Storage Assignment to Decrease Code Size*

Stan Liao Srinivas Devadas
MIT Department of EECS
Cambridge, MA 02139-4307

Kurt Keutzer Steve Tjiang Albert Wang
Synopsys, Inc.
Mountain View, CA 94043-4033

Abstract

DSP architectures typically provide indirect addressing modes with auto-increment and decrement. In addition, indexing mode is 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 auto-increment and auto-decrement modes improves the size of the generated code.

In this paper 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. We then generalize the problem to multiple address registers. For both cases heuristic algorithms are given. Our experimental results indicate an improvement of 3% to 20% in code size.

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, cost, power and reliability considerations are forcing 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

*A version of this paper was presented at the 1995 ACM/SIGPLAN Conference on Programming Languague Design and Implementation, La Jolla, California, June 19–22, 1995.

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 [7, 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 poor code—thus programming in a HLL can incur significant code performance and code size penalties.

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., [2]) has thus far been unable to generate code that efficiently uses the features of fixed-point DSP microprocessors.

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 paper 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 auto-increment/decrement arithmetic. These features allows 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 auto-increment/decrement addressing. Sometimes, DSPs and controllers have such a restricted set of addressing modes that the set does not include a mode for indexing with an offset. Therefore, it is necessary to allocate a register and perform address arithmetic to access variables. Subsuming the address arithmetic into auto-increment and auto-decrement modes improves both performance and size 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, thus increasing opportunities to use efficient auto-increment/auto-decrement modes. We formulate this delayed storage allocation as the *offset assignment problem.*

First, we consider a simpler problem that we call simple offset assignment (SOA). A solution to the SOA problem assigns optimal frame offset to variables of a procedure assuming that the target machine has *a single indexing register with only the indirect, auto-increment, and auto-decrement addressing modes*. For the SOA problem, we represent a procedure by its sequence of variable accesses. We convert the access sequence into an access graph with weighted edges. We show that the SOA problem is equivalent to a linear graph covering problem of the access graph and that the decision problem for SOA is NP-complete.

Bartley was the first to address the SOA problem and presented an approach based on finding a Hamiltonian path on the graph [4]. However, his algorithm runs in $O(N^3 + L)$ time, where N is the number of variables and L is the length of the sequence of variable accesses.

In this paper we provide a more formal treatment of the problem and present an $O(E \log E + L)$ algorithm that produces near-optimal solutions, where E is the number of edges in the graph. We also extend SOA to the general offset assignment problem (GOA) that handles multiple index registers. We show how the heuristics used for SOA are used to efficiently solve GOA. Our formulation of the offset-assignment problem also lends itself naturally to application-specific code optimization in the presence of trace information from actual applications. Experimental results are presented.

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 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 ARi for indirect addressing, then in the same instruction ARi can be optionally post-incremented or post-decremented by one 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.

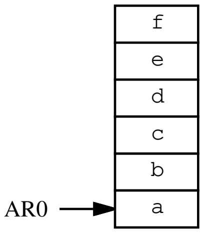
We use *(ARi), *(ARi)+, *(ARi)− to denote indirect addressing through ARi, indirect addressing with post-increment, and indirect addressing with post-decrement, respectively.

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 corresponding to 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.

```
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)

Figure 1: (a) Code sequence (b) Offset assignment (c) Assembly code

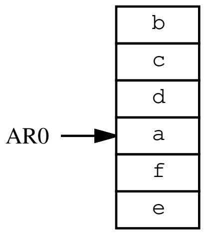
3.1 Example

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 after an instruction indicates which variable `AR0` points to after the instruction is executed. The instructions `SBAR` and `ADAR` are used to change `AR0` to point to the frame location accessed in the next instruction.

Assume that `AR0` initially points to the bottom of the frame, i.e., variable `a`. The value of the variable `a` is loaded in the accumulator, and `AR0` is incremented in the first `LOAD` instruction. In the second `ADD` instruction, the values in `a` and `b` are summed and stored in the accumulator; further, `AR0` is incremented. Next, using the instruction `STOR` the contents of the accumulator is 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` points to `f`. Therefore, we

```
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)

Figure 2: (a) Code sequence (b) Different Offset assignment (c) Assembly code

have to subtract 5 from the contents of `AR0` using an explicit instruction `SBAR AR0,5`. All in all, nine `SBAR` and `ADAR` instructions are required to execute the code of Figure 1(a), given the offset 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. We require the shorter assembly code sequence of Figure 2(c) to execute the C code of Figure 2(a). 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.

