An Effective Heuristic for Simple Offset Assignment with Variable Coalescing

Hassan Salamy and J. Ramanujam

Department of Electrical and Computer Engineering
and Center for Computation and Technology
Louisiana State University, Baton Rouge, LA 70803, USA
{hsalam1,jxr}@ece.lsu.edu

Abstract. In many Digital Signal Processors (DSPs) with limited memory, programs are loaded in the ROM and thus it is very important to optimize the size of the code to reduce the memory requirement. Many DSP processors include address generation units (AGUs) that can perform address arithmetic (auto-increment and auto-decrement) in parallel to instruction execution, and without the need for extra instructions. Much research has been conducted to optimize the layout of the variables in memory to get the most benefit from auto-increment and auto-decrement. The simple offset assignment (SOA) problem concerns the layout of variables for machines with one address register and the general offset assignment (GOA) deals with multiple address registers. Both these problems assume that each variable needs to be allocated for the entire duration of a program. Both SOA and GOA are NP-complete. In this paper, we present a heuristic for SOA that considers coalescing two or more non-interfering variables into the same memory location. SOA with variable coalescing is intended to decrease the cost of address arithmetic instructions as well as to decrease the memory requirement for variables by maximizing the number of variables mapped to the same memory slot. Results on several benchmarks show the significant improvement of our solution compared to other heuristics. In addition, we have adapted simulated annealing to further improve the solution from our heuristic.

1 Introduction

Embedded processors are found in many electronic devices such as telephones, cameras, and calculators. Due to the tight constraints on the design of embedded systems, memory is usually limited. In contrast, the memory requirement for the execution of digital signal processing and video processing codes on an embedded system is significant. Moreover, since the program code resides in the on-chip ROM, the size of the code directly translates into silicon. So code minimization becomes a substantial goal in order to optimize the amount of memory needed.

Many Digital Signal Processors (DSPs) such as the TI C2x/C5x, Motorola 56xxx, Analog Devices 210x and ST D950 have address generation units (AGUs) [5]. The AGU is responsible for calculating the effective address. A typical AGU consists of an address register file and a modify register file as shown in Figure 1. The architectures of such DSPs support only indirect memory addressing. Since the base-plus-offset addressing mode is not supported, an extra instruction is needed, in general, to add (subtract) an offset to (from) the current address in the address register to compute the new address. However, such architectures support auto-increment and auto-decrement of the address register. When there is a need to add an offset of 1 or subtract an offset of 1 from the current address, this can be done in parallel with the same LOAD/STORE instruction using auto-increment or auto-decrement; and this does not require an extra address arithmetic instruction in the code. Exploiting this characteristic will lead to code compaction and thus less memory used since the length of the code in DSP directly translates into required silicon area. One method for minimizing the instructions needed for address computation is to perform *offset assignment* of the variables. Offset assignment refers to the problem of placing the variables in the memory to maximally utilize auto-increment/decrement and thus reduce code size.

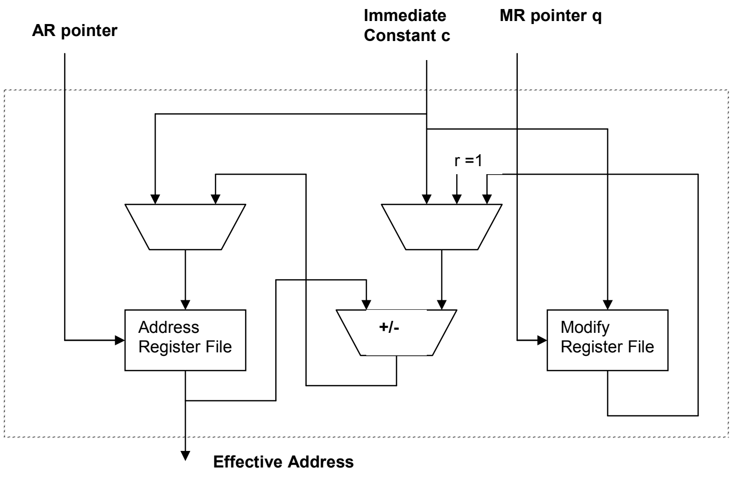

Fig. 1. A typical Address Generation Unit (AGU) contains a modify register file, address register file and ALU

Simple offset assignment (SOA) refers to the case where there is only one address register (AR), whereas general offset assignment (GOA) refers to the case where there are multiple address registers [12]. In both SOA and GOA considered in this paper, the value of auto-increment/decrement is 1; SOA and GOA are NP-complete [12]. Several researchers have studied the offset assignment problem and have proposed different heuristics.

