Procedural Level Address Offset Assignment of DSP Applications with Loops

Youtao Zhang
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083

Jun Yang
Department of Computer Science
The University of California at Riverside
Riverside, CA 92521

Abstract

Automatic optimization of address offset assignment for DSP applications, which reduces the number of address arithmetic instructions to meet the tight memory size restrictions and performance requirements, received a lot of attention in recent years. However, most of current research focuses at the basic block level and does not distinguish different program structures, especially loops. Moreover, the effectiveness of modify register (MR) is not fully exploited since it is used only in the post optimization step.

In this paper, a novel address offset assignment approach is proposed at the procedural level. The MR is effectively used in the address assignment for loop structures. By taking advantage of MR, variables accessed in sequence within a loop are assigned to memory words of equal distances. Both static and dynamic addressing instruction counts are greatly reduced. For DSPSTONE benchmarks and on average, 9.9%, 17.1% and 21.8% improvements are achieved over address offset assignment [4] together with MR optimization when there is 1, 2 and 4 address registers respectively.

1 Introduction

DSP processors, e.g. TI TMS320C2x/5x [8], AT&T DSP 16xx [10] usually provide dedicated *address generation units* (AGUs) for address calculation. The most important addressing mode, *register-indirect addressing*, accesses a memory location whose address is already in an address register (AR) and could do auto modification of the AR after its use. Two types of post-access AR modifications are commonly used: one is autoincrement and autodecrement, the other is to modify the AR using the value stored in an additional modify register (MR). Both types update AR automatically after its use by an instruction and the updates are done in parallel with the execution of the instruction

itself. If the address of the next variable could be automatically calculated from a preceding instruction, the variable would be accessed directly. Otherwise, an explicit address arithmetic instruction has to be inserted in the object code to calculate and store the correct address into the AR. By carefully arranging the variables in the memory, DSP applications can be compiled with less addressing instructions, achieving compacted object code size and improved program performance.

Recently a lot of research has been done to optimize the address offset assignment of variables to minimize the total number of address arithmetic instructions. Liao *et. al.* [1] was the first to formulate it as a Maximum Weighted Path Covering (MWPC) problem. Since the problem is NP-hard, they proposed a greedy heuristic algorithm to solve it efficiently. Both simple offset assignment (SOA) which deals with a single AR and general offset assignment (GOA) which deals with multiple ARs are discussed in that paper. Leupers *et. al.* [4] extended their work by proposing a *tie-breaking* approach in building the access graph and a heuristic algorithm for assigning data items into different registers for GOA. They also proposed an algorithm for efficient use of MRs. Sudarsanam *et. al.* [6] gave the algorithm provided the hardware supports auto post-increments range of $[-L, +L]$. Leupers *et. al.* [5] presented a genetic optimization technique that can simultaneously handle arbitrary register file sizes and auto-increment ranges. Rao and Pande [3], Choi and Kim [2] discover operations that have commutative operands and schedule the variable access sequence, i.e., reorder the operands as well as the operations themselves, to achieve minimization. Udayanarayanan and Chakrabarti [7] proposed several heuristic algorithms for MR optimization and address assignment.

In general, most address assignment algorithms take a two step approach (as shown in Figure 1): in the first step, address offset assignment scheme is generated by a heuristic algorithm; in the second step, MR optimization is car-

ried to minimize the number of address jumps introduced to the code. Each address jump or MR assignment eventually translates to an extra addressing instruction in the code. The problem with the above two-step approach is that MR is used only to amend problems that can not be solved from the first step. The MR's power is neglected in the heuristic algorithm itself. The approach proposed in this paper fully utilizes both the MR and AR to reduce the number of addressing instructions (AIs).
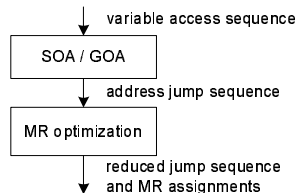

Figure 1. Generic Address Offset Assignment Approach with Modify Registers.

[18] and [5] proposed approaches to incorporate MR in offset assignment step. Hardware support is required in [18] for an auto post-increments range of $[-L, +L]$. When $L = 2$, it can be done by presetting $MR = 2$ before assignment. Our approach is more general: (1) MR can only be set to 2 and can not be modified in [18] while MR is adjustable in our approach; (2) [18] didn't consider loop structures which provide the optimization opportunities studied in this paper. In [5], an approach is proposed to generalize fitness function to incorporate MR. Its effectiveness across loop structure is not known.

Array variables also appear frequently in loop structures and they are studied in [14, 17, 16, 15]. These work focused on the computation of array indices with the loop. In contrast, the address offset assignment problem in this paper comes from the address accesses within and outside of loops. As a result, our proposed technique is orthogonal to existing address offset assignment algorithms for array and pointer variables. Our approach could be adapted to help the allocation of multiple array variables.

