Improving Offset Assignment on Embedded Processors Using Transformations

Sunil Atri[1], J. Ramanujam[1]*, and Mahmut Kandemir[2]**

[1] ECE Dept., Louisiana State University, Baton Rouge LA 70803, USA
[2] CSE Dept., The Pennsylvania State University, University Park, PA 16802, USA

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 areas such as signal processing. In this paper, we address 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. These addressing modes allow for efficient sequential access but the addressing instructions increase code size. Most of the previous approaches to the problem aim to find a placement or layout of variables in the memory so that it is possible to subsume explicit address pointer manipulation instructions into other instructions as a post-increment or post-decrement operation. Our solution is aimed at transforming the access pattern by using properties of operators such as commutativity so that current algorithms for variable placement are more effective.

1 Introduction

Embedded processors (e.g., fixed-point digital signal processors, micro-controllers) are found increasingly in audio, video and communications equipment, cars, etc. thanks to the falling cost of processors [6]. These processors have limited code and data storage. Therefore, making efficient use of available memory is very important. On these processors, the program resides in the on-chip ROM; therefore, the size of the code directly impacts the required silicon area and hence the cost. Current compiler technology for these processors typically targets code speed and not code size; the generated code is inefficient as far code size is concerned. An unfortunate consequence of this is that programmers are forced to hand optimize their programs. Compiler optimizations specifically aimed at improving code size will therefore have a significant impact on programmer productivity [4,5].

DSP processors such as the TI TMS320C5 and embedded micro-controllers provide addressing modes with auto-increment and auto-decrement. This feature allows address arithmetic instructions to be part of other instructions. Thus, it eliminates the need for

* Department of Electrical and Computer Engineering. Louisiana State University. Baton Rouge, Louisiana 70803-5901. {jxr,sunil}@ee.lsu.edu

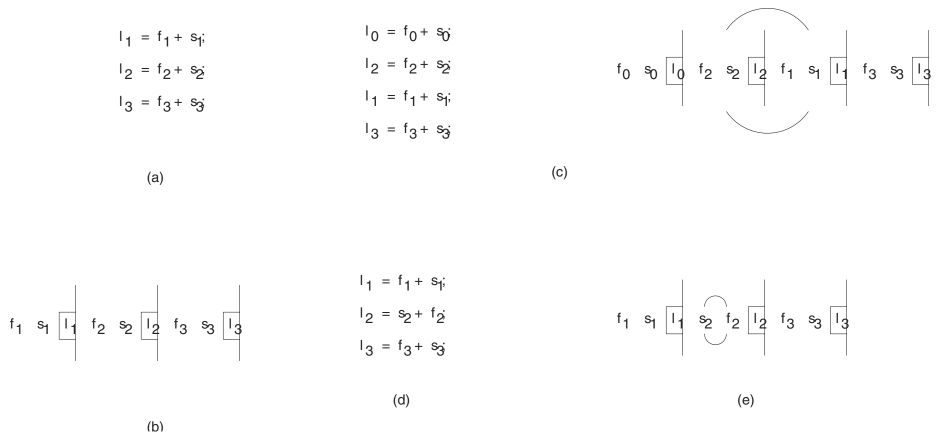
** Department of Computer Science and Engineering. Pennsylvania State University. University Park, PA 16802-6106. kandemir@cse.psu.edu explicit address arithmetic instructions wherever possible, leading to decreased code size. The memory access pattern and the placement of variables has a significant 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 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 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* whose nodes represent variables and with weighted undirected edges. The weight of of an edge (a, b) is the number of times variables a and b are adjacent in the access sequence. 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. This paper presents a technique that modifies the access pattern using algebraic properties of operators such as commutativity. The goal is to reduce the number of edges of non-zero weight in the access graph. Rao and Pande have proposed some optimizations for the access sequence based on the laws of commutativity and associativity [7]. Their algorithm is exponential. In this paper, we present an efficient polynomial time heuristic.

2 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 be 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 1(a); they have an access sequence as shown in Figure 1(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 1(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) $w(l_0, f_2)$++



```
l1 = f1 + s1;
l2 = f2 + s2;
l3 = f3 + s3;
```

(a)

```
l0 = f0 + s0;
l2 = f2 + s2;
l1 = f1 + s1;
l3 = f3 + s3;
```

(c)

f0 s0 l0 f2 s2 l2 f1 s1 l1 f3 s3 l3

f1 s1 l1 f2 s2 l2 f3 s3 l3

(b)

```
l1 = f1 + s1;
l2 = s2 + f2;
l3 = f3 + s3;
```

(d)

f1 s1 l1 s2 f2 l2 f3 s3 l3

(e)

Fig. 1. Commutative transformation concept.

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 1(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 [7,1]. Consider again the access sequence shown in Figure 1(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 1(b), we have a modification in the access sequence, which, corresponds to the commutative transformation shown in Figure 1(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 1(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.

2.1 Detailed Explanation of the Commutative Transformation Heuristic

We now discuss in detail our commutative transformation heuristic, called Commutative-Transformation-SOA(AS), given in Figure 3. 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 $AccessGraph(\mathsf{AS})$ 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 2(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°.

