I exhibits a summary of the examples we tested for offset assignment. The first five examples, ChenDCT through Jrev, are core routines from a JPEG–MPEG implementation. The next eight, LoadGIF through 332Dither, are graphics routines from the xv program. Following them are procedures from the GNU gzip program: GenBitlen through UnLZW. InitDES and UFCDoIt are two procedures in the GNU implementation of the DES encryption algorithm. Finally, MD5c and DecQuan were taken from an implementation of the RSA cryptosystem.

The column labeled $|V|$ shows the number of variables, including compiler-generated temporaries, in the procedure. The next column, $|E|$, gives the number of edges in the initial access graph. It is easily seen that the access graphs are very sparse. As we have indicated previously, this sparsity is favorable to both our heuristic and branch-and-bound procedures. The column labeled "LB" shows the number of instructions in the generated code excluding those that manipulate the address registers (i.e., LDAR, SBRK, and ADRK). The next columns show the number of instructions when variables are assigned to locations based on order of declaration, and the ratio of this number to that shown in "LB," which also forms the basis for the ratios in the tables to follow.

The number of instructions in "LB" serves as a lower bound against which we can evaluate our results. However, this lower bound is not very tight, since it is impossible to completely eliminate all such instructions under the assumption that address registers will be used to address variables. Therefore, the ratios shown in the tables to follow are on the conservative side—the actual lower bounds may be somewhat higher.

Table II shows the experimental results of simple offset assignment using the greedy heuristic and using a branch-and-bound procedure. CPU times are measured in seconds on a SparcStation 20. On the average, the greedy heuristic reduces the number of instructions by 5% (with respect to the lower bound), or approximately 20% of address-arithmetic instructions. Note that for all examples the difference between the results between the heuristic and the branch-and-bound procedure are very close.

Table III shows the experimental results for general offset assignment with six address

Table I. Summary of Examples

| Procedure | $\|V\|$ | $\|E\|$ | LB | Decl Ord | |
|---|---|---|---|---|---|
| | | | Inst | Inst | Ratio |
| ChenDCT | 24 | 63 | 561 | 718 | 1.280 |
| IChenDCT | 31 | 82 | 579 | 790 | 1.364 |
| LeeDCT | 26 | 82 | 616 | 836 | 1.357 |
| ILeeDCT | 48 | 131 | 686 | 974 | 1.420 |
| Jrev | 29 | 141 | 3293 | 4524 | 1.374 |
| LoadGIF | 126 | 150 | 1597 | 1797 | 1.125 |
| AutoCrop | 23 | 53 | 506 | 585 | 1.156 |
| AutoCrop24 | 128 | 290 | 1719 | 2113 | 1.229 |
| SmoothX | 27 | 120 | 621 | 795 | 1.280 |
| SmoothY | 60 | 152 | 763 | 979 | 1.283 |
| SmoothXY | 50 | 102 | 513 | 671 | 1.308 |
| Dither | 97 | 219 | 1345 | 1712 | 1.273 |
| 332Dither | 62 | 143 | 823 | 1057 | 1.284 |
| GenBitLen | 18 | 45 | 344 | 420 | 1.221 |
| HuftBuild | 39 | 92 | 702 | 896 | 1.276 |
| InflateC | 36 | 62 | 623 | 779 | 1.250 |
| InflateD | 48 | 79 | 819 | 963 | 1.176 |
| InflateS | 15 | 19 | 241 | 284 | 1.178 |
| LongMatch | 35 | 66 | 454 | 532 | 1.172 |
| ScanTree | 16 | 33 | 191 | 223 | 1.168 |
| UnLZW | 34 | 68 | 771 | 909 | 1.179 |
| InitDES | 38 | 63 | 888 | 1005 | 1.132 |
| UFCDoIt | 18 | 52 | 280 | 386 | 1.379 |
| MD5c | 10 | 19 | 2366 | 2643 | 1.117 |
| DecQuan | 73 | 129 | 790 | 961 | 1.216 |
| Cumulative | — | — | 22091 | 27552 | 1.247 |

