Address Register-Oriented Optimizations for Embedded Processors

J. Ramanujam *
Louisiana State University
jxr@ee.lsu.edu

Jinpyo Hong†
Louisiana State University
jphong1@ee.lsu.edu

Mahmut Kandemir‡
Pennsylvania State University
kandemir@cse.psu.edu

Sunil Atri§
Louisiana State University
sunil@ee.lsu.edu

Abstract

Embedded systems consisting of the application program ROM, RAM, the embedded processor core, and any custom hardware on a single wafer are becoming increasingly common in application domains such as signal processing. Given the rapid deployment of these systems, programming on such systems has shifted from assembly language to high-level languages such as C, C++, and Java. The processors used in such systems are usually targeted toward specific application domains, e.g., digital signal processing (DSP). As a result, these embedded processors include application-specific instruction sets, complex and irregular data paths, etc., thereby rendering code generation for these processors difficult. In this paper, we present new code optimization techniques for embedded fixed point DSP processors which have limited on-chip program ROM and include indirect addressing modes using post-increment and decrement operations. We present a heuristic to reduce code size by taking advantage of these addressing modes. Our solution aims at improving the offset assignment produced by Liao et al.'s solution. It finds a layout of variables in RAM, so that it is possible to subsume explicit address register manipulation instructions into other instructions as a post-increment or post-decrement operation. Experimental results show the effectiveness of our solution. Next, we propose an algorithm that uses commutative transformations to change the access sequence and thereby reducing the code size. Some DSP cores allow for the post-increment or decrement value to be larger than one. For such processors, we also present an approach that is incremental and has some advantages over another proposed solution that requires the expensive generation of cliques.

1 Introduction

With the falling cost of microprocessors and the advent of very large scale integration, more and more processing power is being placed in portable electronic devices [6, 9, 10, 13]. Such processors (in particular, fixed-point DSPs and micro-controllers) can be found, for example in audio, video, and telecommunications equipment and have severely limited amounts of memory for storing code and data, since the area available for ROM and RAM is limited. This renders the efficient use of memory area very critical. Since the program code resides in the on-chip ROM, the size of the code directly translates into silicon area and hence the cost. The minimization of code size is, therefore, of considerable importance [1, 3, 5, 6, 7, 8, 9, 14, 15, 16, 17], while simultaneously preserving high levels of performance. However, current compilers for fixed-point DSPs generate code that is quite inefficient with respect to code size and performance. As a result, most application software is hand-written or at least hand-optimized, which is a very time consuming task [8]. The increase in developer productivity can therefore be directly linked to improvement in compiler techniques and optimizations.

Many embedded processor architectures such as the TI TMS320C25 include indirect addressing modes with *auto-increment* and *auto-decrement* arithmetic. This feature allows address arithmetic instructions to be part of other instructions. Thus, it eliminates the need for explicit address arithmetic instructions wherever possible, leading to decreased code size. The memory access pattern and the placement of variables has a significant impact on code size. The auto-increment and auto-decrement modes can be better utilized if the placement of variables is performed after code selection. This delayed placement of variables is referred to as *offset assignment.*

This paper considers the *Simple Offset Assignment* (SOA) problem where there is just one address register. A solution to the problem assigns optimal frame-relative offsets to the 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. The problem is modeled as follows. A basic block [11] is represented by an *access sequence*, which is a sequence of variables written out in the order in which they are accessed in the high level code. This sequence is in turn further condensed into a graph called the *access graph* with weighted undirected edges. The SOA problem is equivalent to a graph covering problem, called the *Maximum Weight Path Cover* (MWPC) problem. A solution to the MWPC problem gives a solution to the SOA problem. We present a new algorithm, called $Incremental\text{-}Solve$-SOA, for the SOA problem and compare its performance with previous work on the topic. Next, we present heuristics for exploiting commutativity of algebraic expressions which can further lead to reduced code size. In addition, we discuss a variant this algorithm where the value for post-increment/decrement can be larger than one.

The remainder of this paper is organized as follows. We present a brief explanation of graphs and some additional required notation and background in Section 2. Then, in Section 3, we consider the problem of storage assignment, where the arithmetic permitted on the address register is plus or minus 1. We present our experimental results in Section 4. We then propose a heuristic in Section 5 that modifies the access sequence by exploiting operator commutativity

* **Corresponding author.** Dept. Electrical & Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA.

† Dept. Electrical & Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA.

‡ Dept. Computer Science & Engineering, Pennsylvania State University. University Park, PA 16802, USA.

§ Dept. Electrical & Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA. Currently with Intel Corporation, Folsom, CA, USA.

to improve the effectiveness of offset assignment. We discuss an extension of our technique for non-unit post-increment (or decrement) in Section 6. We conclude the paper with a summary in Section 7.

2 Background

We model the sequence of data accesses as weighted undirected graphs [8]. Each variable in the program corresponds to a vertex (or node) in the graph. An edge i, j indicates that variable i is accessed after j or vice-versa; the weight of an edge $w(i, j)$ denotes the number of times variables i and j are accessed successively.

Definition 1 *Two paths are said to be disjoint if they do not share any vertices.*

Definition 2 A disjoint path cover *(which will be referred to as just a 'cover') of a weighted graph $G(V, E)$ is a subgraph $C(V, E')$ of G such that, for every vertex v in C, $deg(v) < 3$ and there are no cycles in C. The edges in C may be a non-contiguous set [4].*

Definition 3 *The* weight *of a cover C is the sum of the weights of all the edges in C [6]. The cost of a cover C is the sum of the weights of all edges in G but not in C:*

$$cost(C) = \sum_{(e \in E) \wedge (e \notin C)} w(e)$$

2.1 Motivating Example

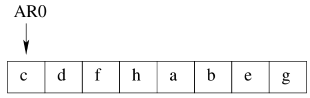
As mentioned earlier, many embedded processors provide register indirect addressing modes with auto-increment and auto-decrement arithmetic [6]. It is possible to use these modes for efficient sequential access of memory and improve code density. The placement of variables in memory has a large impact on the exploitation of auto-increment and auto-decrement addressing modes, which is in turn affected by the pattern in which variables are accessed. If the assignment of location of variables is done after code selection, then we get the freedom of assigning locations to variables depending on the order in which the variables are accessed. The placement of variables in storage has a considerable impact on code size and performance.

