Code Size Optimization for Embedded Processors using Commutative Transformations

Sai Pinnepalli[†], Jinpyo Hong[‡], J. Ramanujam[†], Doris L. Carver[†]
[†]*Louisiana State University,* [‡]*Korea University of Technology and Education*

Abstract

Code optimization of the offset assignment generated in embedded systems allows for power and space efficient systems. We propose a new heuristic that uses edge classification to commutatively transformation and optimize the assignment. We introduce concept of breakable and unbreakable edges, which assists in selecting edges for path cover and edges for commutative transformation.

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 [11]. 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 [1, 11]; 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 [8-10].

DSP processors such as the Texas Instruments TMS3205 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 explicit address arithmetic instructions wherever possible, leading to decreased code size. The memory access pattern and the placement of variables have 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 the processor has 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 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 [12] have proposed some optimizations for the access sequence based on the laws of operator commutativity and associativity. Their algorithm is exponential. Here, we present an efficient polynomial time heuristic.

The rest of this paper is organized as follows. Section 2 introduces the commutative transformation and Section 3 provides a motivating example. Section 4 introduces the classification of edges in the access graph into breakable and unbreakable edges, which is a key idea in this paper. Section 5 presents our heuristic and Section 6 presents a detailed example. Experimental results demonstrating the efficacy of our approach are presented in Section 7. Related work is discussed in Section 8 and Section 9 concludes with a summary and discussion of ongoing and planned work.

2. Commutative transformation

Two operands x and y are said to be commutative under an operator α if they satisfy $x \alpha y = y \alpha x$

Some instructions have commutable operations,

13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA 2007)
0-7695-2975-5/07 $25.00 © 2007 IEEE

while others do not. An instruction ADD(a, b) is equivalent to " = a + b". In this operation, a and b are commutable since a + b = b + a. Similarly MUL(a, b), which is the same as " = a * b", is commutable since a * b = b * a.

Some instructions such as SUB(a, b) and DIV(a, b) are not commutable. SUB(a, b) is equal to " = a – b" and a - b ≠ b – a, unless a = b; DIV(a,b) is equal to "= a / b" and a / b ≠ b / a unless a = b.

From the definition of a commutative operation, some instructions might appear to be commutative, but commutative operations in such instructions is not allowed due to the implementation of the operation. For example MPYA(a, b, c) is equal to t = MPY(a, b), followed by ADD(t, c) where t is an internal variable and the result of MUL(a, b) is stored temporarily in this variable. This operation may be represented as "(a * b) + c". This operation implies that (a * b) + c = c + (a * b). Algebraically, this assertion is true. But, while computing the values, the ADD operation cannot be completed before the MUL operation is complete. However MUL(a,b) within MPYA is still commutable. Such operations are implemented in a MAC.

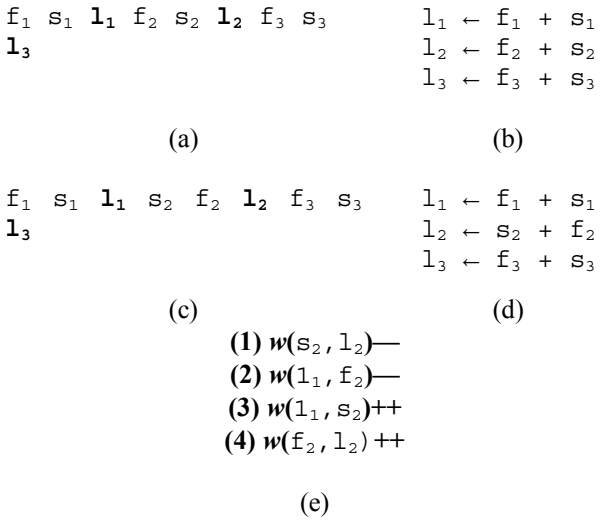

Figure 1. Commutative Transformation Concepts

Operation such as SUB(a, b) may be considered equal to ADD(-b,a) ≡ ADD(SIG(b),a). Such an operation makes ADD non atomic, and the nature of the operation also makes ADD non-commutative as SIG has to be completed before addition is performed. Similarly DIV(a,b) ≡ MUL(INV(b),a) makes MUL non-commutative. This operation is not atomic, and MUL is dependent on the result of the INV operation.