registers ($k = 6$ in Figure 8), compared against the ratios based on the branch-and-bound SOA. We use the greedy SOA heuristic for the function Solve-SOA, because the heuristic performs very well in practice. As in previous tables, the ratios shown in the column "GOA Ratio" are based on the simple lower bound. We have also experimented with varying number of variables ($p$), between two and six, selected in the procedure Select-Variables of Figure 8; the best numbers are shown in the column "Sel." The CPU times given are the total times for trying different values of $p$. Also, although six address registers were allocated, not all of them were used, for the setup costs may outweigh the benefits when too many address registers are used. The number of address registers that are actually used is shown in the column "Reg." On the average, using multiple address registers further reduces the number of instructions by 9.1%, or another 46% of the address-arithmetic instructions that SOA could not eliminate. Since the lower bound is obviously loose, the results are in fact closer to the optimal than shown in the tables.

## 7. SUMMARY AND ONGOING WORK

The optimization techniques described in this article are incorporated into our framework for developing compilers for embedded systems [Araujo et al. 1995]. A diagram showing the stages of the compiler is shown in Figure 10.

Table II. Results of SOA—Heuristic *vs.* Branch-and-Bound

| Procedure | LB | Decl Ord | Greedy SOA | | | B&B SOA | | |
|---|---|---|---|---|---|---|---|---|
| | Inst | Ratio | Inst | Ratio | CPU | Inst | Ratio | CPU |
| ChenDCT | 561 | 1.280 | 689 | 1.228 | 0.5s | 688 | 1.226 | 0.7s |
| IChenDCT | 579 | 1.364 | 753 | 1.300 | 0.5s | 750 | 1.295 | 0.7s |
| LeeDCT | 616 | 1.357 | 787 | 1.278 | 0.6s | 784 | 1.273 | 0.9s |
| ILeeDCT | 686 | 1.420 | 907 | 1.322 | 0.7s | 905 | 1.319 | 1.8s |
| Jrev | 3293 | 1.374 | 4302 | 1.306 | 2.8s | 4285 | 1.301 | 3.9s |
| LoadGIF | 1597 | 1.125 | 1727 | 1.081 | 2.5s | 1727 | 1.081 | 5.4s |
| AutoCrop | 506 | 1.156 | 571 | 1.128 | 0.8s | 571 | 1.128 | 0.9s |
| AutoCrop24 | 1719 | 1.229 | 2050 | 1.192 | 3.0s | 2049 | 1.192 | 23.9s |
| SmoothX | 621 | 1.280 | 753 | 1.213 | 0.7s | 752 | 1.211 | 1.8s |
| SmoothY | 763 | 1.283 | 929 | 1.218 | 0.8s | 929 | 1.218 | 3.0s |
| SmoothXY | 513 | 1.308 | 626 | 1.220 | 0.7s | 626 | 1.220 | 1.5s |
| Dither | 1345 | 1.273 | 1658 | 1.233 | 2.0s | 1650 | 1.227 | 6.9s |
| 332Dither | 823 | 1.284 | 1014 | 1.232 | 1.2s | 1006 | 1.222 | 2.8s |
| GenBitLen | 344 | 1.221 | 405 | 1.177 | 1.4s | 403 | 1.172 | 0.4s |
| HuftBuild | 702 | 1.276 | 848 | 1.208 | 0.9s | 844 | 1.202 | 1.3s |
| InflateC | 623 | 1.250 | 740 | 1.188 | 0.8s | 736 | 1.181 | 0.9s |
| InflateD | 819 | 1.176 | 938 | 1.145 | 1.1s | 935 | 1.142 | 1.5s |
| InflateS | 241 | 1.178 | 266 | 1.104 | 0.3s | 266 | 1.104 | 0.3s |
| LongMatch | 454 | 1.172 | 523 | 1.152 | 0.5s | 521 | 1.148 | 0.8s |
| ScanTree | 191 | 1.168 | 223 | 1.168 | 0.2s | 223 | 1.168 | 0.3s |
| UnLZW | 771 | 1.179 | 872 | 1.131 | 0.9s | 872 | 1.131 | 1.1s |
| InitDES | 888 | 1.132 | 970 | 1.092 | 0.8s | 970 | 1.092 | 1.0s |
| UFCDoIt | 280 | 1.379 | 354 | 1.264 | 0.3s | 354 | 1.264 | 0.4s |
| MD5c | 2366 | 1.117 | 2620 | 1.107 | 1.2s | 2620 | 1.107 | 1.2s |
| DecQuan | 790 | 1.216 | 944 | 1.195 | 1.3s | 944 | 1.195 | 3.4s |
| Cumulative | 22091 | 1.247 | 26469 | 1.198 | — | 26410 | 1.196 | — |