Consider the C code sequence shown in Figure 1(a), an example from [6]; let the placement of variables in memory be as in Figure 1(b). This assignment of variables to memory locations here is based on *first use*, i.e., as the variables are referred to in the high level code, the variables are placed in memory. The assembly code for this section of C code is shown in Figure 1(c). The first column shows the assembly code, the second column shows the register transfer, and the third, the contents of the address pointer. The instructions in bold are the explicit address pointer arithmetic instructions, i.e., **SBAR**, Subtract Address Register and **ADAR**, *Add Address Register. The objective of the solution to the SOA problem is to find the minimal address pointer arithmetic instructions required using proper placement of variables in memory.* A brief explanation of Figure 1 follows.

The first instruction LDAR AR0, &c loads the *address* of the first variable 'c' into the address register AR0. The next instruction LOAD (AR0)+ loads the variable 'c' into the accumulator. This instruction shows the use of the auto-increment mode. Ordinarily, we would need an explicit pointer increment to get it to point to 'd' which is the next required variable, but it is subsumed into the LOAD instruction in the form of a post-increment operation, indicated by the trailing '+' sign. The pointer decrement operation can also be similarly subsumed by a post-decrement operation indicated by a trailing '-' sign for example as in the ADD *(AR0)- instruction. It can be seen, that the instructions in bold are the ones

```
c = c + d + f;
a = h + c;
b = b + e;
c = g – b;
a = a – c;
```

(a)



(b)

LDAR AR0, &c		*AR0* ← *&c*
LOAD *(AR0)+	*acc* ← *c*	*AR0* ← *&d*
ADD *(AR0)+	*acc* ← *acc + d*	*AR0* ← *&f*
ADD *(AR0)	*acc* ← *acc + f*	
SBAR AR0, 2		*AR0* ← *&c*
STOR *(AR0)	*c* ← *acc*	
ADAR AR0, 3		*AR0* ← *&h*
LOAD *(AR0)	*acc* ← *h*	
SBAR AR0, 3		*AR0* ← *&c*
SUB *(AR0)	*acc* ← *acc – c*	
ADAR AR0, 4		*AR0* ← *&a*
STOR *(AR0)+	*a* ← *acc*	*AR0* ← *&b*
LOAD *(AR0)+	*acc* ← *b*	*AR0* ← *&e*
ADD *(AR0)–	*acc* ← *acc + e*	*AR0* ← *&b*
STOR *(AR0)	*b* ← *acc*	
ADAR AR0, 2		*AR0* ← *&g*
LOAD *(AR0)	*acc* ← *g*	
SBAR AR0, 2		*AR0* ← *&b*
SUB *(AR0)	*acc* ← *acc + b*	
SBAR AR0, 5		*AR0* ← *&c*
STOR *(AR0)	*c* ← *acc*	
ADAR AR0, 4		*AR0* ← *&a*
LOAD *(AR0)	*acc* ← *a*	
SBAR AR0, 4		*AR0* ← *&c*
SUB *(AR0)	*acc* ← *acc – c*	
ADAR AR0, 4		*AR0* ← *&a*
STOR *(AR0)	*a* ← *acc*	

(c)

Figure 1: Code example from [5,6].

that do only address pointer arithmetic in Figure 1(a). *The number of such instructions in the generated code may be very high, as typically the high-level programmer does not consider the variable layout while writing the program.* AR0 is auto-incremented after the first LOAD instruction. Now, AR0 is pointing to 'd', as 'd' is the next variable required, so it can be accessed immediately without having to change AR0. Similarly for variable 'f', the next variable required is 'c', which is at a distance 2 from 'f'. Consider the STOR instruction that writes the result back to 'c', an explicit SBAR AR0, 2 instruction has to be used to set AR0 to point to 'c', because the address of 'f' and that of 'c' differ by two and auto-decrement cannot be used along with the previous ADD instruction. This can be seen in the other instances of ADAR and SBAR, where for every pair of accesses that do not refer to adjacent variables, either an SBAR or ADAR instruction must be used. In total, ten such instruction are needed to execute the code in Figure 1(a), given the offset assignment of Figure 1(b).

2.2 Assumptions in SOA

The simple offset assignment (SOA) problem is one of assigning a frame-relative offset to each local variable to *minimize* the number of address arithmetic instructions (ADAR and SBAR) required to execute a basic block. The cost of an assignment is hence defined by the number of such instructions. With a single address register, the initializing LDAR instruction is *not* included in this cost. We make the following assumptions for the SOA problem: (1) every data object is of size one word; (2) a single address register is used

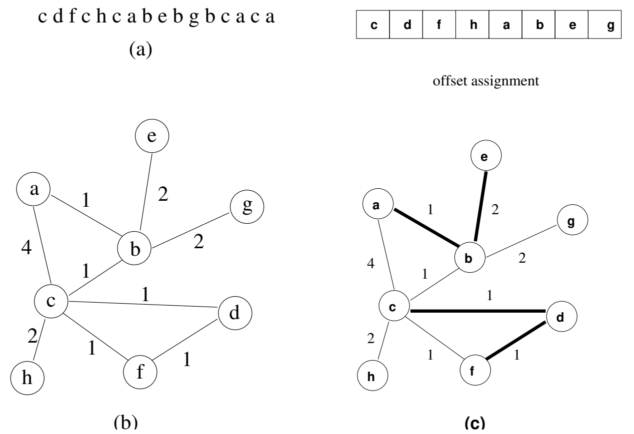


Figure 2: (a) Access sequence; (b) Access graph; (c) Offset assignment and cover C (thick edges).

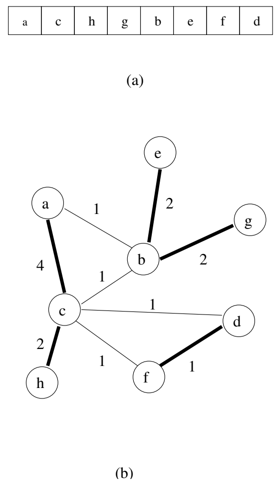


Figure 3: Optimal offset assignment and cover C (thick edges).

to address all variables in the basic block; (3) one-to-one mapping of variables to locations; (4) the basic block has a fixed evaluation order; (and) (5) special features such as address wrap-around are not considered.

2.3 Approach to the Problem

The SOA problem can be formulated as a *graph covering problem*, called the *Maximum Weight Path Covering Problem* (MWPC) [6, 7]. From a basic block, a graph, called the *access graph* is derived, that shows the various variables and their relative adjacency and frequency of accesses. From the solution to the MWPC problem, a minimum cost assignment can be constructed.

Given a code sequence S that represents a basic block, one can define a unique *access sequence* for that block [7]. In an operation '$z = x\ op\ y$', where '*op*' is some binary operator, the access sequence is given by 'xyz'. The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation in the appropriate order. For example, the access sequence for the C code example in Figure 1(a) is shown in Figure 2(a).