When extending the address offset assignment problem from basic block to procedural level, we need to distinguish static and dynamic cost. Since one addressing instruction may be executed multiple times and have multiple instances if it is in a loop, address offset assignment can reduce *either* static AI counts *or* dynamic AI counts but it is very hard to reduce both. This is because the memory layout generated in favor of one type of AI counts is usually too rigid and restricts reducing the other. Our scheme addresses this problem by providing a balanced memory layout which aims at reducing *both* static and dynamic AI counts effectively.

The rest of the paper is organized as follows. We use an example to illustrate our motivation in section 2. The al-

gorithms for SOA and GOA are discussed in section 3 and 4 respectively. Section 5 presents our experimental results with comparisons to existing algorithms. Section 6 concludes the paper.

2 Motivation

Most existing address assignment algorithms construct an *access sequence* and an *access graph* at the level of basic blocks [1, 2, 3, 4]. The access graph is used to generate variable memory layout. While these basic blocks are picked up from core functions, each of them covers only a small portion of memory. In order to generate a complete memory layout, we need to generate a procedural or program level access sequence and accordingly its access graph. Depending on how to weight each basic block, there are two approaches to combine the basic block level access sequences [6]. For the purpose of optimize static AI counts (reducing code size), each basic block is weighted equally; for the purpose of optimizing dynamic AI counts (improving performance), each basic block is weighted with its profiled execution frequency.

As we mentioned in the introduction, two problems are not fully exploited in current approaches:

- They do not distinguish different program structures. An access sequence $...(...ab)^3...$ and $...ab...ab...ab...$ are very different even though the edge between a and b is the same in both access graphs and thus a and b are allocated similarly in both cases. The former appears in a loop that iterates 3 times and the later may be scattered in the code randomly. Suppose a and b are allocated with a distance d. A single assignment $MR = d$ is likely to suffice for the loop but not enough for the second sequence.

- The effectiveness of MR is not fully exploited since it is used after offset assignment which did not have MR in mind. Let us consider a procedure with loops. To reduce static AIs, all basic blocks are weighted equally. Address jumps with different jump values (distances) may be inserted into a loop body. Although at runtime, these jumps appear repeatedly and closely several times, with one MR, we can only reduce jumps with one distance, the rest will still be there. To reduce dynamic AIs, loops are profiled with higher execution weights. Related variables in the loops are allocated consecutively. Address jumps are inserted to the code outside the loop, which is even harder to optimize by MR.

Figure 2 demonstrates the idea discussed above. Fig.2(a) is a small piece of code in which the loop is executed 3 times. If each basic block is weighted equally, the partial access sequence and its corresponding access graph
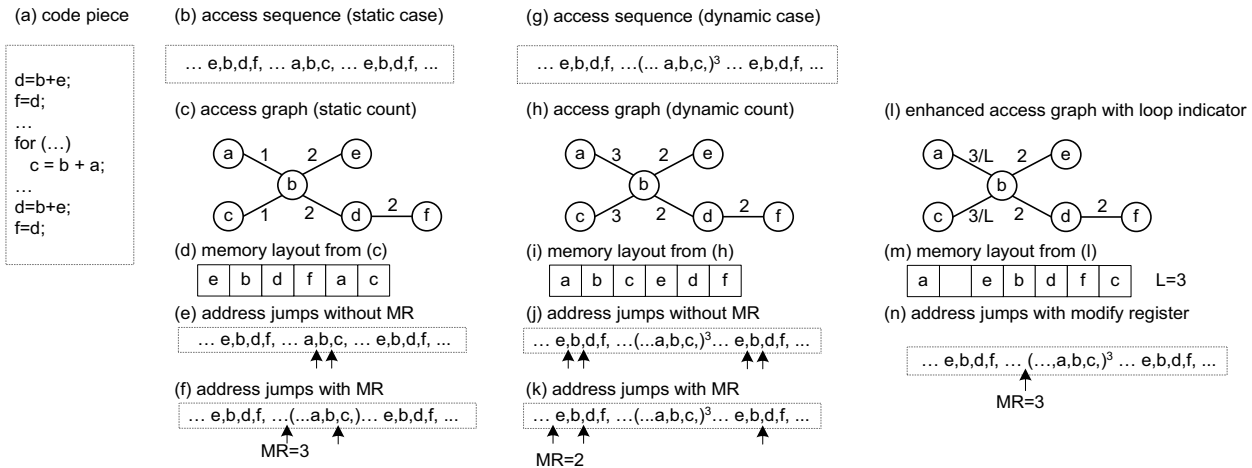
(a) code piece

```
d=b+e;
f=d;
…
for (…)
   c = b + a;
   …
d=b+e;
f=d;
```