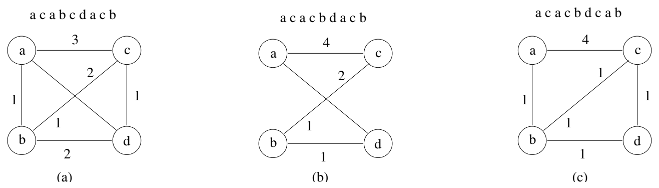


Fig. 2. Incompatible edges (a, b), and (d, a).

```
// INPUT :   Access Sequence AS
// OUTPUT :   Optimized Access Sequence AS°
Procedure Commutative-Transformation-SOA(AS)
$(G = (V, E))$ ⟵ AccessGraph(AS)
AS° ⟵ AS
$S$ ⟵ the set of all edges with weight one
$E^1_{sort}$ ⟵ $\phi$
for (each edge $(u, v) \in S$) do
   Compute_Primary_Benefit($(u, v)$)
   if (Primary Benefit $(u, v)$ is positive)
      Compute_Secondary_Benefit($(u, v)$)
      Add $(u, v)$ to $E^1_{sort}$
   endif
enddo
// In the next sorting step, the Secondary Benefit is used to break ties
Sort the entries in $E^1_{sort}$ in descending order of Primary Benefit
// $T$ holds the set of edges of weight one that are chosen
$T$ ⟵ $\phi$
while ($E^1_{sort} \neq \phi$)
   // Extract_Edges extracts (removes) the set of all edges with the
   // highest primary benefit and the same largest secondary benefit.
   // Let $H = \{e_0, e_1, ..., e_r\}$ be these edges
   $H$ ⟵ Extract_Edges($E^1_{sort}$)
   $C$ ⟵ the maximum compatible subset of $H \cup T$
   Perform_Transformation(AS°, $C$)
   $T$ ⟵ $T \cup C$
endwhile
return(AS°)
```

Fig. 3. Commutative-Transformation-SOA.

2.2 Commutative Transformation Heuristic Example

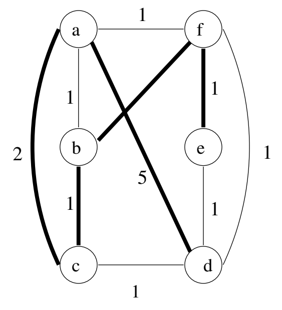
We now show the working of the heuristic through an example. Consider the access sequence Figure 4(b), with the resulting access graph in Figure 4(c). The edges of weight 1 are: (a, f), (a, b), (b, c), (c, d), (e, f), (d, e), (b, f) and (d, f). Table 1 summarizes the different computation of the benefits. The first column lists the edges and the second column shows the transition which would need to occur in the access sequence in Figure 4(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 1 and assigns the Primary and Secondary Benefit value. The *Assign-*

```
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)

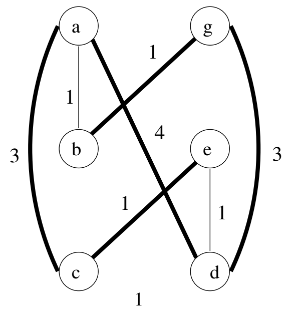
Fig. 4. Example code and the associated access graph.

```
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)

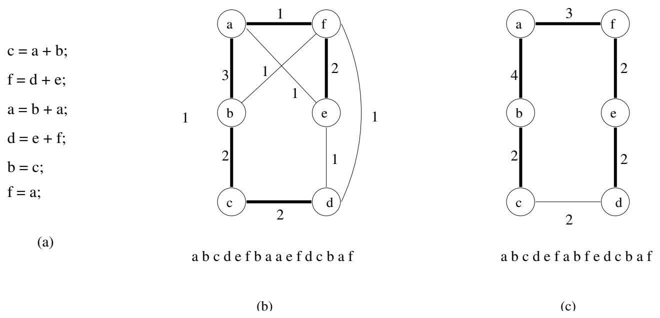


(c)

Fig. 5. Optimized code, the associated access graph, and the offset sequence.

$Compatibility$ 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 5(b). This access sequence $\mathsf{AS}°$ will be returned in line 21. The Liao cost of the offset assignment obtained now has fallen from 5 to 2 as shown in Figure 4(c) and Figure 5(c).

Let us concentrate now on the example shown in Figure 6(b). We follow the same procedure as explained above. The primary and secondary benefit values that will be generated are shown in Table 2. The transformations that would be performed are: (1) $aefd \rightarrow afed$, and (2) $fbaa \rightarrow faba$. The input access sequence is shown in Figure 6(b). and the output access sequence is shown in Figure 6(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



```
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 f e d c b a f
(c)

Fig. 6. Optimal solution of the SOA.

Table 1. Primary and secondary benefit measures for the example in Figure 4(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*								

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.

3 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. The offset assignment problem is useful in reducing code size on embedded processors. In this paper, we explored the use of commutative transformations in order to reduce the number of edges in the access graph so that the probability of finding a low cost cover is increased. We have considered commutative transformations, but it is also possible to look at the others transformations, like associative and distributive transformations and even statement reordering. Rao and Pande's solution [7] computes all the possible transformations, which is exponential. The heuristic presented in this paper identifies specific edges and selects the corresponding transformation to perform. As the number of transformation which we perform is

Table 2. Primary and secondary benefit measures for the example in Figure 6(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*								

bounded by the number of edges, this algorithm is much faster. 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 NSF Young Investigator Award CCR–9457768 and NSF grant CCR–0073800.

References

1. S. Atri. *Improved Code Optimization Techniques for Embedded Processors.* M.S. Thesis, Dept. Electrical and Computer Engineering, Louisiana State University, Dec. 1999.
2. S. Atri, J. Ramanujam, and M. Kandemir. *The effect of transformations on offset assignment for embedded processors.* Technial Report, Louisiana State University, May 1999.
3. 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.
4. S. Y. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. Thesis. MIT, June 1996.
5. 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.
6. P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*, Kluwer Acad. Pub., 1995.
7. 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.