With the definition of cost given earlier, it can be seen that the cost is the number of consecutive accesses to variables that are not assigned to adjacent locations. The access sequence is the sequence of memory references made by a section of code and it can be obtained from the high-level language program. The access sequence can be summarized in an edge weighted, undirected graph. The access graph $G(V, E)$ is derived from an access sequence as follows. Each vertex $v \in V$ of the access graph corresponds to a unique variable in the basic block. An edge $e(u, v) \in E$ between vertices u and v exists with weight $w(e)$ if variables u and v are adjacent to each other $w(e)$ times in the access sequence. The order of the accesses is *not* significant as either auto-increment or auto-decrement can be performed. The access graph for Figure 2(a) is shown in Figure 2(b).

2.4 SOA and Maximum Weight Path Cover

Given the definitions earlier in this paper, if a maximum weight cover for a offset assignment graph is found, then that also means that the minimum cost assignment has also been found. 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 [6]. Given an offset assignment A and an access graph G, there exists a disjoint path cover which implies A and which has the same cost as A. Every offset assignment implied by an optimal disjoint path cover is optimal. An example of a sub-optimal path cover is shown in Figure 2(c).

The thick lines show the disjoint path cover and the corresponding offset assignment is also shown. The cost of this assignment is 10. This can be seen from the edges not in the cover.

3 An Incremental Algorithm for SOA

3.1 Previous Work

Bartley [3] and Liao [6, 7] studied the simple offset assignment problem. Liao formulated the simple offset assignment problem. The problem was modeled as a graph theoretic optimization problem similar to Bartley [3] and shown to be equivalent to the Maximum Weighted Path Cover (MWPC) problem. This problem is proven to be NP-hard. A heuristic solution to the above problem proposed by Liao will be explained in the following subsection. Consider the example shown earlier. Using Liao's algorithm we get an offset assignment as shown in Figure 3(a) which is implied by the access graph in Figure 3(b). The cover of the access graph is shown by the heavy edges, and in this case it is optimal. This can be seen from the graph itself. Picking any of the four non-included edges will cause the dropping of some edge from the cover which will in turn increase the cost of the cover. The assembly code for the offset assignment implied by the cover is shown in Figure 4(c). The address arithmetic instructions are highlighted and there are four such instructions corresponding to the four edges not in the cover. For example because 'a' and 'b' could not be placed adjacent to each other, we need to use the instruction ADAR *(AR0) 4. The offset assignment that this section of code needs to use is shown in Figure 4(b), along with the C code (Figure 4(a)) for reference. Leupers and Marwedel [5] present a heuristic for choosing among different edges with the same weight in Liao's heuristic.

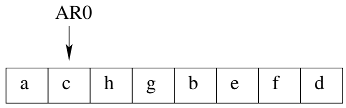
3.2 Liao's Heuristic for SOA

Because SOA and MWPC are NP-hard, a polynomial-time algorithm for solving these problems optimally is not likely to exist unless P=NP. Liao's heuristic for the simple offset assignment problem is shown in Figure 5. This algorithm is similar to Kruskal's minimum spanning tree algorithm [4]. The heuristic is greedy, in the sense that it repeatedly selects the edge that seems best at the current moment.

Consider the algorithm $Solve$-SOA(L) in Figure 5. This algorithm takes as input a sequence 'L' which uniquely represents the high level code, and produces as output an offset assignment. In line 4, graph $G(V, E)$ is produced from the access sequence 'L'.

c = c + d + f;
a = h – c;
b = b + e;
c = g – b;
a = a – c;

(a)



(b)

LDAR AR0, &c		*AR0* ← *&c*
LOAD *(AR0)+	*acc* ← *c*	
ADAR AR0, 6		*AR0* ← *&d*
ADD *(AR0)–	*acc* ← *acc + d*	*AR0* ← *&f*
ADD *(AR0)	*acc* ← *acc + f*	
SBAR AR0, 5		*AR0* ← *&c*
STOR *(AR0)+	*c* ← *acc*	*AR0* ← *&h*
LOAD *(AR0)–	*acc* ← *h*	*AR0* ← *&c*
SUB *(AR0)–	*acc* ← *acc – c*	*AR0* ← *&a*
STOR *(AR0)	*a* ← *acc*	
ADAR AR0, 4		*AR0* ← *&b*
LOAD *(AR0)+	*acc* ← *b*	*AR0* ← *&e*
ADD *(AR0)–	*acc* ← *acc + e*	*AR0* ← *&b*
STOR *(AR0)–	*b* ← *acc*	*AR0* ← *&g*
LOAD *(AR0)+	*acc* ← *g*	*AR0* ← *&b*
SUB *(AR0)	*acc* ← *acc – b*	
SBAR AR0, 3		*AR0* ← *&c*
STOR *(AR0)–	*c* ← *acc*	*AR0* ← *&a*
LOAD *(AR0)+	*acc* ← *a*	*AR0* ← *&c*
SUB *(AR0)–	*acc* ← *acc – c*	*AR0* ← *&a*
STOR *(AR0)	*a* ← *acc*	

(c)

Figure 4: Code after the optimized offset assignment.

```
1  // INPUT :  Access Sequence, L
2  // OUTPUT :  Constructed Assignment E′
3  Procedure Solve − SOA(L)
4  G(V, E) ← AccessGraph(L)
5  E_sort ← Sorted edges in E in descending order of weight
6  // Initialize C(V′, E′) the constructed cover
7  E′ ← { }
8  V′ ← V
9  while (|E′| < |V| − 1 and E_sort not empty) do
10    e ← first edge in E_sort
11    E_sort ← E_sort − e
12    if ((e does not cause a cycle in C) and
13      (e does not cause any vertex in V′ to have degree > 2))
14      add e to E′
15    else
16      discard e from E_sort
17    endif
18 enddo
19 return E′
```

Figure 5: Liao's maximum weight path cover heuristic [6].

Producing the access sequence takes $O(L)$ time. Line 5 produces a list of sorted edges in descending order of weight. $C(V', E')$ is the cover of the graph G, which starts with all the vertices included but no edges. The condition for the while statement makes sure that no more than $V-1$ edges are selected, as that is the maximum needed for any cover. If the cover is disjoint, the order in which the disjoint paths are positioned does not matter as far as the cost of the offset assignment is concerned, because the cost of moving from one path to another will always have to be paid. The complexity of Liao's heuristic is $O(|E| \log |E| + |L|)$ [6], where $|E|$ is the number of edges in the access graph and $|L|$ is the length of the access sequence. Construction of the access sequence takes $O(|L|)$ time. The $(|E| \log |E|)$ term is due to the need to sort the edges in descending order of weight. The main loop of the algorithm runs for $|V|$ iterations. The test for a cycle in line 12 takes constant time, and the total time for the main loop is bounded by $O(E)$. The test for a cycle is achieved in constant time by using a special data structure proposed by Liao [6].