We note that commutativity is limited to atomic operations such as ADD and MUL that do not depend on internal results. In all the examples and experiments, we considered one or two operands only; but the results may be extended to any commutative transformation.

Consider the access sequence shown in Figure 1(a) and its basic block in Figure 1(b). Consider a valid commutative transformation of the second statement in Figure 1(b). This transformation results in a new set of statements in Figure 1(d). The statement $l_2 \leftarrow f_2 + s_2$ is transformed into $l_2 \leftarrow s_2 + f_2$. This changes the access sequence from '$f_1\ s_1\ \mathbf{l_1}\ f_2\ s_2\ \mathbf{l_2}\ f_3\ s_3\ \mathbf{l_3}$' to '$f_1\ s_1\ \mathbf{l_1}\ s_2\ f_2\ \mathbf{l_2}\ f_3\ s_3\ \mathbf{l_3}$'. The commutative transformation in statement 2 may be represented as change in the weights of edges $\langle s_2, l_2\rangle, \langle l_1, f_2\rangle, \langle l_1, s_2\rangle$, and $\langle f_2, l_2\rangle$. This change is represented in Figure 1(e) as increment and decrement of weights for the corresponding edges.

In some instances, the weight of an edge may go from 1 to 0, which implies that the edge being considered will not exist in the new access graph. Similarly the weight of an edge may change from 0 to 1, which implies creation of a new edge in the access graph. We exploit this feature in our heuristic to improve the cost of the assignment.

3. Motivating example

Consider the basic block in Figure 2.

```
e ← d
a ← f + e
f ← d
a ← d + e
d ← e + b
f ← b
f ← c + a
e ← d
```

Figure 2. Basic Block for Example

The corresponding access graph for this basic block is shown in Figure 3. This basic block results in a cost of 8 using Hong's heuristic [6] with "b f d e a c" as its layout. Edges not part of the layout need additional instructions, which is the cost of the layout. For this example we consider a different assignment with a cost of 10, the layout for which is "c a f d e b". Edges involved in the layout are shown in bold. We have primarily chosen edges <d,f> and <d,e> while ignoring the edge <a,e> all of which have a weight 3. All three of the transitions between a and e may be transformed using commutative transformations. a ← f + e may be transformed to a ← e+ f. This reduces the weight of edge <a,e>. A similar transformation may be made to the statement

a ← d + e. This transformation will further reduce the weight of edge <a,e>. In addition a transformation of the statement d ← e + b to d ← b + e can reduce the weight of the edge <a,e> to 0. We only perform two of these transformations that affect the edge <a,e>. The resulting basic block and its access graph are shown in the Figure 4 and Figure 5, respectively.

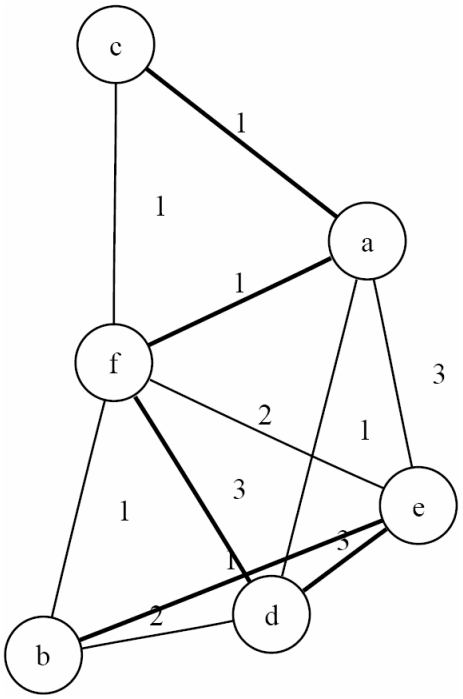

Figure 3. Access Graph for Basic Block in Figure 2

This example suggests that a mechanism for classifying edges of the access graph may be beneficial in commutatively transforming the graph and therefore the offset assignment associated with it.

```
e ← d
a ← e + f
f ← d
a ← d + e
d ← b + e
f ← b
f ← c + a
e ← d
```

Figure 4. Basic Block for Access Graph in Figure 3

4. Classification of edges

We identify the edges that can be transformed and edges that cannot be transformed. Edges that can be commutatively transformed are defined as "breakable" edges, while edges that cannot be commutatively transformed are defined as "unbreakable".