In this paper, we present an effective heuristic for the simple offset assignment problem with *variable coalescing*. Coalescing allows two or more variables to share the same memory location provided that their live ranges do not overlap. Based on the live ranges of all the variables, an interference graph (IG) is constructed in which an edge (a, b) indicates that variables a and b interfere and thus they can not be mapped into the same memory location. Variable coalescing improves the results by decreasing the number of address arithmetic instructions needed as well as the memory requirement for storing the variables.

The remainder of the paper is organized as follows. Section 2 presents related work in this area. Section 3 presents our algorithm for simple offset assignment with variable coalescing. Section 4 gives an example that shows how our algorithm works. Section 5 presents the simulated annealing algorithm to further improve the results. Section 6 summarizes the results. Finally Section 7 presents our conclusions.

2 Related Work

The problem of simple offset assignment was first discussed by Bartley [2]. Then Liao et al. [12] showed that the SOA problem is NP-complete and that it is equivalent to the Maximum Weight Path Cover (MWPC) problem. They proposed heuristics for both SOA and GOA. Given an access sequence of the variables, the access graph has a node for each variable with an edge of weight w between nodes a and b meaning that variables a and b appear consecutively w times in the access sequence.In this greedy heuristic, edges are selected in decreasing order of their weights provided that choosing an edge does not introduce a cycle and it does not result in a node of degree more than two. Finally, the access graph considering only the selected edges will determine the placement of the variables in the memory. One possible result of applying Liao's heuristic to the access sequence in Figure 2(a) is shown in Figure 2(c), where the bold edges are the selected edges and the final offset assignment is [*ebacd*]. The cost of a solution is the sum of the weights of all unselected edges (i.e., non-bold edges). For the example in Figure 2(a), the cost is 1 which represents the non-bold edge that refers to the one address arithmetic operation needed to go from a to e in the access sequence since variables a and e are mapped to non-consecutive memory locations.

Leupers and Marwedel [9] extended Liao's work by proposing a tie-break heuristic for the SOA problem. Liao et al. did not state what happens if two edges have equal weight. Leupers and Marwedel used the following tie-break function: if two edges have the same weight, they pick the edge with the smaller value of the tie-break function $T_2(a, b)$ defined for an edge (a, b) as in equation 5.

Atri et al. [1] solved the SOA problem using an incremental approach. They tried to overcome some of the problems with Liao's algorithm, mainly in the case of equal weight edges as well as the greedy approach of always selecting the maximum weight edges. Starting with an initial offset assignment (which could be the result of any SOA heuristic), their incremental-SOA tries to explore more points in the solution space by considering the effect of selecting currently unselected edges.

Leupers [7] compared several algorithms for simple offset assignment. Ottoni et al. [13] studied the simple offset assignment problem with variable coalescing (CSOA). Their algorithm uses liveness information to construct the interference graph. In the interference graph, the nodes represent variables and an edge between two variables means that they interfere and thus they can not be coalesced. The authors used the SOA heuristic proposed by Liao et al. [12] enhanced with the $tie-break$ in [9], with the difference that at each step the algorithm chooses between (i) coalescing two variables; and (ii) selecting the edge with the maximum weight as in Liao's algorithm. Their algorithm finds the pair of nodes that can be coalesced with maximum $csave$ where $csave$ represents the actual saving from coalescing this pair of nodes. At the same time, it finds the edge with the maximum weight w that can be selected using Liao's algorithm. If there are candidates for both coalescing and selection, then it will use coalescing if $csave$ is larger than w, otherwise use selection.

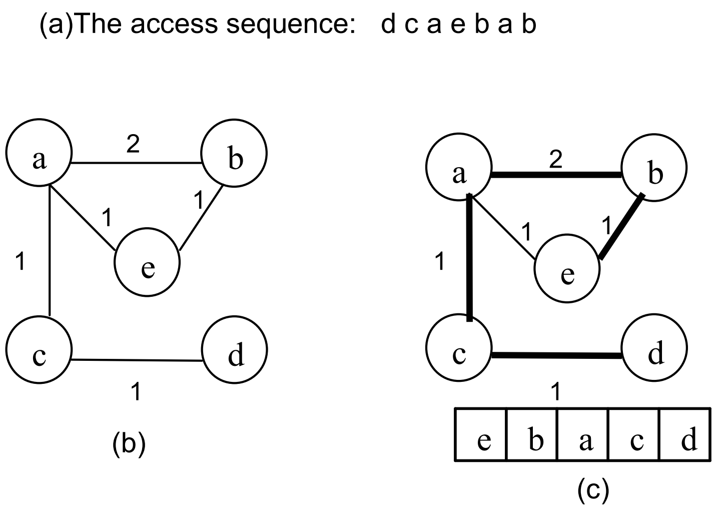

Fig. 2. (a) Access sequence. (b) Access graph corresponding to the access sequence. (c) Offset assignment where bold edges represent the selected edges and the cost of such assignment is 1.