3.3 Improvement over Solve-SOA

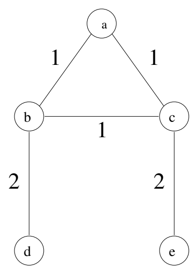
Before suggesting the improvement, we want to point out two deficiencies in $Solve$-SOA. First, even though the edges are sorted in descending order of weight, the order of consideration of the edges of the same weight are ordered is not specified. We believe this to be important in deriving the optimal solution. Second, the maximum weight edge is always selected since this is a greedy approach. The proposed $Incremental$-$Solve$-SOA heuristic addresses both

d = a + b;
e = b - c;
a = c + 2;

(a)

a b d b c e c a

(b)



(c)

Figure 6: An example where $Solve$-SOA could possibly return suboptimal results.

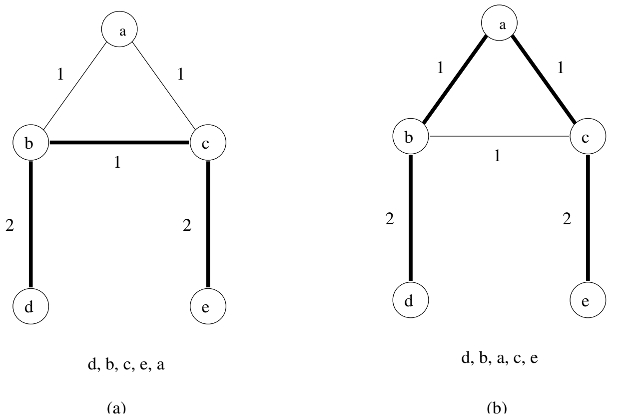


Figure 7: Suboptimal and optimal cover of G.

these cases. This algorithm takes as input an offset sequence, produced either by Liao's $Solve$-SOA or by some other means, and tries to include edges not previously included in the cover. Consider the example of Figure 6. The code sequence is shown in Figure 6(a) and the corresponding access sequence is in Figure 6(b). The access graph which in turn corresponds to this access sequence is shown in Figure 6(c). Let us now run Liao's $Solve$-SOA using the access sequence in Figure 6(b); one possible outcome is shown in Figure 7(a). The offset assignment associated with the cover is a, d, b, c, e or d, b, c, e, a. This is clearly a non-optimal solution. The cost of this assignment is 2. The optimal solution would be d, b, a, c, e. It is possible to have achieved the optimal cost of 1 by having considered either edge (a, b) or edge (a, c) before edge (b, c). But since $Solve$-SOA does not consider the relative positioning of the edges of the same weight in the graph, we get the cost of 2. The solution that is produced by the proposed $Incremental$-$Solve$-SOA is d, b, a, c, e as shown in Figure 7(b).

3.3.1 Incremental-Solve-SOA

Figure 9 shows the proposed algorithm. The algorithm picks the maximum weight edge not included in the cover and tries to include that. This is done as follows. Let the maximum weight edge not included in the cover be between two variables $a(n)$ and $a(n+x)$, in that order. We consider the case where we try to include that edge and see the effect on the cost of an assignment. There are four offset assignments when we try to bring two variables together not previously adjacent. The initial offset assignment is $...a(n-1)a(n)a(n+1)...a(n+x-1)a(n+x)a(n+x+1)....$ We consider the following four sequences that would result when edge $(a(n)a(n+x))$ is included in the cover:

(1) $...a(n-1)a(n+x)a(n)a(n+1)...a(n+x-1)a(n+x+1)...$

(2) $...a(n-1)a(n)a(n+x)a(n+1)...a(n+x-1)a(n+x+1)...$

(3) $...a(n-1)a(n+1)...a(n+x-1)a(n)a(n+x)a(n+x+1)...$

(4) $...a(n-1)a(n+1)...a(n+x-1)a(n+x)a(n)a(n+x+1)...$

The cost of each of these is evaluated and the best assignment, i.e., the one with the least cost of the four is chosen for the next iteration. A running minimum cost assignment, BEST, is used to store the best assignment discovered. This is returned at the end of the procedure.

Theorem 1 *The $Incremental$-$Solve$-SOA will either improve or return the same cost assignment.*

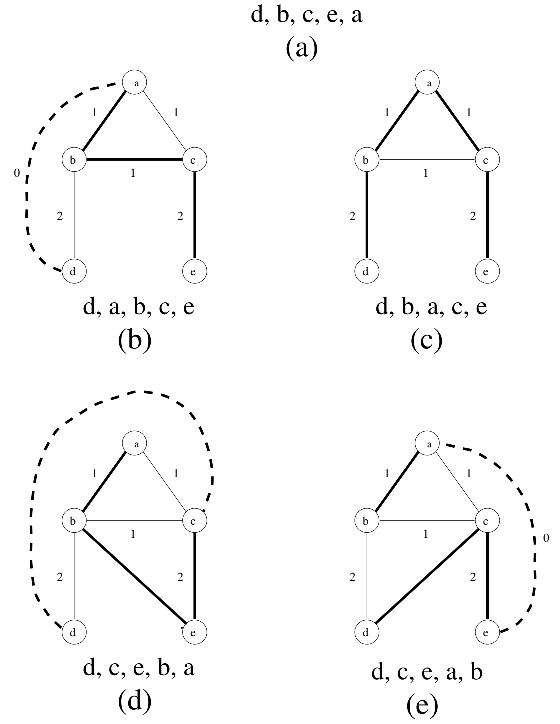


Figure 8: Four different offset assignments.

$Proof$: As different assignments with different costs are produced, a running minimum is maintained. If the minimum is the initial assignment that is the one considered again, and finally returned when all the edges are locked, or there are no non-zero edges available for inclusion. □

In the example, the initial assignment is d, b, c, e, a, that has a cost of 2. Let us try to include edge (b, a). The resulting four assignments for the initial assignment of d, b, c, e, a with cost = 2 are:

(1) d, a, b, c, e cost = 3

(2) d, b, a, c, e cost = 1

(3) d, c, e, b, a cost = 4

(4) d, c, e, a, b cost = 4

These four assignments are shown in Figure 8(b)-(e). Figure 8(a) shows the initial offset assignment for the purpose of comparison.