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:

- A single address register.

- 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 gives 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 how to reduce an instance of the Hamiltonian path problem into an instance of MWPC, demonstrating that a fast exact algorithm for SOA will elude us. At the end of this section, we present a heuristic algorithm to solve for SOA.

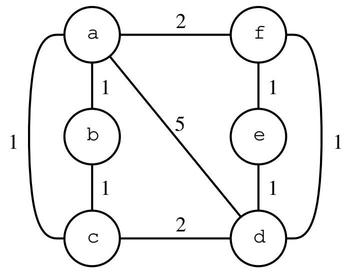
3.4 Access Sequence and Access Graph

Given a code sequence C that represents a basic block, we can uniquely define an *access sequence* for the block. Given an operation `c = a` *op* `b`, the access sequence is `a b c`. 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 non-adjacent 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 $e = \langle v_i, v_j\rangle \in E$ between nodes v_i and v_j exists `a b c d e f a d a d a c d f a d`

(a)



(b)

Figure 3: (a) Access sequence (b) Access graph

with weight $w(e)$ if variables `i` and `j` are adjacent to each other $w(e)$ times in the access sequence. Note that it does not matter if `i` is before `j` or if `i` is after `j` since we can both auto-increment and auto-decrement `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 ⟨a,b⟩, ⟨a,c⟩, ⟨d,e⟩, and ⟨d,f⟩ are such edges, and these edges have a total weight of four.

3.5 SOA and Maximum Weighted Path Covering

Definition 3.1 *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.*

Definition 3.2 *Two paths are said to be* disjoint *if they do not share any nodes.*

Definition 3.3 *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 3.4 *The* weight *of a cover C is the sum of the weights of all edges of C. The* cost *of a cover C is the sum of the weights of all edges in G but* not *in C:*

$$\text{cost}(C) = \sum_{e \in G, e \notin C} w(e).$$

Definition 3.5 *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 3.6 (MWPC) *Given an access graph G, find a cover C with maximum weight. This is equivalent to finding a cover with minimum cost.*

We now show that solving the MWPC problem is equivalent to solving the simple offset assignment problem.

Lemma 3.1 *Given a cover C of G, all offset assignments implied by C have cost 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, 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. ■

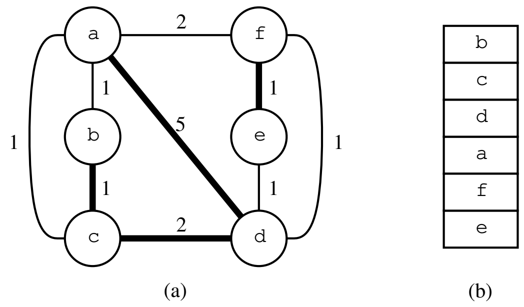


Figure 4: (a) A disjoint path cover (b) an implied assignment with a cost of four

Figure 4 gives an example of a cover and an implied assignment with cost less than that of the cover. The edge $\langle \texttt{a,f} \rangle$ is not in the cover; but it does connect two variables assigned to adjacent locations. Thus, the cost of the cover is six, whereas the cost of this particular implied assignment is four. Comparing with the cover in Figure 5, it is evident that this cover is not optimal.