We use SUIF [Wilson et al. 1994] as our front-end. Machine-independent optimizations such as global common-subexpression elimination is carried out in SUIF. The SUIF intermediate form is then translated into another intermediate form called TWIF, which is parametrized according to the machine description. It is on this intermediate form that instruction scheduling, offset assignment, and register allocation are performed, along with machine-specific dataflow analyses and related optimizations. (At the time of this writing we have only implemented the offset assignment procedure and the final code generation pass. Scheduling and register allocation are problems we are currently investigating and will be implemented in the near future.) Object code is then finally obtained through the final phase of code generation and peephole optimization. Code compression on object code [Fraser et al. 1984; Liao 1996] proves to be effective in further increasing the code density.

Code generation for irregular data-paths and machines with severely restricted instruction sets, such as those used in DSP and embedded microprocessors, is a problem that has received relatively little attention to date. Previous work [Ellis 1985; Fisher 1981; Goossens et al. 1986; Rimey 1989] on VLIW machines, microcode generation, and application-specific instruction processors has covered the topic of irregular data paths, but restricted addressing, and code density has never been their primary concern. Liem

Table III. Results of GOA Using Six Address Registers

| Procedure | LB | B&B SOA | GOA | | | | |
|---|---|---|---|---|---|---|---|
| | Inst | Ratio | Inst | Ratio | Reg | Sel | CPU |
| ChenDCT | 561 | 1.226 | 598 | 1.066 | 6 | 2 | 4.0s |
| IChenDCT | 579 | 1.295 | 606 | 1.047 | 6 | 2 | 4.6s |
| LeeDCT | 616 | 1.273 | 699 | 1.134 | 5 | 4 | 5.0s |
| ILeeDCT | 686 | 1.319 | 756 | 1.102 | 6 | 4 | 7.1s |
| Jrev | 3293 | 1.301 | 3510 | 1.066 | 6 | 2 | 30.7s |
| LoadGIF | 1597 | 1.081 | 1716 | 1.075 | 5 | 3 | 21.6s |
| AutoCrop | 506 | 1.128 | 571 | 1.128 | 1 | — | 4.5s |
| AutoCrop24 | 1719 | 1.192 | 2016 | 1.172 | 5 | 3 | 33.1s |
| SmoothX | 621 | 1.211 | 698 | 1.124 | 5 | 3 | 7.2s |
| SmoothY | 763 | 1.218 | 866 | 1.135 | 6 | 2 | 10.0s |
| SmoothXY | 513 | 1.220 | 589 | 1.148 | 3 | 3 | 7.5s |
| Dither | 1345 | 1.227 | 1560 | 1.160 | 4 | 5 | 18.3s |
| 332Dither | 823 | 1.222 | 931 | 1.131 | 5 | 3 | 12.5s |
| GenBitLen | 344 | 1.172 | 387 | 1.125 | 4 | 4 | 3.0s |
| HuftBuild | 702 | 1.202 | 787 | 1.121 | 5 | 2 | 9.2s |
| InflateC | 623 | 1.181 | 719 | 1.154 | 2 | 4 | 5.1s |
| InflateD | 819 | 1.142 | 893 | 1.090 | 4 | 2 | 9.8s |
| InflateS | 241 | 1.104 | 266 | 1.104 | 1 | — | 4.3s |
| LongMatch | 454 | 1.148 | 498 | 1.097 | 6 | 3 | 6.0s |
| ScanTree | 191 | 1.168 | 213 | 1.115 | 5 | 2 | 3.1s |
| UnLZW | 771 | 1.131 | 857 | 1.112 | 3 | 4 | 7.3s |
| InitDES | 888 | 1.092 | 957 | 1.078 | 5 | 4 | 6.2s |
| UFCDoIt | 280 | 1.264 | 317 | 1.132 | 5 | 2 | 2.7s |
| MD5c | 2366 | 1.107 | 2468 | 1.043 | 2 | 2 | 6.1s |
| DecQuan | 790 | 1.195 | 931 | 1.178 | 2 | 5 | 8.5s |
| Cumulative | 22091 | 1.196 | 24409 | 1.105 | — | — | — |