(b) access sequence (static case)

… e,b,d,f, … a,b,c, … e,b,d,f, …

(g) access sequence (dynamic case)

… e,b,d,f, …(… a,b,c,)³ … e,b,d,f, …

(c) access graph (static count)

a —1— b —2— e
c —1— b —2— d —2— f

(h) access graph (dynamic count)

a —3— b —2— e
c —3— b —2— d —2— f

(l) enhanced access graph with loop indicator

a —3/L— b —2— e
c —3/L— b —2— d —2— f

(d) memory layout from (c)

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

(i) memory layout from (h)

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

(m) memory layout from (l)

| a | | e | b | d | f | c | L=3

(e) address jumps without MR

… e,b,d,f, … a,b,c, … e,b,d,f, …

(j) address jumps without MR

… e,b,d,f, …(…a,b,c,)³… e,b,d,f, …

(n) address jumps with modify register

… e,b,d,f, … (…,a,b,c,)³ … e,b,d,f, …

MR=3

(f) address jumps with MR

… e,b,d,f, …(…a,b,c,)… e,b,d,f, …

MR=3

(k) address jumps with MR

… e,b,d,f, …(…a,b,c,)³ … e,b,d,f, …

MR=2

Figure 2. Motivation Example.

are shown in Fig. 2(b) and (c) respectively. By applying traditional address assignment algorithms [1, 4], we can get memory layout like Fig. 2(d). This layout generates 2 static addressing instructions (Fig.2(e)) and 6 dynamic addressing instructions since they are in the loop body. Fig.2(f) shows the optimization results with one MR. It generates 2 static instructions and 4 dynamic instructions.

Similarly, if using the execution profile, edge "ab" and "bc" are given more priority, the access sequence, access graph and memory layout are shown in Fig. 2(g,h,i) respectively. It generates 4 static addressing instructions in the code. Since they are out of the loop, the dynamic instruction count is also 4. With the help of the modify register, the optimized results (Fig.2(k)) have 3 static and 3 dynamic instruction counts.

The power of MR could be exploited for the loops *while* deciding the memory layout, not after. We enhance the access graph with loop indicators and allocate related nodes with equal distance in the memory. As shown in Fig.2(m), a better memory layout can be found when the distance is 3. When we separate a, b, and c 3 words apart, e, d, and f can be easily filled in without additional AIs among them. Both static and dynamic AI counts are now reduced to one.

A possible drawback of our approach is the extra memory it requires for allocation. In the example, we need one extra memory word. However, as shown in our experimental results, the extra memory is small. Moreover, the algorithm achieves good results even with very restricted extra memory.

3 Simple Address Offset Assignment with Loops (SOA/Loop)

In this section, we discuss how to generate a procedural address offset assignment with one AR and one MR. The

procedure might have several loops but different loops are independent from each other.

An overview of the algorithm is presented in section 3.1. In section 3.2, we introduce the distance vector and discuss how to use it in generating a procedural level access graph in section 3.3. Validation of a set of selected edges from the access graph is discussed in section 3.4. A heuristic algorithm is proposed in 3.5 for efficient implementation of the algorithm.

3.1 Algorithm Overview

Figure 3 presents the overview of the algorithm. The basic idea is to distinguish access edges generated from each individual loop and select them differently. The inputs are the control flow graph (CFG), the access sequences and execution profiles for all basic blocks in the CFG. The first two steps generate the procedural access graph and sort all edges by their weights in the graph. From step 8 to 12, we search the sorted edge list and greedily mark the edges with higher weights. The validation algorithm at step 10 makes sure all marked edges can generate a valid address assignment. We also apply modify register optimization in querying the cost in step 13. The optimal address offset assignment is found by exploring all possible combinations of distance values for each loop (step 5). Steps 1, 5 and 10 are discussed in more detail in following subsections.

3.2 Distance Vector

To take advantage of the loop information during address offset assignment, we need to distinguish loops from each other and the rest of the code. This is done by assigning a unique identifier – loop distance ID L_i, for each loop i in the procedural CFG. L_i can only take positive integer values. Intuitively, by pre-setting MR to L_i outside of the loop i, a consecutive address access inside the loop with a distance of L_i will not generate extra address arith-

```
Algorithm SOA/Loop:
Input:
        CFG: Control flow graph of the procedure;
        AS: Access sequences of all basic block B_i;
        NP: Node profiles for all basic block B_i;
Output:
        Mem: Memory Layout Scheme

(1)    G = GetProceduralAccessGraph(CFG, AS, NP). /*
       section 3.3 */
(2)    SortE = {sorted edge list of all edges in graph G}.
(3)    MinCost=MAX;
(4)    MinLayout=nil;
(5)    for each L⃗,/* section 3.5 */
(7)        SelectedEdge=nil;
(8)        for each edge e_i in SortE
(9)            SelectedEdge = SelectedEdge ∪ {e_i};
(10)           F=CheckCompatibility(SelectedEdge,L⃗); /*
       section 3.4 */
(11)           if ( F == FALSE )
(12)               SelectedEdge = SelectedEdge − {e_i};
(13)       Cost= GetCost(SortE,SelectedEdge);
(14)       if (MinCost > Cost)
(15)           Mem = LayoutMemory(SelectedEdge,L⃗);
(16)           MinCost = Cost;
(17)   return Mem;
```