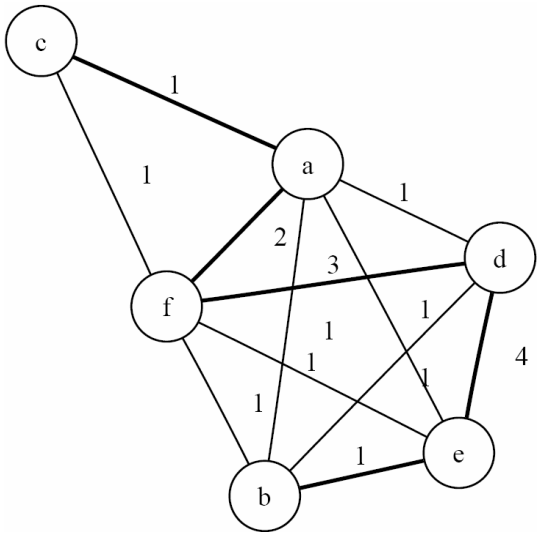

Figure 5. Access Graph for Basic Block in Figure 4

Statement "C ← A + B" generates an access sequence ABC. The statement may also be commutatively transformed as "C ← B + A", which generates an access sequence BAC. Edge AB still exists in the transformed access sequence as BA. This edge is "unbreakable". While edge BC can be eliminated in the second access sequence, such edges are "breakable".

The following cases define other "breakable" (BR) and "unbreakable" (UB) edges. If both operands in the following statement are same, the edge between the lhs (left hand side) of the current statement and node in the rhs (right hand side) of the next statement is "unbreakable". If the two operands are different, the edge is breakable.

S1: C ←
S2 Z ← X + X

The edge <C,X> is "unbreakable" and edge <Z,X> is "unbreakable" in the above code segment.

S1: C ←
S2: Z ← X + Y

The edge <C,X> is classified as "breakable", as is the edge <Z,Y>. The newly added edges after commutative transformation are <C,Y> and <Z,X>.

S1: C ←
S2: Z ← X

The edges <C,X> and <Z,X> are "unbreakable".

S1: C ←
S2: C ← A + B

The edges <C,A> and <C,B> are "unbreakable". It may be argued that this situation is a case for dead-code elimination, where statement S1 may be deleted. Statements in this form are not a complete representation of a task of the embedded system. Since the program code is executed in real-time, it is possible to have a synchronous effect due a change in the value set at an address, i.e., values set in C have an effect on the operation of the device.

We group each edge into groups of breakable and unbreakable edges. When we choose the edges, our heuristic gives preference to unbreakable edges over breakable edges.

5. Heuristic xformSOA

We propose a heuristic that uses classification of the edges in an access graph to derive an empirically optimal offset assignment. The principal idea of this heuristic is presented in the Figure 6. We find an initial assignment that gives a cost C to compare the effects of the transformation on the initial layout of the access graph (G). In practice, we used a standard SOA heuristic to compare the final result of all of the transformations.

This heuristic iterates (line 2) until there are no further transformations that reduce the cost of the previous iterations (line 5). After each set of transformations that affect the access graph, we find the cost of the new assignment C' (line 4). If C' is less than or equal to C, the cost estimated in the earlier iteration, the transformed graph G' becomes the new access graph G (line 6).

```
0:   flag ← 1
1:   C ← measure( layout(G))
2:   while ( flag == 1 ) {
3:           G' ← xform(G)
4:           C' ← measure( layout(G' )
5:           if (C' ≤ C) {
6:                   G ← G'
7:                   flag ← 1 }
8:           else
9:                   flag ← 0 }
10:  optimalSOA(G)
```

Figure 6. *xformSOA* Heuristic

This heuristic has two procedures: *xform* and *measure*. Procedure *measure*, shown in Figure 7, finds an offset assignment for a graph using the classification of edges such as breakable and unbreakable. An edge whose weight is 3 might have any combination of breakable and unbreakable edges (3BR+0UB, 2BR+1UB, 1BR+2UB, and 0BR+3UB, where BR is a breakable edge and UB is unbreakable edge). Procedure *xform*, Figure 8, transforms a given access graph (G) and its basic block to obtain a different access sequence.

```
measure(G, B) {
// G is access graph, B is the basic block defining G
   BRsort ← sorted list of breakable edges (B)
   UBsort ← sorted list of unbreakable edges(B)
// this is essentially SOA with UB
   P ← MWPC with UBsort
//add additional edges to path cover
   add edges from BRsort not yet covered
   C ← wt of uncovered edges // cost of uncovered
edges
   return(C)
}
```