In [21], the authors studied the cases of SOA with variable coalescing at the same time as [13]. Their coalescing algorithm first separates values into atomic units called webs by applying variable renaming. Their proposed heuristic starts by applying pre-iteration coalescing rules. Then the algorithm picks the two variables (i.e., nodes) with maximum saving for coalescing provided that they respect the validity conditions. If the saving is positive, then the two nodes are coalesced. Liao's SOA will then be applied to the new access graph. This process will continue as long as there are two variables that can be coalesced. Several others [10], [15], [16], [17], [19], [20] have addressed problems related to offset assignment.

3 CSOA: Offset assignment with variable coalescing

In simple offset assignment (SOA), each memory location or slot is assigned only one variable. Simple offset assignment with variable coalescing (CSOA) refers to the case where more than one variable can be mapped into the same memory location. Variable coalescing is intended to decrease the memory requirement by further decreasing the number of address arithmetic instructions as well as by decreasing the memory requirements for storing the variables. Two variables can be coalesced if their live ranges do not overlap at any time which means that at any time, those two variables are not needed to be simultaneously live.

In CSOA, an interference graph (IG) is constructed by examining the live ranges of all the variables. Each node in the graph represents a variable, and an edge between two nodes means they interfere and thus they cannot be coalesced. Two variables can be coalesced if they meet all the following conditions:

– the two variables do not interfere;
– after coalescing, no node in the access graph has more than two selected edges incident at it; (and)
– the resulting access graph is still acyclic considering only the selected edges.

So instead of always selecting an edge as in SOA, CSOA can either select an edge or coalesce two variables that meet the three conditions listed above.

Our algorithm presented in Figure 3 integrates both selection and coalescing options in a way to minimize the total cost, which is represented by the number of address arithmetic instructions, as well as to decrease the memory requirement for storing the variables in memory. The algorithm takes as an input, the interference graph (IG) and the access sequence, and outputs the mapping of the variables to memory locations possibly with coalescing. From the access sequence, it constructs the access graph (AG) which captures the frequency of consecutive occurrence of any two variables in the access sequence. Then it sorts the edges whose end-point vertices interfere in decreasing order of their weights as a guide for selection. Since one of the purposes of the heuristic is to decrease the memory requirement for storing the variables, an edge (a, b) such that $(a, b) \notin$ IG will not be considered for selection. Such an edge will be a candidate for coalescing which means that fewer edges will be considered for selection and thus more variables will probably be coalesced. Note that the selection of an edge may prevent variable coalescing opportunities in the future. So only those edges whose endpoints interfere will be considered as candidates for selection in each iteration of the algorithm.

Any two variables that do not interfere are considered as candidates for coalescing. In each iteration, all pairs of variables that meet the three conditions for variable coalescing (mentioned earlier) are candidates for coalescing. We define the following values:

$$\begin{aligned}
Gain(a,b) &= \frac{Actual\_Gain(a,b)}{Possible\_Loss(a,b)} && (1)\\
Actual\_Gain(a,b) &= W(a,b) \\
&+ \sum_{\substack{x\in Adj(a)\cap Adj(b)\\ (b,x)\in Selected\_Edge\\ (a,x)\notin Selected\_Edges}} W(a,x) \\
&+ \sum_{\substack{y\in Adj(a)\cap Adj(b)\\ (b,y)\notin Selected\_Edges\\ (a,y)\in Selected\_Edges}} W(b,y) && (2)\\
Possible\_Loss(a,b) &= 1 + \sum_{\substack{(a,x)\notin IG,(b,x)\in IG\\ (b,x)\notin Selected\_Edges}} (a,x) \\
&+ \sum_{\substack{(b,y)\notin IG,(a,y)\in IG\\ (a,y)\notin Selected\_Edges}} (b,y) && (3)
\end{aligned}$$

A *Gain* value for each of these candidate pairs is calculated that captures the benefit of coalescing as well as the possible loss of future opportunities for coalescing. The value $Gain(a,b)$ is defined as the actual saving that results from coalescing variables a and b divided by the possible loss of future coalescing opportunities due to coalescing a and b. When variables a and b are coalesced, all edges incident at a and b of the form (a,x) and (b,x) will be merged, and if edge (a,b) exists, it will be deleted. When edges (a,x) and (b,x) are merged into edge (ab,x), if at least one of the edges was already selected, then (ab,x) is also considered to be selected. The value $Gain(a,b)$ is defined as shown in Equation 1 and the value $Actual\_Gain(a,b)$ is defined in Equation 2. The value $Actual\_Gain(a,b)$ is basically the sum of the weights of the edges incident at a or b that were not selected before and became selected after being merged with a selected edge plus the weight of the edge (a,b).