3.3.2 Detailed Explanation of the Incremental-Solve-SOA

The input is an access sequence and the initial offset assignment, that we will attempt to improve upon. The output is the possibly improved offset assignment. In line 4, we call the function $AccessGraph$ to obtain the access graph from the access sequence. BEST is a data structure that stores an offset assignment along with its cost. It is initialized with the input offset assignment in line 5. Lines 6 thru 34 is the outer loop. The exit condition for this loop is the OUTER_FLAG being set to FALSE. Line 7 produces E^{U}_{sort} holds the sorted list of edges present in the access graph but not in the cover, in decreasing order of weight. This is done so as to be able to consider edges for inclusion in the order of decreasing weight. The edges carry a flag that is used for 'locking' or 'unlocking' edge. The 'lock' on an edge, if broken, is

```
1  //INPUT : Access Sequence AS, Init. Offset Assignment, OA
2  //OUTPUT : Final Offset Assignment
3  Procedure Incremental-Solve-SOA(AS, OA)
4  G = (V, E) ← AccessGraph(AS)
5  BEST ← Initial offset assignment OA
6  repeat
7    E^U_sort ← Sorted list of unselected edges from BEST
8    OUTER_FLAG ← FALSE
9    Unlock all edges in E^U_sort
10   INNER_BEST ← BEST
11   repeat
12     INNER_FLAG ← FALSE
13     e ← topmost edge from E^U_sort
14     (A_0, ..., A_3) ← The four possible assignments due to e
15     // An assignment is illegal if it involves changing a
16     // locked edge; Otherwise, an assignment is legal
17     S ← the set of legal assignments from (A_0, ..., A_3)
18     if (S has at least one legal assignment)
19       INNER_FLAG ← TRUE
20       CURRENT ← MinCost(S)
21       lock the edges that change
22       Delete locked edges from E^U_sort keeping E^U_sort sorted
23       if (CostOf(CURRENT) < CostOf(INNER_BEST))
24         INNER_BEST ← CURRENT
25       endif
26     else (E^U_sort ≠ φ)
27       INNER_FLAG ← TRUE
28     endif
29   until (INNER_FLAG ≠ TRUE)
30   if (CostOf(INNER_BEST) < CostOf(BEST))
31     BEST ← INNER_BEST
32     OUTER_FLAG ← TRUE
33   endif
34 until (OUTER_FLAG ≠ TRUE)
35 return BEST
```

Figure 9: Incremental-Solve-SOA

used to indicate an edge available for inclusion in the cover. Lines 11 thru 29 form the inner loop. The exit condition for this loop the INNER_FLAG being FALSE. In line 13, the top most edge is extracted from E^U_{sort} and the four assignments are produced as explained in the earlier section on reordering of variables. These four assignments are stored in $(A_0, ..., A_3)$. The cover formed by each is checked to see if a locked edges not included earlier in being included, or if a locked edge included in earlier is being excluded. The assignments where this does not happen are included in S. This is done line 17. Of these the legal assignments are stored in the set S in line 17. In line 18 if there is at least one assignment available, set S is not empty, then the minimum cost one of those is assigned to CURRENT in line 20 and INNER_BEST is set to TRUE. The edges which undergo transitions as explained earlier are locked in line 21. INNER_BEST maintains a running minimum cost assignment for the inner loop, and if the CURRENT cost is less than INNER_BEST, then that is made the INNER_BEST. E^U_{sort} is reassigned for the list of unselected and unlocked edges from the CURRENT cover in line 22. If no legal assignments could be found for the edge extracted from E^U_{sort} in line 17, and there is at least another edge available for consideration, then the INNER_FLAG is set to TRUE. This is done in lines 26 and 27. Once there are no more legal assignments and there are no more edges in E^U_{sort} available, we exit the inner loop and check if the cost of INNER_BEST is less than the BEST found. If there is an improvement we perform the whole process of the inner loop all over again. This is made possible be setting OUTER_FLAG to TRUE. If no improvement was found, then we exit the outer loop too and the BEST offset assignment discovered is returned in line 35.

Table 1: Results from $Incremental\text{-}Solve$-SOA as compared to $Solve$-SOA.

Number of Variables	Size of Access Sequence	Percentage of Cases Improved	Percent of Improvement in Improved Cases
5	10	2.4	37.08
5	20	4.6	19.00
8	16	4.0	17.13
8	30	7.4	8.71
8	40	6.0	6.85
10	50	7.8	4.87
10	100	5.4	2.41
15	20	4.8	12.66
15	30	4.4	7.46
15	56	5.4	3.29
15	75	5.4	2.37
20	40	2.8	5.38
20	75	4.0	2.71
20	100	3.4	1.92

3.4 Complexity of Incremental-Solve-SOA

As mentioned before, the running time of Liao's $Solve$-SOA heuristic is $O(|E|\log|E| + |L|)$, where $|E|$ is the number of edges in the access graph and $|L|$ is the length of the access sequence. The running time of the $Incremental\text{-}Solve$-SOA is $O(|E|)$ for the inner while loop. Sorting the edges in descending order of weight for E^U_{sort} takes $O(|E|\log|E|)$ time,and the marking of the edges is $O(|E|)$, the number of iterations of the outer loop in our experience runs for a constant number of times, an average of 2. So, the complexity of each outer loop iteration in practice is $O(|E|\log|E|)$. This is the same as Liao's, though in practice we need to incur a higher overhead; but, the use of our heuristic is justified by the fact that the code produced by this optimization will be executed many times whereas the compilation is done only once. Also, its use could possibly result in a smaller ROM area.

4 Experimental Results

We implemented both $Solve$-SOA and $Incremental\text{-}Solve$-SOA. The results shown in Table 1 are for the case where the initial offset assignment used was the result of using Liao's heuristic (If the initial offset assignment is an unoptimized one, the improvements will be much higher).

The experiments were performed by generating *random access sequences* and using these as input to Liao's heuristic. The offset sequence returned was then used, in turn, along with the access sequence in the $Incremental\text{-}Solve$-SOA to produce a possible change in the offset sequence. This change is guaranteed to be the same or better as reflected in Table 1. The third column shows the percentage improvement in the number of cases is of relevance here, as it shows an improvement in the cost of the *cover* of the access graph. It is always possible to increase all the edge weights in the access graph by some constant value to achieve a higher magnitude improvement for the same change in cover, but the change in cover would still be the same.

In Table 1, the first column lists the number of variables, the second column lists the size of the access sequence. The third column shows the average improvement in the number of cases. That is, for example in the first row, there was an improvement on an average of 2.4% of all the random access sequences considered. The fourth column shows, of the improved cases, the extant of improvement. For the first row that would be 37.08% improvement in 2.5% of the cases.

The overall average improvement (in the third column) is 5.23%. This figure reflects the cases in which $Incremental$-$Solve$-SOA was able to improve upon the cover of the access graph given the offset assignment produced from Liao's $Solve$-SOA as input. The improvement takes significance from the many times the code would be executed, and also that it would result in a saving of ROM area.