Figure 7. Procedure Measure for *xformSOA* Heuristic

Procedure *measure* classifies edges in G into two categories – breakable and unbreakable. In steps 2 and 3 of the procedure, a sorted list of breakable edges (BRsort) and another of unbreakable edges (UBsort) are created. If two edges have the same weight in UBsort, then the edge with higher total edge weight is given higher priority. If two edges in BRsort have the same edge weight, then the current implementation gives higher priority to the edge with the larger weight. Other such tie-break possibilities can also be considered; this is a subject of future investigation.

From the sorted list of unbreakable edges (UBsort), a Maximum Weight Path Cover (MWPC) is generated. The process of generating this path is akin to the generation of offset assignment in other heuristics. Edges from BRsort are then considered for addition to the path. Any edges that are not part of the path are now considered to contribute to the cost of the offset assignment (C). C is the return value of this procedure.

After the path cover is obtained, edges may be left in BRsort, that can be transformed. These edges are then transformed and a measure of comparison for each of these transformations is computed as shown in step 5 in Figure 8. A transformation is accepted only if Δ_{eff} is non-negative. (Transformations may also be limited to Δ_{eff}, as a variation)

A measure of this transformed access graph (G'') is obtain using procedure *measure*(G''). If this cost is lower than the cost computed in the earlier iteration, the cycle of procedure *xform* and procedure *measure* are repeated. Once *xformSOA* stops improving the cost of the access graph, the cost of the access graph is computed using any benchmark heuristics, labeled

optimalSOA, which obtain assignments using only the edge weights and not their classifications.

```
0:    xform(G) {
1:    G' ← G
2:      for each edge e that is uncovered {
3:        for each breakable instance j of edge e {
4:          G'' ← xform of edge instance (G')
5:          Δeff ← (
6:            # of covered / self edges whose weight ↑
7:            + # of uncovered edges whose weight ↓
8:            - # of uncovered edges whose weight ↑ )
9:            if Δeff ≥ 0
10:             G' ← G''}}}
```

Figure 8. Procedure xform for *xformSOA* Heusristic

6. Detailed example

Consider the following basic block used in Atri et al. [2-4]. This basic block shown in Figure 9 yields the access graph shown in the Figure 10. The edges of this access sequence are classified into breakable and unbreakable edges as shown in the Figure 11. The basic block is converted into access sequence '`a b c d e f b a a e f d c b a f`' by xformSOA heuristic.

```
c ← a + b
f ← d + e
a ← b + a
d ← e + f
b ← c
f ← a
```

Figure 9. Basic Block from Atri et al. [2-4]

In procedure *measure* for the SOA, the edges are first computed as shown in Figure 11(a). These edges are classified into UBsort (unbreakable) and BRsort (breakable) edges. For example, there are two instances of edge <c,d>, one between statements "c ← a + b" and "f ← d + e" and second between statements "d ← e + f" and "b ← c". The first edge can be eliminated by commuting the statement "f ← d + e" into "f ← e + d". However, the second edge cannot be commuted. Hence edge <c,d>, whose weight is two, is classified both in breakable edges and unbreakable edges. i.e., edge <c,d> cannot be completely eliminated. It can at most be reduced to an edge of weight 1.

Using the classification, *measure* derives an assignment as highlighted in the access graph shown in Figure 12. The *xform* procedure then commutes edges in BRsort that do not negatively affect the cost of the assignment. From the graph, edges <a,e>, <d,f>, <a,f>, and <b,f> are the edges not included in the cover. It is desirable that these edges be commuted so that the cost of edges not covered by the MWPC is reduced, if not fully eliminated. Of the four edges, <a,e>, <d,f>, and <b,f> are classified as breakable edges. Breaking the edge <a, e> requires commuting "d ← e + f" to "d ← f + e" resulting in the elimination of edge <d, f>. The result of this transaction is w<a,e>--, w<d,f>--, w<a,f>++, and w<d,e>++. The net result is the elimination of two breakable edges not part of the path cover (<a,e> and <d,f>), and increase in the weight of an edge that is not part of the path cover (<a,f>) and one that is part of the path cover (<d,e>). This commutative transformation affects the net weight by "-1". The edge <b,f> may also be broken by transforming "a ← b + a" into "a ← a + b". This transformation does not affect the cost but changes the access graph. After the first iteration of transformations, the basic block with transformations that amount to "-1" is shown in the Figure 13.