Figure 3. Overview of SOA with loops.

metic instructions. The combination of n loop distance IDs is denoted using a distance vector $\vec{L} = (L_0, L_1, ..., L_n)$ in which L_0 is used for all address accesses outside any loop and $L_0 \equiv 1$.

Two important properties about loop distance indicators are as follows.

- It is flexible and independent. A loop distance identifier can change independently from the rest. For a program of N variables, the maximum value that a L_i can take to be N as it provides enough space to assign all other variables in between. With M loops, the possible combinations are N^M – referred as *distance space* in the paper.

- It is polymorphic. Even an edge in the procedural access graph is annotated with a L_i and L_i is assigned to a value bigger than 1, its associated two variables still have a choice to be allocated next to each other. For example, if we select two edges "$a - b/L_1$" and "$c - d/L_1$", it is valid to assign "a" and "b" with a distance of 4 (L_1=4) while "c" and "d" are allocated next to each other.

3.3 Procedural Access Graph

The procedural access graph is constructed by combining the access sequences from all basic blocks plus the

edges indicating the control flows between basic blocks. In this paper, each basic block is weighted by its execution frequency.

Compared to existing profiling based address assignment schemes [6], the major difference and enhancement is the introduction of distance vector for loops. All accesses from a loop i are marked by L_i during the construction of the access graph. As a result, multiple edges might co-exist between two nodes in the access graph.
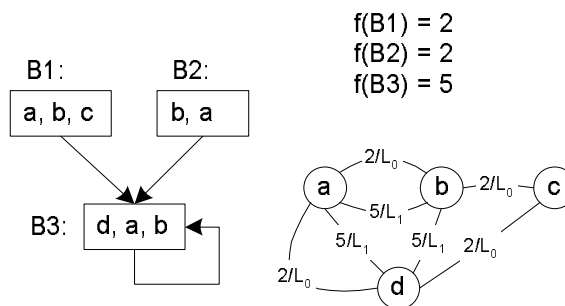

Figure 4. Generating Procedural Access Graph.

This step is explained by the example in Figure 4. There are two edges between "a" and "b": "ab/L_0" and "ab/L_1". They indicate the accesses in B_1 and in the loop body B_3 respectively. These two edges will be processed and selected independently from each other. However, if both are selected, their corresponding distance identifier will merge, i.e. take the same value. In this example, both would be 1.

3.4 Checking the Compatibility of Selected Edges

Our algorithm greedily selects edges of higher weights while sequentially processing the sorted edge list. An edge is inserted to the selected edge list if it is possible to form a valid memory layout with all previously selected edge and this edge.

Given a set of selected edges, a fast checking algorithm is given in Figure 5. Mem keeps a set of disjoint memory segments. $\vec{L}$ is the distance vector such that all nodes in each segment have deterministic relative position. For each edge $(a, b)/L_i$ being processed, we distinguish the following cases:

- If neither a nor b appear in some segment, they form a new segment.

- If a and b appear in two different segments, we try to combine these two segments. They are compatible if we do not have to allocate two nodes into the same location in the combined segment.

- If both nodes are in a same segment, it is compatible if the distance between a and b in the segment is L_i.