The value $Possible\_Loss(a,b)$ is defined in Equation 3 as the sum of the edges (a,x) such that $(a,x)\notin$IG, (b,x) is not selected, and $(b,x)\in$ IG plus the sum of the edges (b,y) such that $(b,y)\notin$IG, (a,y) is not selected, and $(a,y)\in$IG. As depicted in equation 3, $Possible\_Loss(a,b)$ considers only vertices that are neighbors to a or b. Although other definitions of the loss can be used, we found that our definition captures the possible effect of coalescing on solutions that can be constructed. Even though coalescing involves vertices and not edges, using the number of edges as the essence for the loss in Equation 3 leads to better results. The rationale behind this is that an edge whose corresponding vertices interfere will probably end up as a selected edge and thus it may prevent some coalescing opportunities and as a result it may degrade the quality of the final solution.

It is worth noting that although our heuristic integrates both selection and coalescing, it gives priority to coalescing, which can be clearly deduced from the definition of loss. We believe this is one of the main reasons for our improvements in terms of the cost as well as the memory requirement for storing the variables. We divide the value $Actual\_Gain(a, b)$ with the value $Possible\_Loss(a, b)$ to account for the number of edges whose corresponding variables were interference-free and now interfere as a result from coalescing a and b. The reason behind this is that coalescing two variables with large *Possible_Loss* value may prevent some future coalescing opportunities and thus may prevent achieving smaller cost compared to coalescing two variables with smaller *Possible_Loss* value.

Among all the pairs that are candidates for coalescing, our algorithm picks the pair with the maximum *Gain*. If the algorithm is able to find a pair for coalescing as well as an edge for selection, then it will coalesce if the *Actual_Gain* from coalescing is greater than or equal to the weight of the edge considered for selection; otherwise, it will select the edge. One way our heuristic attempts to maximize the number of variables mapped to each memory location is to allow the coalescing of pairs of variables with zero *Gain* value (if possible) after no more variables with positive *Gain* can be coalesced.

Coalescing variables without a good guide may prevent possible improvements over the standard SOA solution. Consider the example in Figure 4. Figure 4(b) shows Liao's greedy solution. The cost of this offset assignment is 4. Figure 4(c) shows the solution using the algorithm in [13] whose cost is also 4. Although there is potential for improvement through variable coalescing, the algorithm in [13] fails to capture the improvement over Liao's solution. This is because the algorithm in [13] first chooses to coalesce vertices b and e since they have the maximum *csave*. However, this choice will prevent any future coalescing opportunities. Our algorithm alleviates this shortcoming by calculating the $Possible\_Loss(b, e) = 5$ and thus $Gain(b, e) = 3/5$. So our algorithm first picks a and b for coalescing since $Gain(a, b) = 1$; edge (b, e) will not be considered for selection since b and e do not interfere. The cost of the final solution of our algorithm is zero, as shown in Figure 4(d). For selection, we used two $tie-break$ functions T_1 and T_2 defined below,

$$T_1(a, b) = degree(a) + degree(b) \tag{4}$$

$$T_2(a, b) = \sum_{x \in Adj(a)} W(a, x) + \sum_{y \in Adj(b)} W(b, y) \tag{5}$$

where $T_1(a, b)$ is the sum of the degree of a and degree of b in the access graph. $T_2(a, b)$ is the Leupers $tie - break$ function defined as the sum of the weights of the edges that are incident at a plus the sum of the weights of the edges that are incident at b. If two edges that are candidates for selection have the same weight then we try to tie break using the function T_1; if T_1 cannot break the tie, we use T_2. An edge with smaller T_1 or T_2 will win the tie. If two pairs of variables (a, b) and (c, d) that are candidates for coalescing are such that $Gain(a, b)=$ $Gain(c, d)$, then we first try to break the tie using T_0 which is the *Actual_Gain* such that we choose the pair with the bigger *Actual_Gain*. If both candidate pairs have the same actual gain, then we tie break using T_1 followed by T_2, if needed.

```
Coalescence SOA Algorithm
Input: the Access sequence.
       the Interference graph IG.
Output: Offset assignment.

Build the access graph (AG) from the access sequence.
L = list of edges (x,y) such that (x,y) $\in$ IG in decreasing order of their weights using
$T_1$ then $T_2$ for tie break.
Coalesce = false.
Select = false.
Do
    Find a pair of nodes (a,b) for coalescing that satisfy:
        1. (a, b) $\notin$ IG.
        2. AG will still be acyclic after a and b are coalesced considering
           selected edges.
        3. No node will end up with degree > 2 considering selected edges.
        4. (a,b) has max Gain where Gain is calculated as in equation (1).
       where $T_0, T_1$, and $T_2$ are the three tie break functions used in that order.
    If such a pair of nodes is found, then Coalesce = true.

    Among the edges that belong to L pick the first edge (c,d) such that:
        1. Selecting (c,d) will not result in a cyclic AG considering selected edges.
        2. Selecting (c,d) will not result in a node with degree > 2 considering
           selected edges.
    If such an edge is found, then Select = true; remove (c,d) from L.

    If (Coalesce && Select)
        If (Actual_Gain(a, b) = Weight(c, d))
            Update access graph AG with (a, b) coalesced.
            Update interference graph IG with (a, b) coalesced.
            Update list L
        Else
            Select edge (c,d)
    Else
        if (Coalesce)
            Update access graph AG with (a,b) coalesced
            Update interference graph IG with (a,b) coalesced
            Update list L
        Else if (Select)
            Select edge (c,d)
While (Coalesce || Select)
Return offset assignment
```