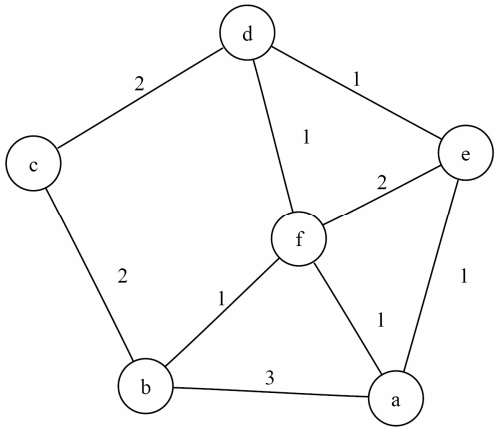

Figure 10. Access Graph for the Basic Block in Figure 9

The access graph and measure is computed for the new basic block. The access graph after the transformations is shown in Figure 14.

It is evident from the access graph that the cost of the new assignment is 2. It is possible to further reduce the cost if <d,e> is breakable and its transformation only decreases the cost. This assignment is feasible only by reversing earlier transformation of "d ← e + f" to "d ← f + e". We stop the transformations here with an optimal cost of two. This access sequence is then presented to a benchmark SOA. The SOA algorithm used in our heuristic is Hong et al.'s SOA algorithm. The cost returned for this assignment is also 2. We consider these transformations empirically optimal.

Edge weights	Unbreakable Edges	Breakable edges
b-f: 1		
a-b: 3	a-b: 3	c-d: 1
d-e: 1	c-d: 1	e-f: 1
a-e: 1	b-c: 1	b-c: 1
a-f: 1	e-f: 1	b-f: 1
c-d: 2	d-e: 1	d-f: 1
b-c: 2	a-f: 1	a-e: 1
e-f: 2		
d-f: 1		
(a)	(b)	(c)

Figure 11. Edge Classification for the Basic Block in Figure 9

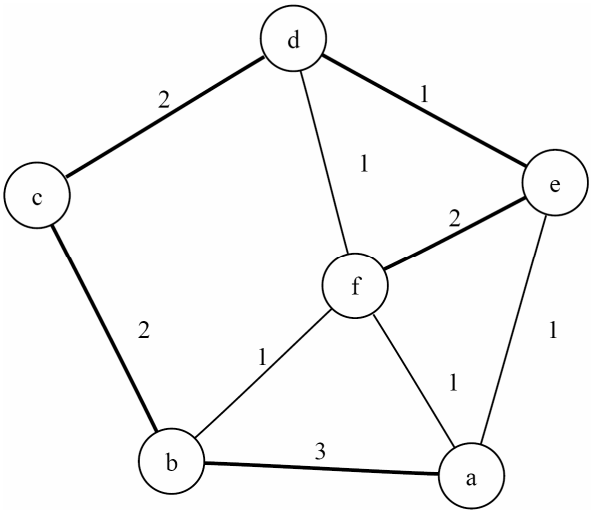

Figure 12. Measure of Access Graph in Figure 10

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

Figure 13. Basic block after transformations

The other basic block (Figure 15) in the motivating example also commutes to an optimal solution in three stages as shown in the Figure 17. The final transformation resulting from the *xformSOA* heuristic, shown in Figure 16, yields an optimal cost of 2.

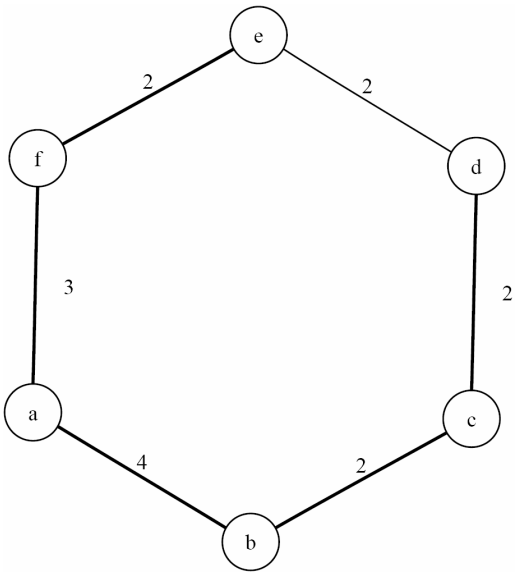

Figure 14. Access Graph for Basic Block in Figure 13

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

Figure 15. Basic block from Atri's Motivating Example

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

Figure 16. Transformed Basic Block of Figure 15

7. Experimental results

The *xformSOA* heuristic was tested with random sequences of varying lengths |S| and number of variables |V|. It is assumed that 80 % of the statements are of the form x ← y + z (two operands in the rhs), and 20 % of the statements are of the form x ← y (one operand in rhs). Each test was repeated 1000 times before generalizing the result. The results of these tests are tabulated in Table 1.