- If one of them is in a segment and the other is not, we try to put the new node into that segment. They are compatible if the relative position in the segment is available for allocation.

```
Compatibility Checking Algorithm:
Input:
        SE: a set of marked edges, each edge e_i has a loop
        distance id L_i;
        L⃗: distance vector;
Output:
        1(0): a (non-)compatible set

(1)     Mem = nil;
(2)     for each edge (n1, n2)/L_i
(3)         /* find two memory segments that include n1,n2 */
(4)         MSeg_1 : MSeg_1 ∈ Mem, n1 ∈ MSeg_1;
(5)         MSeg_2 : MSeg_2 ∈ Mem, n2 ∈ MSeg_1;
(6)         case 1: ( MSeg_1 == MSeg_2 == nil )
(7)             NewSeg = a new memory segment
                    where n1,n2 ∈ NewSeg and
                    Distance(n1, n2)= L_i
(8)             Mem = Mem ∪ NewSeg;
(9)         case 2: ( (MSeg_1 != MSeg_2) and both are not nil )
(10)            p1 = n1 position in MSeg_1
(11)            p2 = n2 position in MSeg_2
(12)            ∀V ∈ MSeg_2, p_v = position in MSeg_2
(13)                assign V to (p1 + p2 − p_v in MSeg_2
(14)            if no conflicts
(15)                update Mem is successful
(16)                return 1;
                else
(17)                return 0;
(18)        case 3: ( (MSeg_1 == MSeg_2) and both are not nil
)
(19)            if Distance(n1, n2) != L_i
(20)                return 0;
                else
(21)                return 1;
(22)        case 4: MSeg_1 != nil, MSeg_2 == nil
(23)            p1 = n1 position in MSeg_1
(24)            if (position p1 + L_i in MSeg_1 is available)
(25)                put n2 at p1 + L_i in MSeg_1
(26)                return 1;
(27)            if (position p1 − L_i in MSeg_1 is available)
(28)                put n2 at p1 + L_i in MSeg_1
(29)                return 1;
(30)            return 0;
(31)        case 5: MSeg_1 == nil, MSeg_2 != nil
(32)            /* similar as case (4) */
(33)     return 1;
```

Figure 5. Compatibility Check Algorithm.

Each node can find its associated segment in O(1) by keeping a link from each node to the segment. The complexity of the algorithm is O(NlogE) where E, N are the number of marked edges and related nodes. The worst case of the algorithm happens when each of the first half of the marked edges form a separate partial layout and each of the second half of the marked edges combines two segments.

3.5 Searching the Distance Space

The most time consuming part of the algorithm is the exploration of distance vector $\overrightarrow{L}$. For each loop distance id L_i, it can vary from 1 to the number of total different variables. For a loop distance vector of 5 items, i.e. 4 loop ids plus constant L_0, a brute force search approach can be done as follows.

```
for(L_1=1; L_1 <=VarNum; L_1++)
    for(L_2=1; L_2 <=VarNum; L_2++)
        for(L_3=1; L_3 <=VarNum; L_3++)
            for(L_4=1; L_4 <=VarNum; L_4++)
                ...
```

Clearly, it is too expensive. Moreover, our experimental results never gave a combination of high values. As a result, we heuristically explore the distance space by searching all combinations of small values plus those that have a large value in one dimension.

```
for(L_1=1; L_1 <=5; L_1++)
    for(L_2=1; L_2 <=5; L_2++)
        for(L_3=1; L_3 <=5; L_3++)
            for(L_4=1; L_4 <=5; L_4++)
                ...
```
and
$L_2 = L_3 = L_4 = 1$; for(L_1=6; L_1 <=VarNum/2; L_1++) ...
$L_1 = L_3 = L_4 = 1$; for(L_2=6; L_2 <=VarNum/2; L_2++) ...
$L_1 = L_2 = L_4 = 1$; for(L_3=6; L_3 <=VarNum/2; L_3++) ...
$L_1 = L_2 = L_3 = 1$; for(L_4=6; L_4 <=VarNum/2; L_4++) ...

For the above example, suppose we have 20 variables, i.e. VarNum=20, we need to explore (5x5x5x5)+(4x(20/2-5))= 645 combinations. It is only 0.4% of the entire distance space (20x20x20x20 =160,000). As shown in the experimental section, this heuristic performs well in practice.

4 General Offset Assignment with Loops (GOA/Loop)

Modern DSP processors provide multiple ARs to help the generation of efficient code. TMS320C54x [8], for example, has 8 ARs, however, it has only one MR. We extend the SOA/Loop algorithm to this case.

Current general offset assignment(GOA) algorithms take a two-step approach: in the first step, variables are heuristically divided into different ARs; MR is used in the second step to reduce the number of addressing instructions. Since variable distribution has no information about the availability of the MR, the address jumps generated from the use of different registers could not effectively eliminated by one MR.

Figure 6 presents an algorithm for DSP processors with N (N≥2) ARs but only one MR. It takes similar steps as shown in [4] and calls the SOA/Loop algorithm discussed in previous section as a subroutine. We use similar heuristic search approach to explore the distance vector space as discussed in section 3.5.