Fig. 3. Our algorithm for Simple Offset Assignment with variable coalescing.

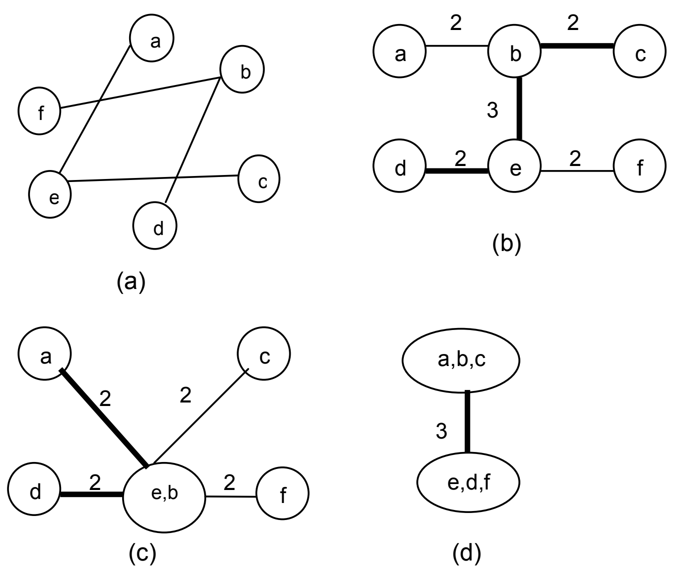

Fig. 4. (a) Interference Graph. (b) Liao's SOA greedy solution where the cost = 4. (c)The solution from the algorithm in [13] of cost 4 where it fails to capture the potential improvements from coalescing. (d) The optimal solution from our algorithm with cost = 0.

4 An Example

For the sake of clarity, consider the example in Figure 5 where Figure 5(a) shows the interference graph (IG) and Figure 5(b) shows the original access graph (AG). Figures 5(c)-(h) show how the access graph is updated when our heuristic is applied to this example. Although not shown, whenever two nodes are coalesced, the interference graph (IG) will be updated to reflect the coalescing of the nodes as well as to update the interference edges accordingly. Table 1 shows the step-by-step execution of our algorithm and the criteria used for choosing the candidates for selection and for coalescing. Note that in Table 1 we do not show coalescing candidates with zero *Gain*. Figure 5(i) shows the final solution with zero cost. If we run the algorithm in [13] on the same example presented in Figure 5, the cost of a possible final solution (which is shown in Figure 6) is 4.

5 Simulated Annealing

Since the offset assignment problem is NP complete, the heuristic presented in Section 3 will very likely produce a suboptimal solution. So in order to further improve the results, we used a simulated annealing approach. Simulated Annealing (SA) [3] is a global stochastic method that is used to generate approximate solutions to very large combinatorial problems. The technique originates from the theory of statistical mechanics, and is based on the analogy between the