5 Commutative Transformations

The access sequence has a great impact on the cost of offset assignment. We explore opportunities for better assignments that can be derived by exploiting the commutativity and other properties of operators in expressions. We propose a heuristic that attempts to modify the access sequence so as to achieve savings in the cost of the final offset assignment. We base our heuristic on the assumption that reducing the number of edges in an access graph will lead to a low cost assignment (see [1] for a justification). Towards this end, we identify edges that can possibly be eliminated by a reordering transformation of the access sequence, which is in turn (because of a transformation in a statement in the C code basic block) based on the rule of commutativity. The edges we target for this kind of transformation are those with weight one. We consider the case where there are only two or less variables on the RHS of a statement. The statements considered are those that are amenable to commutative reordering. Let us focus now on the set of ordered statements as in Figure 10(a); they have an access sequence as shown in Figure 10(b). The variables shown in the boxes are those whose relative position *cannot* be changed. A change in l_2 for example will require statement reordering as shown in Figure 10(c). The weight change will be:

(1) $w(l_2, f_3)$--	(2) $w(l_1, f_2)$--	(3) $w(l_0, f_1)$--
(4) $w(l_1, f_3)$++	(5) $w(l_2, f_1)$++	(6) $w(l_0, f_2)$++

where the trailing ++ or -- indicate a increase in weight by one or a decrease in weight by one, respectively. This can be seen in Figure 10(c). We will explore variable reordering due to commutativity rather than statement reordering; the latter requires data dependence analysis.

There are different ways to evaluate the cost or benefit of a transformation. The one we propose is computationally much less expensive than Rao and Pande's procedure [14, 1]. Consider again the access sequence shown in Figure 10(b). Without loss of generality, we can assume that the weight of the edge $w(s_2, l_2) = 1$. Then if we try the sequence shown in Figure 10(b), we have a modification in the access sequence, which, corresponds to the commutative transformation shown in Figure 10(d). This transformation can now be evaluated in two ways. The first, is to do a local evaluation of the cost, and the second one, is to run Liao's algorithm (or Incremental-Solve-SOA) on the changed access sequence. We propose two procedures based on these two methods of evaluating benefit.

We discuss the local evaluation first. As before, consider Figure 10(b) which is the access sequence, with the variables that cannot be moved marked by a box. As we had assumed the weight of the edge (s_2, l_2) was one and also that reducing the number of edges is possibly beneficial, reducing the weight of (s_2, l_2) to zero will effectively remove the edge from the access graph. If we wish to reduce $w(s_2, l_2)$ from one to zero then the following four edge weight reductions will occur.

(1) $w(l_1, s_2)$++

(2) $w(l_1, f_2)$- -

(3) $w(s_2, l_2)$- -; note that $w(s_2, l_2) = 0$ is possible here

(4) $w(f_2, l_2)$++

We define the *primary benefit* as the following value: (the number of non-zero edges turning zero) $-$ (number of zero edges turning non-zero). In addition, we define a *secondary benefit* $=$ (the sum of the increases in the weights of already present edges) $+$ (the sum of the increases in the weights of self-edges).

If there are two edges with the same primary benefit, then the tie-break between them is done using the secondary benefit measure. The second method of evaluating benefit is to run Liao's heuristic, compute the cost, and select the edge giving the least cost. Of course, this option has a higher complexity.

5.1 Detailed Explanation of the Commutative Transformation Heuristic

We now discuss in detail our commutative transformation heuristic, called Commutative-Transformation-SOA(AS), given in Figure 12. This heuristic is greedy, in that, it performs the transformation that appears to be the best at that point in time. Lines 1 and 2 indicate that the input is the unoptimized access sequence AS and the output is the optimized access sequence AS°. We build the access graph by a call to the $AccessGraph$(AS) function. We also initialize the optimized access sequence AS° with the starting access sequence AS. Lines 8 to 14 constitute the first loop. Here we assign the Primary Benefit and Secondary Benefit measure to the edges of weight one. Edges with negative primary benefit are not considered and so we compute the secondary benefit only if there is a non-negative primary benefit. The data structure $\mathsf{E}^1_{\mathsf{sort}}$ is used to hold the various edges in descending order of the primary benefit, and if the the primary benefit is the same, then the secondary benefit is used as a tie-breaker in finding the ordering of the edges. T hold the set of edges that are considered as a set in the **while** loop. The compatibility of an edge is required as it is possible that a transformation made to AS° could be undone by the transformation motivated by another incompatible edge. As an example, consider Figure 11(a). The primary benefit of removing edge (a, b) from the access graph is 2, as edges (a, b) and (c, d) become zero weight edges. Now, if we try to remove edge (d, a) from the access graph, whose primary benefit is 1, we reintroduce edges (a, b) and (c, d), while removing edge (d, a). This is not the desired change, so when the transformation to remove edge (a, b) is made then the edge (d, a) should not be considered anymore.

We have included the **if** statement since, if there is no primary benefit, i.e., Primary Benefit is zero, it need not be included for consideration in the subsequent **while** loop. In the **while** loop from line 19 to line 27 for the same highest primary and secondary benefit measure, the maximal set of compatible edges are extracted from $\mathsf{E}^1_{\mathsf{sort}}$ and assigned to H. Transformation to AS° is performed in line 25. Once the transformations are done, The set of chosen edges is updated to contain the edges in C. Finally, when $\mathsf{E}^1_{\mathsf{sort}}$ is empty, we return the optimized access sequence AS°.

5.2 Commutative Transformation Heuristic Example

We now show the working of the heuristic through an example. Consider the access sequence Figure 13(b), with the resulting access graph in Figure 13(c). The edges of weight 1 are: (a, f),

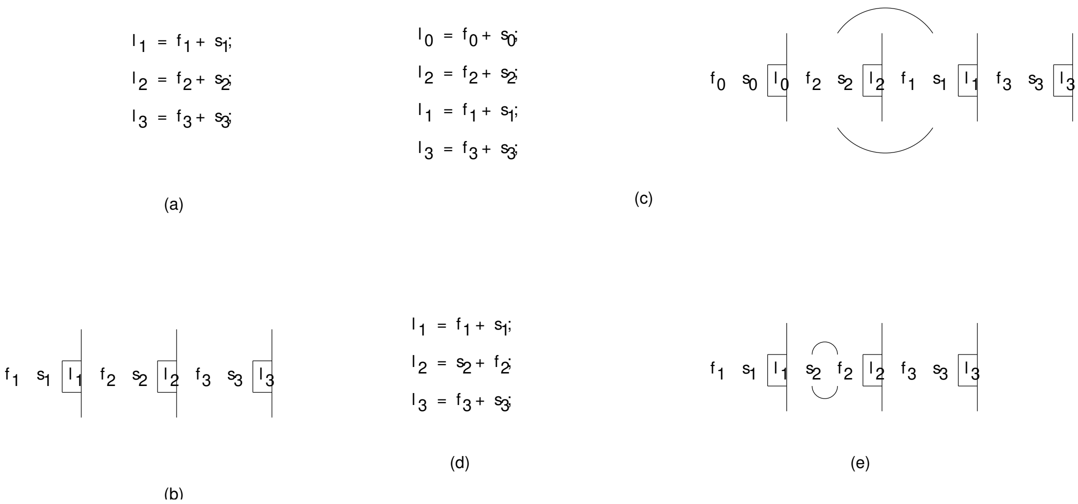


Figure 10: Commutative transformation concept.