The benefit is compared in an SOA heuristic not part of *xformSOA*. We use Hong's SOA heuristic to check initial and final costs. We observe that at least 60 % of the time there could be benefits in commutatively

transforming statements. In some instances the benefits were observed in more than 90% of access sequences. The change in cost is as high as 12 with some access sequences

Table 1. Results of an Implementation of xformSOA

|S|	|V|	% affected	Max. Change
25	6	63.8	5
50	9	81.2	10
50	20	88.9	10
100	20	88.2	12
100	60	78.4	7
100	80	76.6	6
1000	300	90.4	10

Table 2. Effect of transformations on some benchmarks

Code	Cost before	Cost after
GauHer	15	15
GauLeg	21	19
qGauss	9	6
chenDCT	95	87
chenDCT[1]	95	89
chenIDCT	124	124
chenIDCT[1]	124	116
leeDCT	92	88
leeDCT[1]	92	89
leeIDCT	121	116
leeIDCT[1]	121	115
Complex Multiplication	6	3

All random sequences were assumed to be commutative. The benchmark programs were allowed to commute instances of type x – y as (-y) + x. The benchmarks were also tested while restricting such commutation. With this heuristic, a restricted transformation designates some edges earlier considered as breakable as unbreakable, which alters the selection of edges for commutative transformation. In some instances restricting transformation resulted in lower costs.

This implementation was tested on motivating examples used in other research. The xformSOA heuristic produced the optimal value. In addition the heuristic was also tested on some benchmark algorithms shown in Table 2. Commutative transformations demonstrate benefits in most cases. Complex Multiplication lists largest possible benefit.