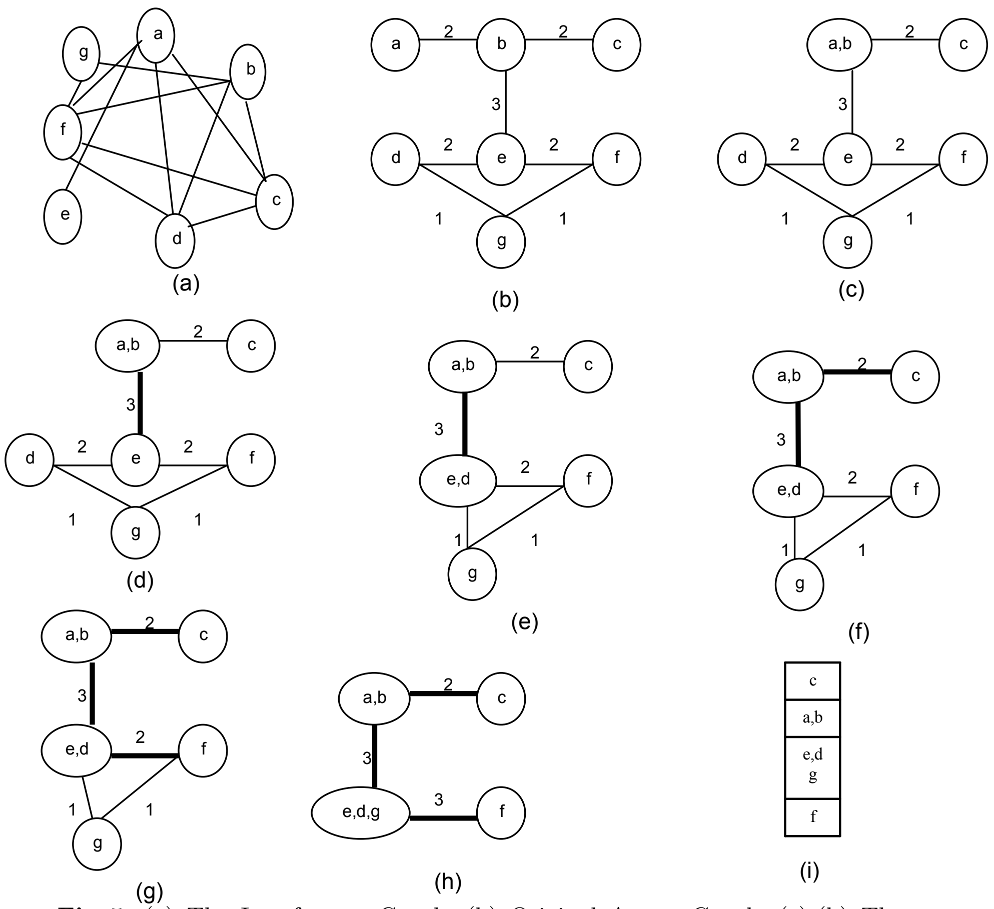

Fig. 5. (a) The Interference Graph. (b) Original Access Graph. (c)-(h) The access graphs after each iteration of our algorithm. (i) The final offset assignment, which incurs zero cost.

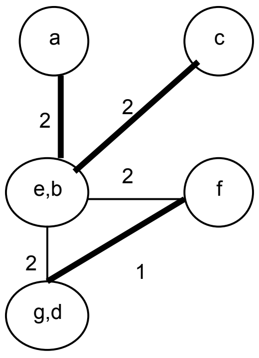

Fig. 6. One possible final solution for the example shown in Figure 5 using the algorithm in [13]

Table 1. A step by step run of our algorithm on the example in Figure 5

Iteration	Coalesce Candidate				Selection		Decision
	vertices	ActualGain	PossibleLoss	Gain	edge	Weight	
1	a,b	2	2	1			Coalesce(a,b)
	b,e	3	4	3/4	(b,c)	2	Tie-break T_0
	d,e	2	3	2/3	(g,f)	1	
	g,d	1	1	1			
	f,e	2	3	2/3			
2	d,e	2	2	1	(ab,e)	3	
	g,d	1	1	1	(ab,c)	2	Select (ab,e)
	f,e	2	2	1	(g,f)	1	
3	d,e	2	2	1			
	g,d	1	1	1	(ab,c)	2	Coalesce (d,e)
	f,e	2	2	1	(g,f)	1	Tie-break T_0
	c,e	2	3	2/3			
4					(ab,c)	2	Select (ab,c)
	ed,g	1	1	1	(ed,f)	2	Tie-break T_1
					(g,f)	1	
5	ed,g	1	1	1	(ed,f)	2	Select (ed,f)
					(g,f)	1	
6	ed,g	2	1	2	(g,f)	1	Coalesce (ed,g)

annealing process of solids and the solution procedure for large combinatorial optimization problems. The annealing algorithm begins with an initial feasible configuration, and then a neighbor configuration is created by perturbing the current solution. If the cost of the neighboring solution is less than that of the current solution, the neighboring solution is accepted; otherwise, it is accepted or rejected with some probability. The probability of accepting inferior solutions is a function of a parameter, called the temperature T, and the change in cost between the neighboring solution and the current solution. The temperature is decreased during the optimization process, and the probability of accepting an inferior solution decreases with the reduction of the temperature value. The set of parameters controlling the initial temperature, stopping criterion, temperature decrement between successive stages, and number of iterations for each temperature is called the *cooling schedule* [3]. Typically, at the beginning of the algorithm, the temperature T is large and an inferior solution has a high probability of being accepted. During this period, the algorithm. acts as a random search to find a promising region in the solution space. As the optimization progresses, the temperature decreases and there is a lower probability of accepting an inferior solution. The algorithm then behaves like a down hill algorithm for finding the local optimum of the current region.