Lemma 3.2 *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 disjoint path 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 two (a variable can have at most two neighbors) and there are no cycles (we are not considering memory wrap-around). Furthermore, C implies A by construction. The edges in G but not in C are exactly those which connect two nodes with non-adjacent assignments, and thus the cost of C is exactly equal to that of A. ∎

Theorem 3.1 *Every offset assignment implied by an optimal disjoint path cover is optimal.*

Proof Let C be an optimal disjoint path cover C with cost c. Suppose there is an assignment (not

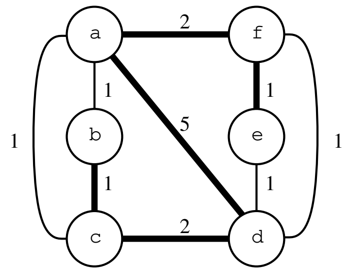


Figure 5: Covering on access graph

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 3.2), 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 3.1). ■

Theorem 3.1 allows us to arrive at an optimal simple offset assignment by solving the corresponding maximum weight path covering 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 weighted path covering (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).

The following theorem shows that the corresponding decision problem for MWPC is NP-complete.

Theorem 3.2 *Given an access graph G and a number n, the problem of deciding whether there* *exists a cover with weight greater than or equal to n is NP-complete.*

Proof MWPC $\in$ NP, since we can verify the weight of a given cover in linear time. We prove that MWPC is NP-complete by reduction from the Hamiltonian path problem. Given an undirected graph H, we assign a unit weight to each edge. Now a Hamiltonian path exists if and only if there exists a cover with weight equal to $n - 1$, where n is the number of nodes. ■

We have in essence *reduced* SOA to MWPC. The following theorem shows that SOA is also NP-complete; hence, MWPC is no harder than SOA.

Theorem 3.3 *Given an undirected graph $G\langle V, E\rangle$ and a weighting function $w : E \rightarrow \mathbf{N}$, there exists an access sequence with access graph $G' = G$ and weighting function $w' = 4w + 2$.*

Hence, 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 SOA/MWPC that is similar to Kruskal's maximum spanning tree algorithm [1]. 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, this algorithm can run in $O(E \log E)$ time. Since constructing the access graph requires $O(L)$ time, where L is the length of the procedure, the total running time is $O(E \log E + L)$. As our experimental results demonstrate, this heuristic often produces a solution quite close to the optimal solution.

As an example of applying the heuristic algorithm consider the access graph of Figure 3(c). We first pick edges $\langle$a,d$\rangle$, $\langle$a,f$\rangle$ and $\langle$c,d$\rangle$. We reject $\langle$a,b$\rangle$ and $\langle$a,c$\rangle$ because each causes a cycle.

SOLVE-SOA(L)
{
    / L = access sequence for basic block */*
    $G\langle V, E\rangle \leftarrow$ ACCESS-GRAPH(L);
    $\tilde{E} \leftarrow$ sorted list of edges in E
        in descending order of weight;
    $G'\langle V', E'\rangle \,:\, V' \leftarrow V,\ E' \leftarrow \phi$;
    while ($|E'| < |V| - 1$ and $\tilde{E} \neq \phi$) {
        choose $e \leftarrow$ first edge in $\tilde{E}$;
        $\tilde{E} \leftarrow \tilde{E} - e$;
        if ((e does not cause a cycle in G') and
            (e does not cause any node in V'
             to have degree > 2))
            add e to E';
        else
            discard e;
    }
    / Construct an assignment from E' */*
    return CONSTRUCT-ASSIGNMENT(E');
}

Figure 6: Heuristic Algorithm for SOA

Next, we pick $\langle$b,c$\rangle$. We reject $\langle$d,e$\rangle$ and $\langle$d,f$\rangle$ and finally pick $\langle$f,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 the generalization of SOA 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 cost of introducing the use of an address register. This set-up cost reflects the cost associated with initialization upon entry to the procedure and re-initialization 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, $\Pi = \{P_1, P_2, \ldots, P_m\}$, 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 six. 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 Figure 7(b) and (c). Assuming