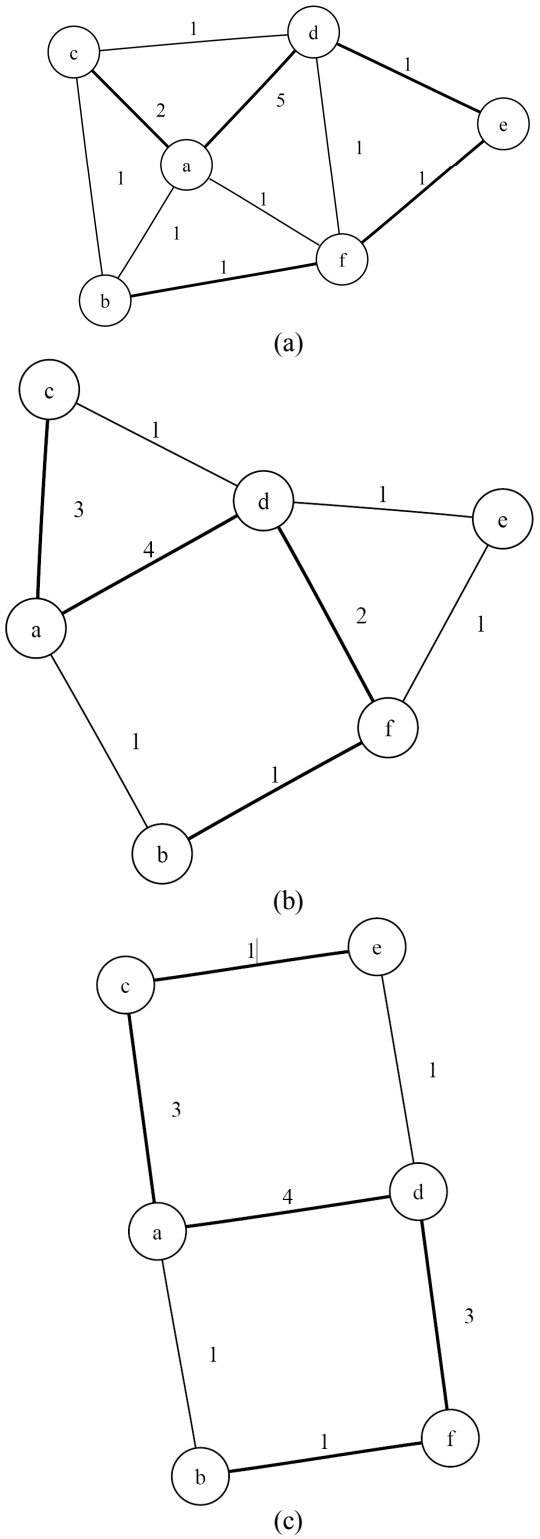

Figure 17. Transformation of Basic Block from Figure 16

[1] SUB was not allowed to commute

8. Related work

Offset assignment problem has addressed by addressed by many heuristics including Hong [6] and Leupers [11]. These heuristics attempt to provide an optimal solution to a given access sequence. Further, some heuristics attempt to modify a given assignment or initial access sequence to obtain a better assignment

Atri et al. [2-4], have shown two different ways of solving the offset assignment problem. They present a heuristic that incrementally checks for the best possible location for edges that are not part of the path cover. The edges that are not part of the assignment are sorted in the descending order of their weight. Each of these edges is found an appropriate location in the assignment. This heuristic has been found to be quite effective in Leupers' comparison of SOA heuristics [7]. This approach may be used in conjunction with any heuristic.

Atri et al., Rao and Pande [12] approach the SOA problem by first performing commutative transformation. Atri et al. look for edges of weight one that may be commutatively transformed to reduce the number of edges in access graph. They propose metrics that quantify each transformation. Transformations that have benefits are considered, while transformations that increase the cost of an assignment are ignored. Rao and Pande find all possible legal combinations of a basic block and its transformations. SOA is performed on each of these transformations. This approach is exhaustive.

9. Conclusion

Optimal offset assignment is desired in embedded systems for the code to function efficiently. In this paper we propose a heuristic that uses knowledge of basic block, not just access sequence, and performs commutative transformations to generate a more efficient code. The heuristic uses knowledge of degree of nodes, average weight, and average edges that may or may not be transformed to determine path cover. We introduce a host of tie-breakers that assist in offset assignment.

The problem of finding an optimal solution is NP-complete. The heuristic was attempted on some randomly generated sequences. The heuristic needs to be applied to a larger set of examples to obtain reliable statistics. Also, further inspection of test cases might reveal other tie-breakers that could provide a more efficient solution.

Adaptation of this heuristic to use Modify Register and for the case of multiple address registers (GOA) is being explored at this time.

10. Acknowledgments

We acknowledge the support of the National Science Foundation through grants 0073800, 0103933, 0121706, 0508245, 0509442, 0541409, the Department of Electrical and Computer Engineering at Louisiana State University, and the Center for Computation and Technology at Louisiana State University. ATT's graphing tool neato was used extensively to generate the graphs.

11. References

[1] A. Aho, R. Sethi, and J. Ullman. Compilers, Principles, Techniques and Tools. Addison Wesley, Boston 1988.

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

[3] S. Atri, J. Ramanujam, and M. Kandemir. Improving offset assignment on embedded processors using transformations. In Proc. High Performance Computing-HiPC 2000, pp. 367-374, December 2000.

[4] S. Atri, J. Ramanujam, and M. Kandemir. Improving variable placement for embedded processors. In Languages and Compilers for Parallel Computing, (S. Midkiff et al. Eds.), Lecture Notes in Computer Science, vol. 2017, pp. 158-172, Springer-Verlag, 2001.

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

[6] J. Hong. Memory Optimization Techniques for Embedded Systems, PhD Thesis etd-0712102-103747, Dept. of Electrical and Computer Engineering, Louisiana State University, July 2002.

[7] R Leupers. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. Compiler Construction, 12th International Conference, pages 290-302, CC 2003

[8] S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors. PhD thesis, MIT Department of EECS, January 1996.

[9] 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.

[10] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. ACM Transactions on Programming Languages and Systems, 18(3):235-253, May 1996.

[11] P. Marwedel, G. Goossens (eds.): Code Generation for Embedded Processors, Kluwer Academic Publishers, 1995

[12] Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. SIGPLAN '99, Atlanta, GA, USA, pages 128-138, May 1999