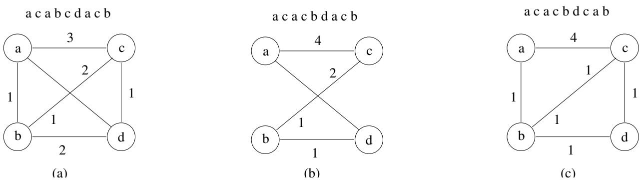


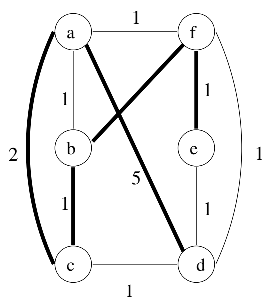
Figure 11: Incompatible edges (a, b), and (d, a).

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

(a)

a b c d e f a d a d a c a d f b

(b)



(c)

Figure 13: Example code and the associated access graph.

```
1  // INPUT : Access Sequence AS
2  // OUTPUT : Optimized Access Sequence AS°
3  Procedure Commutative-Transformation-SOA(AS)
4  (G = (V, E)) ← AccessGraph(AS)
5  AS° ← AS
6  S ← the set of all edges with weight one
7  E¹_sort ← φ
8  for (each edge (u, v) ∈ S) do
9     Compute_Primary_Benefit((u, v))
10    if (Primary Benefit (u, v) is positive)
11       Compute_Secondary_Benefit((u, v))
12       Add (u, v) to E¹_sort
13    endif
14 enddo
15 // In the next sorting step, the Secondary Benefit is used to break ties
16 Sort the entries in E¹_sort in descending order of Primary Benefit
17 // T holds the set of edges of weight one that are chosen
18 T ← φ
19 while (E¹_sort ≠ φ)
20    // Extract_Edges extracts (removes) the set of all edges with the
21    // highest primary benefit and the same largest secondary benefit.
22    // Let H = {e0, e1, ..., er} be these edges
23    H ← Extract_Edges(E¹_sort)
24    C ← the maximum compatible subset of H ∪ T
25    Perform_Transformation(AS°, C)
26    T ← T ∪ C
27 endwhile
28 return(AS°)
```

Figure 12: Commutative-Transformation-SOA.

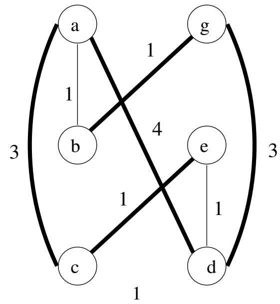
(a, b), (b, c), (c, d), (e, f), (d, e), (b, f) and (d, f). Table 2 summarizes the different computation of the benefits. The first column lists the edges and the second column shows the transition which would need to occur in the access sequence in Figure 13(a). The third column shows the number of non-zero edges turning zero and the fourth column gives the number of zero edges turning non-zero. The fifth and the sixth columns give the increase in the weight of the non zero edge and the decrease in the weight of the non zero edge, respectively. Also shown are the increase and decrease in the weight of the self edge. The second last column (P.B.) show the primary benefit, i.e., difference between column three and four. The last column shows the secondary benefit which is the sum of column five and seven. The function *Assign-Benefit* formulates Table 2 and assigns the Primary and Secondary Benefit value. The *Assign-Compatibility* function call checks for the compatibility of each weight 1 edge with other weight 1 edge. As all the edges listed, except the last two ones, have a positive primary benefit, they are stored in $\mathsf{E}^1_{\mathsf{sort}}$ in the order same as in the table. Only the last two entries will not be in $\mathsf{E}^1_{\mathsf{sort}}$. Here the primary benefit is all 1, and the secondary benefit for edge(a, f) is higher than the rest. The Extract_Edges function call will return all the edges in the table as they are all compatible with each other. The following three transformation will be performed in line 18: (1) $fada \rightarrow fdaa$; (2) $abcd \rightarrow bacd$; and (3) $cdef \rightarrow cedf$. This transformation will result in the access sequence shown in Figure 14(b). This access sequence AS° will be returned in line 21. The Liao cost of the offset assignment obtained now has fallen from 5 to 2 as shown in

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

(a)

b a c e d f d a a d a c a d f b

(b)

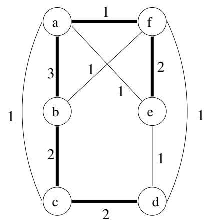


(c)

Figure 14: Optimized code, the associated access graph, and the offset sequence.

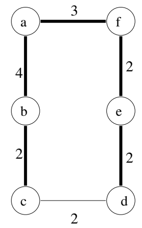
c = a + b;
f = d + e;
a = b + a;
d = e + f;
b = c;
f = a;

(a)



a b c d e f b a a e f d c b a f

(b)



a b c d e f a b a f e d c b a f

(c)

Figure 15: Optimal solution of the SOA.

Figure 13(c) and Figure 14(c).

Let us concentrate now on the example shown in Figure 15(b). We follow the same procedure as explained above. The primary and secondary benefit values that will be generated are shown in Table 3. The transformations that would be performed are: `(1)` $aefd \rightarrow afed$, and `(2)` $fbaa \rightarrow faba$. The input access sequence is shown in Figure 15(b). and the output access sequence is shown in Figure 15(c). The Liao's cost for this example has fallen from 4 to 2. In the case of edge(e, d) the primary benefit is -1, so it would not be included in $\mathsf{E}^1_{\mathsf{sort}}$ because of the **if** statement in line 9. Experimental results and additional details have been omitted for lack of space; see [2] for this.