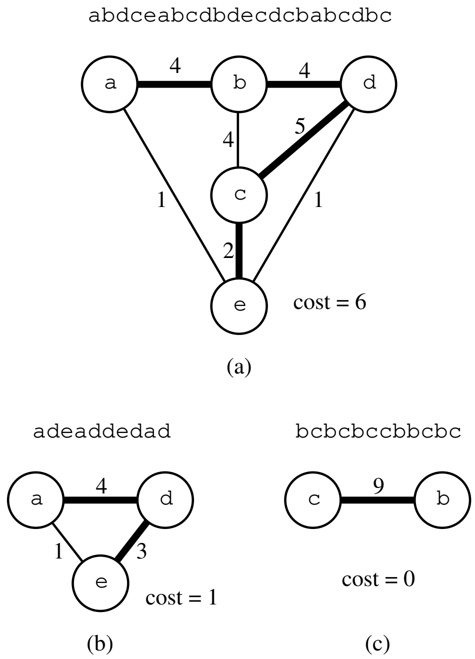


Figure 7: (a) Access sequence and graph (b) Access subsequence and graph generated by $\{$a,d,e$\}$ (c) Access subsequence and graph generated by $\{$b,c$\}$

a setup cost of one, the cost of using two address registers on this partition is two. In this case, there is an advantage in introducing a second address register.

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

```
1 SOLVE-GOA(L, k)
2 {
3     /* L = access sequence of basic block */
4     /* k = number of address registers */
5     H ← SOLVE-SOA(L);
6     if (k == 1)
7         return {H};
8     P ← SELECT-VARIABLES(L);
9     L1 ← SUBSEQ(L, P);
10    L2 ← SUBSEQ(L, L − P);
11    H1 ← SOLVE-SOA(L1);
12    H2 ← SOLVE-SOA(L2);
13    if (setup-cost + cost(H1) + cost(H2) > cost(H))
14        return {H};
15    else
16        return {H1} ∪ SOLVE-GOA(L2, k − 1);
17 }
```

Figure 8: Heuristic Algorithm for GOA

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:

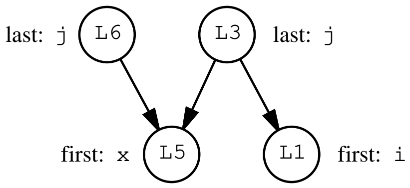


Figure 9: Fragment of a control-flow graph

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.

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.

Thus, a simple heuristic for SELECT-VARIABLES is to select the two variables with the largest penalty, and the cost we have to pay for allocating a new address register is just the setup cost. However, as we have observed in our experiments (see Section 6), it is sometimes more profitable to select more than two variables at once, and the best choice depends on the program itself. We are presently investigating more powerful techniques for performing this variable selection.

5 Offset Assignment for a Procedure

The access graph model for offset assignment gives exact results for basic blocks. However, in the presence of control-flow, modeling the exact cost of offset assignment is more difficult. To see this, consider the fragment of a control-flow graph shown in Figure 9, where each node denotes a basic block. The last variables accessed in blocks `L6` and `L3` are `j`, and the first variables accessed in blocks `L5` and `L1` are `x` and `i`, respectively.

Since `L3` can be followed by either `L5` or `L1`, it is not obvious whether we should add an edge between `j` and `x`, or between `j` and `i`, or both. Whichever we choose, the graph does not model the cost exactly. For instance, if we do both, and it turns out that in the final assignment `j` is adjacent to `i` (say immediately below `i`), then according to the access graph we need to pay a cost of two on the edge ⟨`j`,`x`⟩ for this portion of the control-flow graph due to the control-flow edges ⟨`L6`,`L5`⟩ and ⟨`L3`,`L5`⟩. However, if at the end of `L6` and `L3` we do perform auto-increment after accessing `j` so that upon exit on either block the address register points to `i`, then all we need to do is set the address register to point to `x`, thereby incurring a cost of one, rather than two. It is interesting to note that, even though at the end of `L6` auto-incrementing the address register does not make it point to `x`, we still perform auto-increment anyway. This is because in some machines, such as the TMS320C25, setting the address register from an unknown value has a higher cost than setting it from a known value. The former usually involves an immediate address which takes another instruction word, whereas the latter requires only an address arithmetic instruction. Hence, it is preferable that upon entering a basic block that the contents of the address register are known.