Since simulated annealing requires a significant amount of time in order to converge to a good solution, we decided to use the final solution from our heuristic as the initial solution for SA and then ran SA for a short period of time with a low probability of accepting a bad solution. The neighbor function can perform one of the following operations:

– Exchange the content of two memory locations.
– Move the content of one memory location.
– Uncoalesce a coalesced node into two or more nodes.
– Coalesce two memory locations.

6 Results

We implemented our techniques in the $OffsetStone$ toolset [14] and we tested our algorithms on the $MediaBench$ benchmarks [4]. In Table 2, we compare our results with four different techniques used to solve the SOA problem, mainly Leupers' tie-break [9], incremental with Leupers' tie-break INC-TB[9][7], Genetic algorithm GA[8], and Ottoni's CSOA [13]. We measure the percentage of the number of address arithmetic instructions compared to Liao's algorithm [12]. Our heuristic drastically reduces the cost of simple offset assignment when compared to heuristics that do not allow variable coalescing since variable coalescing increases the proximity between variables in memory, thus it reduces the number of update instructions. Column 6 shows that our heuristic was able to outperform the CSOA heuristic [13] (results of which are shown in Column 5) in all the cases except for one benchmark. This improvement is due to the guide used in our choice between candidates for coalescing where we not only consider the actual saving but also an estimate of the possible loss in future coalescing opportunities. Also the idea of just considering edges whose endpoints interfere for selection increases the opportunity for coalescing nodes with maximum $Gain$ as defined in Equation 1. The ability to coalesce depends on the selected edges and vice-versa. So an algorithm that can choose the right candidates for selection and coalescing, at the right iteration and decide between them, should consider the influence of such a decision on future solutions. This is accounted for in our algorithm by defining the possible loss as a guide for the possible effect of coalescing on future solutions. The three $tie-break$ functions T_0, T_1, and T_2 play a role in achieving the clear improvements to the final solution. We do not show the comparison to the technique in [21] since the authors reported an average cost reduction of 33.3% when compared to [9] which is worse than the results achieved in [13].

Our simulated annealing (SA) algorithm further improved the results by searching the feasible region for better solutions starting from the final solution of our heuristic. Results in Table 2 column 6 shows that the SA further improved the results in all the cases in a short CPU time.

In Table 3, we show the reduction in memory slots needed to store the variables using our algorithm compared to that of using the algorithm presented in [13]. Results show that our algorithm drastically reduces the memory requirement by maximizing the number of variables that are assigned to the same memory location and it outperforms the CSOA algorithm [13] in all the cases.

The reason behind this reduction is that we defined the *Gain* from coalescing in terms of possible loss in coalescing opportunities as well as due to the fact that we did not consider the edges (a, b) such that $(a, b) \notin$ IG as candidates for selection and this will result in more opportunities for coalescing. However, the main reason for this improvement is that our heuristic allows zero *Gain* coalescing between nodes in the final AG. That is, we coalesce pairs of vertices (a, b) (if possible) such that $Gain(a, b) = 0$. This zero *Gain* coalescing will not reduce the cost in terms of the number of address arithmetic instructions but it will contribute to maximizing the number of variables mapped to a memory location. This explains the huge difference between the improvements in Table 2 and Table 3. Although a heuristic designed just to decrease the memory requirement for storing the variables can get better results than those in Table 3, it will be detrimental to the quality of the final solution in terms of the number of address arithmetic instructions. So our heuristic not only decreases the cost (which is defined as the reduction in the number of address arithmetic instructions), but also decreases the number of memory locations needed to store the variables.

Table 2. Comparison between different techniques for solving the SOA problem where column 1 shows different benchmarks, column 2 shows the results by applying Liao's + Tie-break [9], column 3 shows the results of the GA in [8], column 4 shows the results if the Tie-break [9] is combined with the incremental SOA in [1], and column 5 show the results in the case of SOA with variable coalescing [13], column 6 shows our results when applying our algorithm, column 7 shows the results using simulated annealing.

Benchmarks	TB (%) [9]	GA(%) [8]	INC-TB(%) [9][7]	CSOA(%) [13]	Our algorithm (%)	SA (%)
adpcm	89.1	89.1	89.1	45.6	42.1	39.1
epic	96.8	96.6	96.6	50.2	47	44.9
g721	96.2	96.2	96.2	27.9	26.2	23.2
gsm	96.3	96.3	96.3	19.4	14.8	13.5
jpeg	96.9	96.7	96.7	32.2	31	29.1
mpeg2	97.3	97.1	97.2	34.3	31.2	29.9
pegwit	91.1	90.7	90.7	38.8	39.5	36.1
pgp	94.9	94.8	94.8	32.2	29.8	27.4
rasta	98.6	98.5	98.5	21.1	19.9	19.5

7 Conclusions

The problem of offset assignment has received a lot of attention from researchers due to its great impact on code size reduction for DSPs. Reducing the code size is beneficial in the case of DSPs since the code is directly transformed into silicon area. Statistics show that codes for DSPs can have up to 50% address arithmetic instructions [18]. So the main idea of the ongoing research in this field

Table 3. The number of memory slots needed using our algorithm to the algorithm presented in [13].