An extension could be further made to apply our algorithm to N ARs with N MRs: each edge is annotated by $L(i, j)$ where i denotes that it belongs to loop i and j denotes that it belongs to register j. j is determined in processing the edges.

```
GOA/Loop Algorithm:
Input:
        CFG: control flow graph of the procedural;
        AS_i: access sequences for all basic block B_i;
        NP_i: execution profiles for all basic block B_i;
        R: the number of available ARs
Output:
        Mem: Memory Layout Scheme

(1)     G = GetProceduralAccessGraph(CFG, AS, NP). /*
        section 3.3 */
(2)     for each L⃗, /* section 3.5 */
(3)        Mem=nil;
(4)        for each node in the graph
(5)           MinCostIncrease = MAX;
(6)           for each R_i
(7)              NewCost_i = cost of SOA/Loop() for R_i
        with new added node;
(8)              CostIncr = NewCost_i - OldCost_i;
(9)              OldCost_i = NewCost_i;
(10)          if NewCost_i < MinCostIncrease
(11)             MinCostIncrease = NewCost_i;
(12)             x = i;
(13)          assign this node to R_x;
(14)       Cost=sum of the cost of SOL/Loop() for each R_i
(15)       if ( Cost < MinCost)
(16)          Mem = sequentially list the layout of variables
        for each register
(17)          MinCost = Cost;
(18)    return Mem;
```

Figure 6. Overview of GOA with loops.

5 Experimental Results

We implemented and evaluated our algorithm with comparison to other approaches. We ran on a Pentium IV processor 2.0MHz, 512MB RAM. With the heuristic distance space exploration technique, our algorithm finished in several seconds.

5.1 Number of Addressing Instructions

We evaluated our proposed algorithm using DSP-STONE benchmark suite [9] as well as randomly generated access sequences. For DSPSTONE benchmark, we pick up 7 programs with loops. These program are compiled using TI provided compiler cl500 [13] and we collected their memory access sequences from a functional simulator of TI TMS320C54x [11]. We only consider scalar variables in this paper and each array item is treated as a variable.

Figure 7 compares our results to SOL-SOA [4] plus post-pass MR optimization for DSPSTONE benchmarks. In the experiment, we put a limit of extra memory requirements up to 10%. The first four columns are benchmark name, the number of variables, the dynamic sequence length, and the number of identified loops. We distinguish up to 5 loops of longest runtime execution sequences. The experiment are SOA, GOA with 2 ARs, and GOA with 4 ARs. Significant percentage improvements are achieved for both SOA and GOA. On average, 9.9%, 17.1% and 21.8% are observed respectively.

We also collect the results for randomly generated access sequences. The reductions of dynamic instruction count, compared to different assignment algorithms, are summarized in Figure 8 (we omit similar results for static instruction count due to space limit). Since random generated sequences show similarly improvement as that of program sequences, in the rest of the paper, we use the dynamic random sequence in conducting various other experiments.

In Figure 8, column 1 shows the number of different variables ($|V|$) and the static access sequence length ($|S|$) and the dynamic access sequence length ($|D|$). Column 2 to 4 list the number of loops in the procedural (N), the length of each loop (L) and the execution frequency of each loop (F). Column 5 to 8 list the results using OFU (the order of the first used) approach and Sol-SOA [4]. All of them are optimized using one MR. Column 5 and 7 weight each basic block equally. Column 6 and 8 weight each basic block with its execution frequency. Column 9 lists the results using our proposed approach SOA/Loop. Last two columns show the the percentage improvements over OFU and Solve-SOA with execution profiles. We reduce the number of address arithmetic instructions over D-OFU+MR and D-SOA+MR for up to 29.64% and 25.89% in static case, 60.56% and 21.44% in dynamic case. On average, we improve 16.1% over Sol-SOA with MR optimization.

As discussed, a considerable time has to be spent in order to fully explore the distance space. Figure 9 listed the comparison of the results using fully exploration approach and the heuristic algorithm we proposed in section 3. Column 5 lists the instruction count increase due to less space exploration. Column 6 lists the percentage of the candidates that we have to explore in our heuristic algorithm. Column 7 lists the execution time of the heuristic algorithm. On average, we achieved very few performance degradation by exploring a small subset of the distance space.

Name	Var #	Seq. Len.	Loop #	SOA +MR			GOA-2 +MR			GOA-4 +MR		
				Sol-	Loop-	% Impr.	Sol-	Loop-	% Impr.	Sol-	Loop-	% Impr.
adpcm	36	95	4	30	25	16.7 %	4	3	25.0%	4	3	25.0%
convolution	113	179	4	77	64	16.8 %	25	26	4.0 %	4	4	0.0%
fir2dim	87	339	5	184	174	5.4 %	95	79	16.8%	30	20	33.3%
fir	60	347	5	270	197	27.0 %	198	117	40.9%	158	128	19.0%
iir	43	240	4	91	91	0.0 %	17	17	0.0%	11	6	45.5%
lms	60	349	4	199	191	4.0 %	134	108	19.4%	35	31	11.4%
real-update	74	361	4	213	213	0.0 %	141	122	13.5%	44	36	18.2%

Figure 7. Dynamic Addressing Instruction Counts for DSPSTONE Benchmarks.