Our current approach is to merge the access graphs for all basic blocks, with equal weighting, and to treat variables connected by control-flow edges (e.g., ⟨`j`,`i`⟩ and ⟨`j`,`x`⟩) in the same way. Then, after offset assignment, a separate pass is used to determine whether auto-increment/decrement should be used at the end of each basic block.

If our goal is to optimize for execution speed, then this formulation will correctly reflect the actual cost (up to the accuracy of the execution frequency estimate of the basic blocks and control-flow edges). In this case, the access graph for each basic block is weighted by its estimated execution count (from either static estimation or profiling information), and likewise for control-flow edges.

program	decl. order	greedy SOA		b-b SOA		greedy GOA			b-b GOA		
	#inst.	#inst.	%red.	#inst.	%red.	#inst.	%red.	#sel.	#inst.	%red.	#sel.
chendct	756	730	3.5%	728	3.7%	651	13.9%	2	651	13.9%	2
chenidct	817	778	4.8%	776	5.0%	650	20.5%	3	650	20.5%	3
leedct	893	842	5.7%	841	5.8%	760	14.9%	4	760	14.9%	4
ileedct	1017	949	6.7%	948	6.8%	819	19.5%	6	815	19.9%	6
jrev	4296	4070	5.3%	4057	5.6%	3387	21.2%	3	3387	21.2%	3
readgif	648	599	7.6%	589	9.1%	550	15.2%	5	551	15.0%	3
autocrop	549	534	2.7%	531	3.3%	520	5.3%	2	520	5.3%	2
smooth	4002	3841	4.0%	3837	4.1%	3621	9.5%	3	3617	9.6%	3
hufftree	956	914	4.4%	907	5.1%	856	10.5%	3	858	10.3%	3
gnucrypt	3188	3063	3.9%	3065	3.9%	2958	7.2%	2	2955	7.3%	2

Table 1: Experimental results

6 Experiments and Results

We have implemented the heuristic algorithms of Sections 3 and 4. In addition, we have implemented a branch-and-bound procedure for solving SOA 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.

Table 1 shows our experimental results on ten routines typical of those found in DSP and embedded applications. The first five programs are core routines from a JPEG-MPEG package. The next three are graphics routines from the `xv` program. Hufftree is a routine from `gzip` that builds a Huffman encoding tree, and gnucrypt is the GNU implementation of the DES encryption algorithm.

The column labeled "decl. order" gives the (overall) size of each program for the offset assignment based on the order in which the variables are declared. Program sizes, along with the percentage reduction, obtained using the greedy heuristic of Figure 6 and using a branch-and-bound procedure are shown next. The last columns show the results of using GOA (Figure 8) with four address registers ($k = 4$). The column labeled "b-b GOA" uses the same GOA algorithm, but calls the branch-and-bound procedure for SOA instead of the greedy SOA heuristic. In all cases of GOA, a simple heuristic for SELECT-VARIABLES is used: select the m nodes with the highest penalty. However, we have observed that no fixed value of m gives the best result overall, and in Table 1 the best value of m is shown for each example (in the columns marked "# sel."). Hence, there are many opportunities to improve on the variable selection algorithm.

For simple offset assignment, the reduction in code size ranges from 3% to 9%. For general offset assignment, the reduction ranges from 5% to 20%. Thus, depending on the program (number of variables and the way they are accessed), GOA can potentially achieve substantial improvements.

In all cases the greedy heuristic arrives at solutions very close to the optimal. It is interesting to note that in a few cases the greedy heuristic for SOA actually outperforms the branch-and-bound procedure. The reason for this is that the heuristic and branch-and-bound procedure found different optimal solutions for the path-covering problem, and in the presence of control-flow the access-graph model does not exactly reflect the real cost, which depends on the actual offset assignment.

An obvious (though not very tight) lower bound for the offset assignment problem is the size of the program in which there are no address arithmetic instructions. Our experiments indicate that, after GOA, between 1/8 and 1/10 of the instructions are address arithmetic instructions.

7 Summary and Ongoing Work

The optimization techniques described in this paper are incorporated into our framework for developing compilers for embedded systems [3]. A diagram showing the stages of the compiler is shown in Figure 10.

We use SUIF [13] 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.