Benchmarks	#Variables	#Memory slots [13]	#Memory slots our algorithm
adpcm	198	55	43
epic	4163	1125	767
g721	1152	289	199
gsm	4817	1048	433
jpeg	13690	4778	2555
mpeg2	8828	2815	1503
pegwit	4122	1454	910
pgp	9451	2989	1730
rasta	4040	1056	557

is to decrease the number of address arithmetic instructions and thus the code size. The problem is studied as simple offset assignment (SOA) and as general offset assignment (GOA), where different techniques and algorithms are used to tackle these problems with different modifications such as the inclusion of the modify-registers [9] as well as the case where the offset range is greater than 1. In this paper we presented a heuristic to solve the simple offset assignment with variable coalescing that chooses between selection and coalescing in each iteration by calculating the *Actual_Gain* and *Possible_Loss* for each pair of coalescing candidates. Results show that our algorithm not only decreases the number of address arithmetic instructions, but also drastically decreases the memory requirement for storing the variables by maximizing the number of variables that are mapped to the same memory slot. Simulated annealing further improved the final solution from our heuristic.

Acknowledgments. We are indebted to Sam Midkiff for a careful reading of this paper which has resulted in a significant improvement in the presentation of the paper. In addition, we thank the referees for their comments.We are indebted to Sam Midkiff for a careful reading of this paper which has resulted in a significant improvement in the presentation of the paper. In addition, we thank the referees for their comments.

References

1. S. Atri, J. Ramanujam, and M. Kandemir. Improving Offset Assignment for Embedded Processors. Languages and Compilers for High-Performance Computing, S. Midkiff et al. (eds.), Lecture Notes in Computer Science, Springer, 2001.
2. D.H. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. Software-Practice and Experience, vol. 22, no. 2, pp. 102-111, 1992.
3. S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. Science, 220, 4598, 671-680, 1983.

4. C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proc. IEEE International Symposium on Microarchitecture, pp. 330-335, 1997.
5. R. Leupers. Code Generation for Embedded Processors. In Proc. International System Synthesis Symposium, 2000.
6. R. Leupers. Code Optimization Techniques for Embedded Processors. Kluwer Academic Publishers, 2000.
7. R. Leupers. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. 12th International Conference on Compiler Construction (CC), Warsaw (Poland), Apr 2003, Springer Lecture Notes on Computer Science, LNCS 2622.
8. R. Leupers, F. David. A Uniform Optimization Technique for Offset Assignment Problems. 11th Int. System Synthesis Symposium (ISSS), 1998.
9. R. Leupers, P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. Int. Conference on Computer-Aided Design (ICCAD), 1996.
10. B. Li, R. Gupta. Simple Offset Assignment in Presence of Subword Data. CASES, ACM Press, 2003.
11. S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors. Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996.
12. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang. Storage Assignment to Decrease Code Size. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995.
13. D. Ottoni, G. Ottoni, G. Araujo, R. Leupers. Improving Offset Assignment through simultaneous Variable Coalescing. In Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03), in Springer LNCS 2826, pp. 285-297, Vienna, Austria, September 2003.
14. OffsetStone. http://www.address-code-optimization.org.
15. G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, and S. Malik. Optimal Live Range Merge for Address Register Allocation in Embedded Programs. In Proc. 10th International Conference on Compiler Construction, CC 2001, LNCS 2027, pp. 274-288. Springer, April 2001.
16. A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In SIGPLAN Conference on Programming Language Design and Implementation, pages 128-138, 1999.
17. A. Sudarsanam, S. Liao, and S. Devadas. Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures. In Design Automation Conference, pp. 287-292, 1997.
18. S. Udayanarayanan, C. Chakrabarti: Address Code Generation for Digital Signal Processors. 38th Design Automation Conference (DAC), 2001.
19. B. Wess and M. Gotschlich. Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem. In Int. Symp. on Circuits and Systems (ISCAS), 1997.
20. B. Wess, T. Zeitlhofer. Optimum Address pointer Assignment for Digital Signal Processors. ICASSP, IEEE 2004.
21. X. Zhuang, C. Lau, and S. Pande. Storage Assignment Optimizations Through Variable Coalescence for Embedded Processors. In Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems, pp. 220-231, 2003.