et al. [1994] presented techniques for generating compact code; however, the benchmark programs were quite small, and it is not shown how their techniques perform on larger, more realistic programs.

With the increasing use of embedded systems, code generation for them has become very important. In this article we presented algorithms that are able to exploit the addressing mode features of most DSP architectures. Our initial results indicate that these algorithms can obtain substantial improvements in code size beyond those provided by conventional code generation techniques. We believe that this problem bears the same importance for this class of processors as register allocation for general-purpose RISC architectures.

There are several interesting problems that need further investigation. In the procedure Solve-GOA, we have focused on a particular scheme for building up the partition of variables, namely, allocating a new address register in each iteration. There may be other methods of determining the best partition given a number of available registers. In addition, the procedure Select-Variables may be refined to take more parameters into consideration. We can also extend the offset assignment problems to take into account other characteristics of variable accesses than merely those summarized by the access graph. For instance, it is not uncommon to find variables with disjoint lifetimes. We

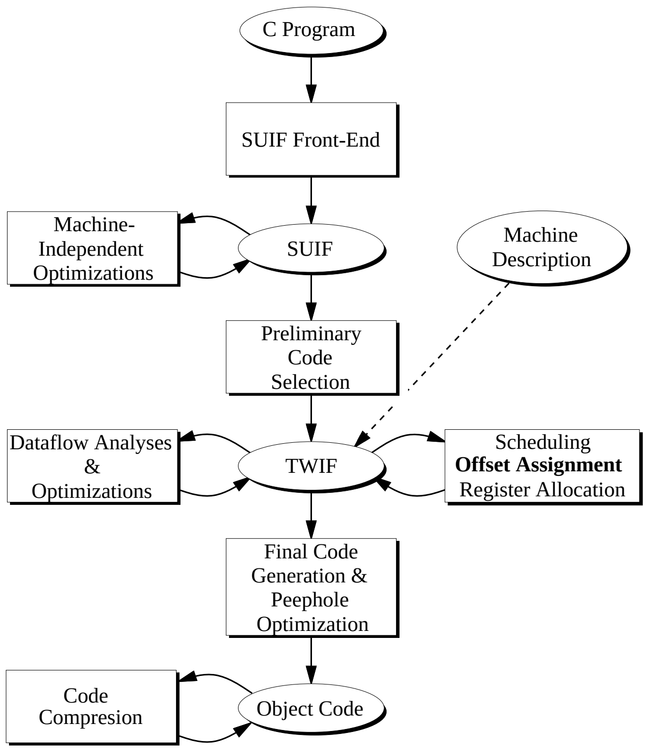

C Program
SUIF Front-End
Machine-Independent Optimizations
SUIF
Machine Description
Preliminary Code Selection
Dataflow Analyses & Optimizations
TWIF
Scheduling
Offset Assignment
Register Allocation
Final Code Generation & Peephole Optimization
Code Compresion
Object Code