	Loops			S-OFU +MR	D-OFU +MR	S-SOA +MR	D-SOA +MR	SOA/Loop +MR	Gain	
$|V|/|S|/|D|$	N	L	F						New/D-OFU+MR	New/D-SOA+MR
10/20/60	1	10	5	25.24	25.24	17.44	13.76	12.06	52.22 %	12.35 %
10/20/60	2	5	5	18.70	18.70	13.86	10.18	8.42	54.97 %	17.29 %
10/30/90	3	5	5	34.00	34.00	26.80	22.00	18.00	47.06 %	18.18 %
10/40/120	2	10	5	48.90	48.90	38.80	31.30	27.40	43.97 %	12.46 %
10/60/180	3	10	5	84.80	84.80	62.70	59.80	49.70	41.39 %	16.89 %
20/30/90	3	5	5	37.30	37.30	27.30	18.80	16.10	56.84 %	14.36 %
20/40/120	2	10	5	66.30	66.30	43.50	32.50	29.20	55.96 %	10.15 %
40/60/180	3	10	5	101.30	101.30	76.80	59.70	46.90	53.70 %	21.44 %
60/90/180	3	10	5	85.10	85.10	59.80	42.10	34.60	59.34 %	17.81 %
60/80/240 (*)	4	10	5	121.20	121.20	88.00	59.20	47.80	60.56 %	19.26 %

Note: The results are averaged numbers over 20 runs.

(*) we didn't run full exploration for this case.

Figure 8. SOA Results (Dynamic Addressing Instruction Count) for Randomly Generated Sequences.

5.2 Memory Requirements

The memory requirements for normal schemes and SOA/Loop are compared in Figure 10. We show the results for the memory layout with minimal dynamic address arithmetic instruction count, i.e. the layout of column 9 in Figure 8.

If memory size is a bottleneck, SOA/Loop could be run with memory restrictions. When 5%, 10%, and 20% extra memory space, Figure 11 shows respectively the increase of dynamic address arithmetic instructions in percentage with comparison to the best results (column 9 in Figure 8). With 5% of extra space, the algorithm degrades by only 3.5% on average leaving still 12% improvement.

5.3 GOA results

GOA results are shown in Figure 12. Compare to the scheme using Sol-GOA plus MR optimization, we achieve good improvements even with large number of registers, e.g. 26.5% improvement for 8 registers with 80 variables.

6 Conclusion

In this paper, we presented a novel address offset assignment approach. It explicitly uses the modify register to optimize access sequences from loops. By assigning each access edge with a unique loop identifier, this approach effectively assigns related variables within a loop with equal distances. By taking advantage of the MR during offset assignment, significant reductions of both static and dynamic address arithmetic instructions have been achieved.

The paper also exploits several implementation issues. A simple heuristic algorithm is proposed to significantly reduce the distance space that we need to explore. It achieves good results with very low overhead. The extra memory required for allocation is generally very small and a further evaluation showed that the algorithm performs well even with very restricted extra memory.

References

[1] S.Liao and S. Devadas and K. Keutzer and S. Tjiang and A. Wang, 'Storage Assignment to Decrease Code Size,' *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, 1995.

[2] Yoonseo Choi and Tasewhan Kim, "Address Assignment Combined with Scheduling in DSP Code Generation," *Proceedings of ACM Design Automation Conference*, pages, 2002.

[3] Amit Rao and Santosh Pande, 'Storage assignment optimizations to generate compact and efficient code on embedded DSPs,' *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, 1999.

COMPUTER SOCIETY

$|V|/|S|/|D|$	Loops			Dyn. Inst. Incr.(%)	Explored Space(%)	Exec. Time(s)
	N	L	F			
10/20/60	1	10	5	0.00 %	50.0 %	0.6
10/20/60	2	5	5	0.00 %	25.0 %	1.2
10/30/90	3	5	5	0.00 %	12.5 %	4.0
10/40/120	2	10	5	0.00 %	25.0 %	1.3
10/60/180	3	10	5	0.00 %	12.5 %	4.7
20/30/90	3	5	5	0.00 %	1.75 %	4.7
20/40/120	2	10	5	1.03 %	8.75 %	1.6
40/60/180	3	10	5	2.13 %	0.27 %	10.1
60/60/180	3	10	5	3.88 %	0.09 %	39.0
60/80/240	4	10	5	-% (*)	0.01 %	85.3

Figure 9. Heuristic Distance Space Exploration.

(*) we didn't run full exploration for this case.