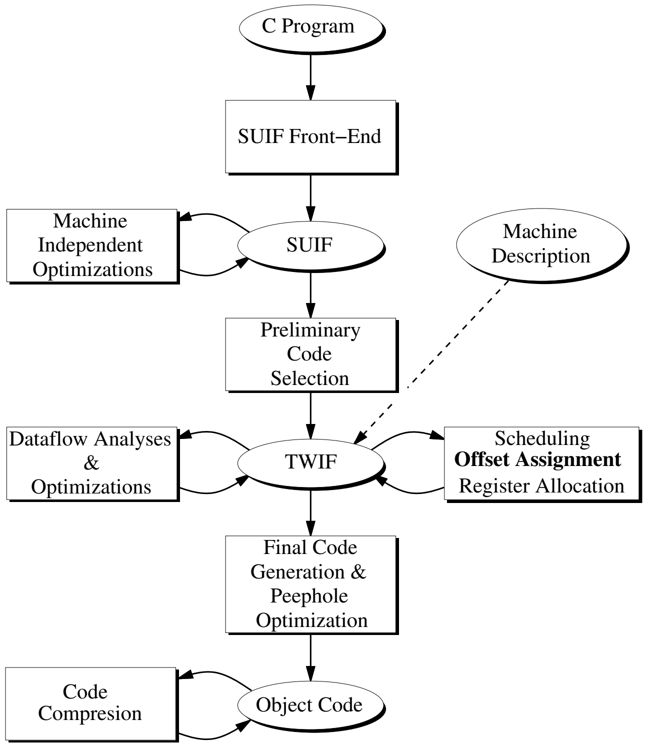


Figure 10: A framework for developing compilers for embedded systems

It is on this intermediate form that instruction scheduling, offset assignment, and register allocation are performed [10], along with machine-specific data-flow 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 [9] 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 [5, 6, 8, 12] 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 et al. [11] 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 paper we presented algorithms that are able to exploit the addressing mode features of most DSP processors. 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 many avenues for further work in offset assignment. Most importantly, scheduling affects access sequences and therefore leads to different offset assignment problems. This interaction needs to be taken into account in the scheduling process. The decision to use single static assignment or to merge variables with disjoint life-times affects the cost of offset assignment as well.

We are also addressing several other code optimization problems that arise in irregular data-paths [10]. Conventional register allocation is not possible for some DSP processors since the number of general-purpose registers available could be very small. Minimizing the number of accumulator spills becomes a relevant optimization problem. Finally, exploiting the different mode settings in instructions (e.g., unsigned versus signed arithmetic) affords the possibility of generating more compact code.

8 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 paper. 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.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. Challenges in Code Generation for Embedded Processors. pages 48–64.

[4] D. H. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software–Practice and Experience*, 22(2):101–110, February 1992.

[5] John R. Ellis. *A Compiler for VLIW Architectures*. MIT Press, 1985.

[6] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trnsactions on Computers*, C-30(7):478–490, 1981.

[7] J. G. Ganssle. *The Art of Programming Embedded Systems*. San Diego, CA: Academic Press, Inc., 1992.

[8] G. Goossens, J. Rabaey, F. Catthoor, J Vanhoof, R. Jain, H. De Man, and J. Vandewalle. A Computer-Aided Design Methodology for Mapping DSP Algorithms onto Custom Multiprocessor Architectures. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 924–925, May 1986.

[9] S. Liao, S. Devadas, and K. Keutzer. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. In *Proceedings of the Chapel Hill Conference on Advanced Research in VLSI*, March 1995.

[10] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimization Techniques for Embedded DSP Microprocessors. In *Proceedings of the 32nd Design Automation Conference*, June 1995.

[11] C. Liem, T. May, and P. Paulin. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. In *Proceedings of European Design and Test Conference*, March 1994.

[12] K. Rimey. *A Compiler for Application-Specific signal Processors*. PhD thesis, University of California, Berkeley, 1989.

[13] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: A Parallelizing and Optimizing Research Compiler. Technical Report CSL-TR-94-620, Stanford University, May 1994.