Fig. 10. A framework for developing compilers for embedded systems.

can reduce data memory requirements if we assign the same location to two variables with disjoint lifetimes. In the context of simple offset assignment, however, the sharing of locations means collapsing two or more vertices into one vertex in the access graph. This may lead to vertices with too many incident edges, most of which cannot be selected; hence, merging variables with disjoint lifetimes may be detrimental to the goal of improving code size and performance we set out to achieve in this article. On the other hand, if we allocate an additional address register, as in general offset assignment, we may be able to circumvent this problem provided the access graph for the subset of variables addressed by this register is sufficiently sparse. Because the number of address registers is limited, it is not always possible to allow for merging of variables. Therefore, a more thorough analysis of variable accesses is needed. The *tile tree* [Callahan and Koblenz 1991] offers a natural and powerful way of analyzing and summarizing variable usage and has been successfully applied to the traditional register allocation problem. By effectively using the information derived from tile tree analysis, we can best utilize the data memory while keeping the program small and efficient. This is an important problem that merits further study.

## ACKNOWLEDGMENTS

The authors would like to thank Mahadevan Ganapathi for interesting discussions on the offset assignment problem, and Scott McFarling for careful reviewing of the article.

The authors would also like to thank the referees for their valuable comments and suggestions.

## REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass.

ARAUJO, G., DEVADAS, S., KEUTZER, K., LIAO, S., MALIK, S., SUDARSANAM, A., TJIANG, S., AND WANG, A. 1995. Challenges in code generation for embedded processors. In *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic, Boston, Mass., 48–64.

BARTLEY, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exper. 22*, 2 (Feb.), 101–110.

CALLAHAN, D. AND KOBLENZ, B. 1991. Register allocation via hierarchical graph coloring. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 192–203.

CHAITIN, G., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. 1981. Register allocation via coloring. *Comput. Lang.* 6, 47–57.

ELLIS, J. R. 1985. *A Compiler for VLIW Architectures*. MIT Press, Cambridge, Mass.

FISHER, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30,* 7, 478–490.

FRASER, C. W., MYERS, E. W., AND WENDT, A. L. 1984. Analyzing and compressing assembly code. In *Proceedings of 1994 ACM SIGPLAN Symposium on Compiler Construction.* ACM, New York, 117–121.

GANSSLE, J. G. 1992. *The Art of Programming Embedded Systems*. Academic Press, San Diego, Calif.

GOOSSENS, G., RABAEY, J., CATTHOOR, F., VANHOOF, J., JAIN, R., MAN, H. D., AND VANDEWALLE, J. 1986. A computer-aided design methodology for mapping DSP algorithms onto custom multiprocessor architectures. In *Proceedings of the 1986 IEEE International Symposium on Circuits and Systems.* IEEE, New York, 924–925.

LIAO, S. 1996. Code generation and optimization for embedded digital signal processors. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Mass.

LIEM, C., MAY, T., AND PAULIN, P. 1994. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the 1994 European Design and Test Conference.* IEEE, New York.

RIMEY, K. 1989. A compiler for application-specific signal processors. Ph.D. thesis, Univ. of California, Berkeley, Calif.

ŽIVOJNOVIĆ, V., VELARDE, J. M., AND SCHLÄGER, C. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*. Miller Freeman, San Francisco, Calif.

WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S.-W., TSENG, C.-W., HALL, M., LAM, M., AND HENNESSY, J. 1994. SUIF: A parallelizing and optimizing research compiler. Tech. Rep. CSL-TR-94-620, Stanford Univ., Stanford, Calif. May.

Received August 1995; revised January 1996; accepted March 1996