$|V|/|S|/|D|$	Loops			SOA/Loop over Optimal
	N	L	F	
10/20/60	1	10	5	101.16 %
10/20/60	2	5	5	104.29 %
10/30/90	3	5	5	102.27 %
10/40/120	2	10	5	103.33 %
10/60/180	3	10	5	104.44 %
20/30/90	3	5	5	112.90 %
20/40/120	2	10	5	121.14 %
40/60/180	3	10	5	158.31 %
60/60/180	3	10	5	146.60 %
60/80/240	4	10	5	143.89 %

Figure 10. Extra Memory Required in SOA/Loop.

$|V|/|S|/|D|$	Loops			5%	10%	20%
	N	L	F			
10/20/60	1	10	5	0.00 %	0.00 %	0.00 %
10/20/60	2	5	5	0.48 %	0.48 %	0.48 %
10/30/90	3	5	5	2.98 %	2.98 %	0.00 %
10/40/120	2	10	5	0.00 %	0.00 %	0.00 %
10/60/180	3	10	5	0.00 %	0.00 %	0.00 %
20/30/90	3	5	5	4.35 %	3.11 %	1.24 %
20/40/120	2	10	5	3.08 %	3.08 %	2.40 %
40/60/180	3	10	5	4.69 %	3.62 %	3.41 %
60/60/180	3	10	5	7.31 %	4.12 %	1.99 %
60/80/240	4	10	5	11.24 %	1.67 %	0.00 %

Figure 11. Performance Reduction with Restricted Extra Memory.

$|V|/|S|/|D|$	Loops			AR #	Sol-GOA + MR	Loop-GOA + MR	Improvement (%)
	N	L	F				
20/30/70	2	5	5	2	8.2	8.2	0.0 %
20/45/105	3	5	5	2	14.2	12.6	11.3 %
20/45/105	3	5	5	4	4.3	3.7	20.0 %
30/90/210	3	10	5	4	20.8	16.8	19.2 %
40/90/210	3	10	5	4	24.2	20.6	14.9 %
80/120/280	4	10	5	4	37.3	33.3	10.6 %
80/120/280	4	10	5	8	11.3	8.3	26.5 %

Note: The results are averaged numbers over 20 runs.

Figure 12. GOA/Loop for Randomly Generated Sequences.

[4] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," *Proceedings of IEEE/ACM international conference on Computer-aided design*, pages 109-112, San Jose, CA, 1996.

[5] R. Leupers and F. David, "A Uniform Optimization Technique for Offset Assignment Problems," *International Symposium on System Synthesis*, pages 3-8, 1998.

[6] A. Sudarsanam and S. Liao and S. Devadas, "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," *Design Automation Conference*, pages 287-292, 1997.

[7] S. Udayanarayanan and C. Chakrabarti, "Address Code Generation for Digital Signal Processors," *Design Automation Conference*, pages 287-292, 2001.

[8] Texas Instruments, "TMS320C54x DSP Reference Set: CPU and Peripherals", March 2001.

[9] V. Zivojnovic, J. Martinez Velarde, C. Schlager, and M. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, volume 1, page 715-720, Dallas, TX, USA, October 1994.

[10] P. Lapsley, J. Bier, A. Shoham, and EA Lee, "DSP Processor Fundamentals: Architectures and Features," Berkeley Design Technology, Inc., 1996.

[11] TI TMS320C54x functional simulator. http://www.utdallas.edu/~zhangyt/c54sim/.

[12] Texas Instruments, "TI TMS320C54x Code Generation Tools: Getting Started Guide," March 1997.

[13] Texas Instruments, "TMS320C2x/C2xx/C5x Optimizing C Compiler User Guide," August 1999.

[14] W.K. Cheng, Y.L. Lin, "Addressing optimization for loop execution targeting DSP with auto-increment/decrement architecture," *International Symposium on Systems Synthesis*, 1998, Hsinchu, Taiwan.

[15] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, and S. Malik, "Optimal Live Range Merge for Address Register Allocation in Embedded Programs," *in Proceedings of the 10th International Conference on Compiler Construction, CC2001, LNCS 2027*. April 2001, pp 274-288, Springer-Verlag.

[16] M. Cintra, G. Araujo, "Array Reference Allocation Using SSA-Form and Live Range Growth," *in Proceedings of the ACM SIGPLAN LCTES 2000*, June 2000, pp. 26-33.

[17] R. Leupers, A. Basu, and P. Marwedel, "Optimized array index computation in DSP programs," in *Proceedings of the IEEE ASP-DAC*. February 1998.

[18] B. Wess, "Minimization of data address computation overhead in DSP programs," *Kluwer International Journal on Design Automation for Embedded Systems,* vol. 4, pp. 167-185, March 1999.

IEEE COMPUTER SOCIETY