6 Iterative-l-SOA

So far, we looked at the problem of storage assignment when the value of the post increment permitted with indirect addressing modes in embedded processors was 1. However, many recent DSP processors permit increments and decrements of values greater than one. For example, the Texas Instruments TMS320C80 allows for increments in the range of -7 to +7 and has eight address registers. Sudarsanam *et al.* [15] have formulated a method for a solution to the problem of considering the offset range which depends on discovering cliques in the access graph. They provide this method through a parameterizable optimization algorithm that operates on a procedure-wide basis [15]. We provide a brief outline of our solution to problem of address offset assignment for the case of a single address register and an enlarged increment range of an arbitrarily value of l, $i.e.$ $(-l, ..., -1, 0, 1, ..., +l)$.

We propose a heuristic that is computationally less expensive than Sudarsanam *et al.*'s solution. We formulate the offset assignment just like in $Solve\text{-}SOA$ and ask the question as to how we can layout the different disjoint paths in the access graph to minimize the cost. This is because we have to pay a cost regardless of the disjoint path layout when $l = 1$. It is possible to reduce the cost of these inter-disjoint path transitions because if we are allowed to move with a distance of $l > 1$, then some of these inter-disjoint path edges have an effective zero weight. This heuristic reflects an iterative method where we start with $l = 1$ and with each iteration increment l by one. The number of disjoint paths reduces progressively. In the end, we have a single cover for the vertices of the access graph with possibly some zero weight edges. Due to lack of space, we omit the detailed comparison of our approach with that of Sudarsanam *et al.*'s and refer the interested reader to [1].

7 Conclusions

Optimal code generation is important for embedded systems in view of the limited area available for ROM and RAM. Small reductions in code size could lead to significant changes in chip area and hence reduction in cost. We looked at the Simple Offset Assignment (SOA) problem and proposed a heuristic which, if given as input, the offset assignment from Liao or some other algorithm will attempt to improve on that. It does this by trying to include the highest weighted edges not included in cover of an access graph. The proposed heuristic is quite simple and intuitive. Unlike algorithms that are used in different computer applications, it is possible to justify a higher running time for an algorithm designed for a compiler (especially for embedded systems), as it is run once to produce the code, which is repeatedly executed. In the case of embedded systems, there is the added benefit of savings in ROM area, possibly reducing the cost of the chip.

In addition, we have addressed two important issues: the first one is the use of commutative transformations to change the access sequence and thereby reducing the code size; the second deals with exploiting those cases where the post-increment or decrement value is allowed to be greater than one. We are currently exploring several issues. First, we are looking at the effect of statement reordering on code density. Second, we are evaluating the effect of variable life times and static single assignment on code density. In addition, reducing code density for programs with array accesses is an important problem.

Acknowledgments

The work of J. Ramanujam is supported in part by an NSF grant CCR-0073800 and by NSF Young Investigator Award CCR–9457768. Mahmut Kandemir is supported in part by NSF grant CCR-0093082 and by the Pittsburgh Digital Greenhouse.

References

[1] S. Atri. *Improved Code Optimization Techniques for Embedded Processors.* M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1999.

[2] S. Atri, J. Ramanujam, and M. Kandemir. *The effect of transformations on offset assignment for embedded processors.* Technial Report, Louisiana State University.

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

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivist. *Introduction to Algorithms.* MIT Press, 1990.

Table 2: Primary and secondary benefit measures for the example in Figure 13(b).

Edge	Trans.	NZ $\rightarrow$ 0	0 $\rightarrow$ NZ	NZ $\uparrow$	NZ $\downarrow$	self $\uparrow$	self $\downarrow$	P.B.	S.B.
(a, f)	$fada \rightarrow fdaa$	1 (a, f)	0	1 (f, d)	0	1 (a, a)	0	1	2
(a, b)	$abcd \rightarrow bacd$	1 (b, c)	0	1 (a, c)	0	0	0	1	1
(b, c)	$abcd \rightarrow bacd$	1 (b, c)	0	1 (a, c)	0	0	0	1	1
(c, d)	$cdef \rightarrow cedf$	2 (c, d) (e, f)	1 (c, e)	0	0	0	0	1	0
(e, f)	$cdef \rightarrow cedf$	2 (c, d) (e, f)	1 (c, e)	0	0	0	0	1	0
(d, e)	$cdef \rightarrow cedf$	2 (c, d) (e, f)	1 (c, e)	0	0	0	0	1	0
(b, f)	*fixed*								
(d, f)	*fixed*								

Table 3: Primary and secondary benefit measures for the example in Figure 15(b).

Edge	Trans.	NZ $\rightarrow$ 0	0 $\rightarrow$ NZ	NZ $\uparrow$	NZ $\downarrow$	self $\uparrow$	self $\downarrow$	P.B.	S.B.
(a, e)	$aefd \rightarrow afed$	2 (a, e) (f, d)	0	2 (a, f) (e, d)	0	0	0	2	0
(f, d)	$aefd \rightarrow afed$	2 (a, e) (f, d)	0	2 (a, f) (e, d)	0	0	0	2	0
(b, f)	$fbaa \rightarrow faba$	1 (b, f)	0	1 (a, b)	0	0	1	1	1
(e, d)	$cdef \rightarrow cedf$	0	1 (c, e)	2 (e, d) (d, f)	1	0	0	-1	2
(f, d)	*fixed*								

[5] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. International Conference on Computer Aided Design,* pages 109–112, Nov. 1996.

[6] S. Y. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. Thesis. MIT, June 1996.

[7] S. Y. Liao, S. Devadas, K. Keutzer and S. Tjiang, and A. Wang. Storage Assignment to Decrease code Size Optimization. In *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation.* pages 186-195, June 1995.

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Zivojnovic and H. Meyr. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In *Hardware/Software Co-Design*, Kluwer Acad. Pub., G. De Micheli and M. Sami, Editors, 1995.

[9] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*, Kluwer Acad. Pub., 1995.

[10] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill 1994.

[11] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[12] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[13] P. G. Paulin, M. Cornero, C. Liem *et al. Trends in Embedded System Technology, an Industrial Perspective.* Hardware/Software Co-Design, M. Giovanni M. Sami, editors. Kluwer Acad. Pub., 1996.

[14] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation.* pages 128–138, June 1999.

[15] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and Evaluation of Address Arithmetic Capabilities if Custom DSP Architectures. In *Proceedings of 1997 ACM/IEEE Design Automation Conference.* pages 297-292, 1997.

[16] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of 1995 International Conference on Computer-Aided Design.* pages. 388-392, 1995.

[17] A. Sudarsanam, S. Malik, S. Tjiang, and S. Liao. Optimization of Embedded DSP Programs Using Post-pass Data-flow Analysis. In *Proceedings of 1997 International Conference on Acoustics, Speech, and Signal Processing.*