Offset Assignment Using Simultaneous Variable Coalescing

DESIREE OTTONI
Rutgers University
GUILHERME OTTONI
Princeton University
GUIDO ARAUJO
UNICAMP
and
RAINER LEUPERS
Aachen University of Technology

The generation of efficient addressing code is a central problem in compiling for processors with restricted addressing modes, like digital signal processors (DSPs). Offset assignment (OA) is the problem of allocating scalar variables to memory, so as to minimize the need of addressing instructions. This problem is called simple offset assignment (SOA) when a single address register is available, and general offset assignment (GOA) when more address registers are used. This paper shows how variables' liveness information can be used to dramatically reduce the addressing instructions required to access local variables on the program stack. Two techniques that make effective use of variable coalescing to solve SOA and GOA are described, namely coalescing SOA (CSOA) and coalescing GOA (CGOA). In addition, a thorough comparison between these algorithms and others described in the literature is presented. The experimental results, when compiling MediaBench benchmark programs with the LANCE compiler, reveal a very significant improvement of the proposed techniques over the other available solutions to the problem.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, code generation, optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Stack offset assignment, variable coalescing, autoincrement addressing modes, address registers, register allocation, DSPs

A preliminary version of this paper was presented at the 7th International Workshop on Software and Compilers for Embedded Systems, Vienna, September 2003, and received the best paper award.
This work was partially supported by FAPESP (Proc. 01/12762-5).
Authors' Addresses: Desiree Ottoni, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903; Guilherme Ottoni, Department of Computer Science, Princeton University, Princeton, New Jersey 08540; Guido Araujo, Institute of Computing, UNICAMP, Brazil; Rainer Leupers, Institute for Signal Processing Systems, RWTH Aachen University of Technology, Germany.

1. INTRODUCTION

Digital signal processors (DSPs) have been extensively used in modern consumer electronics. The increasing demand for new and complex applications running on these processors has brought a strong interest to compilers capable of generating efficient DSP code. DSPs are very irregular architectures, making them hard targets [Leupers 2000] to traditional compilation techniques designed for general-purpose processors [Aho et al. 1986; Muchnick 1997]. As a result, new techniques tailored to these processors have been intensively studied. Because of code size and performance constraints, many DSPs have no offset-based addressing mode. They usually provide a small register set capable of indirect addressing. To support indirect addressing, DSPs have specialized *address generation units* (*AGU*), which provide address computation in parallel to the execution in the main datapath. AGUs perform *autoincrement* (*decrement*) in address registers (AR) by some fixed values.[1] When a different value is demanded, the program is required to provide an explicit *update instruction* (prior to the memory access) in order to compute the memory address. This extra update instruction increases code size and degrades performance, especially in critical innerloops. To produce efficient DSP code, autoincrement (decrement) addressing modes must be carefully used, so as to minimize the need for update instructions. Needless to say, such problems are faced not only in DSPs architectures, but also in many highly constrained *application specific instruction-set processors* (ASIPs).

Offset assignment (OA) is the optimization that tries to minimize the number of *update instructions* in a program, by making use of autoincrement (decrement) to access local scalar variables. A solution to an OA problem seeks to find a stack layout for these variables such that fewer update instructions are required. An OA problem is called *simple offset assignment* (SOA), when a single address register is present, and *general offset assignment* (GOA), when more than one address register is available in the processor.

This paper describes the *coalescing SOA* (CSOA) algorithm [Ottoni et al. 2003] and a new heuristic for the GOA problem, called *coalescing general offset assignment* (CGOA). Moreover, it presents a thorough comparison between these techniques and the main methods in the literature to solve the OA problem.

The proposed CSOA and CGOA algorithms are based on a new approach to the OA optimization problem, which uses liveness information [Aho et al. 1986; Muchnick 1997] to coalesce variable memory slots while solving OA. This new approach was independently proposed by Zhuang et al. [2003] and by Ottoni et al. [2003]. The interference graph [Muchnick 1997] is used to identify which pairs of variables can be coalesced. During the CSOA optimization, only variables that do not interfere[2] are considered for coalescing. The experimental results show that variable coalescing can produce a large improvement in code quality (61.8% fewer update instructions) when comparing CSOA to the

[1]Generally, autoincrement (decrement) values are one, but in some architectures these values can be larger sometimes.
[2]Two variables interfere if they are simultaneously live at any program point.

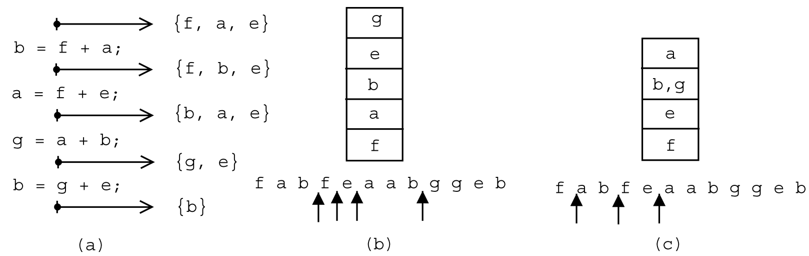


Fig. 1. (a) A fragment of C code. (b) Memory layout with one slot per variable. (c) Memory layout with more than one variable per slot.

previously best algorithm from OffsetStone [Leupers 2003] and (59.5% fewer update instructions) when comparing CGOA to the GOA heuristic proposed by Leupers and Marwedel [1996], for two address registers. Moreover, CSOA and CGOA (for two address registers), respectively, produce 15.5 and 9.3% fewer update instructions when comparing them to the heuristics proposed in Zhuang et al. [2003]. These results show that CSOA and CGOA are the most efficient algorithms for the OA problem and dismisses the first conjectures to this problem [Liao 1996], which seemed to indicate that variable coalescing could lead to worse offset assignment costs.

The rest of this paper is organized as follows. Section 2 describes an example that shows how the number of required update instructions can be reduced by adopting coalescing. Section 3 lists the previous work on OA. The CSOA heuristic is described in Section 4 and Section 5 presents the new CGOA technique. In addition, Section 6 compares CSOA and CGOA with previous heuristics. Finally, Section 7 summarizes the main results of this work.

2. MOTIVATION

This section shows an example that illustrates how coalescing variables can decrease the number of update instructions. For this example, consider that only a single address register is available in the processor, which can be autoincremented (decremented) only by one.

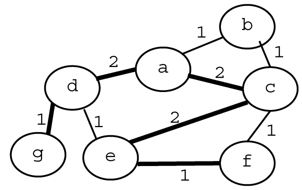
To illustrate coalescing, Figure 1a shows a fragment of C code, annotated with liveness information at each program point. Figures 1b and c contain two possible memory layouts for the program variables and the sequence in which variables are accessed in memory at run-time. The arrows in Figures 1b and c show points where explicit address calculation instructions (i.e., update instruction) are required between two consecutive variable accesses. Update instructions are used to redirect the address register from the first to the second variable, whenever the memory distance between these variables is larger than one. In Figure 1c, the memory layout has one slot that is shared between two variables (b and g) that do not interfere. By sharing a single memory slot for these variables, one less update instruction is required in the program. Clearly, variable coalescing increases the proximity between variables in memory, thus reducing the number of update instructions.

```
b = a + 65000;
c = b << 4;
f = a + c;
e = f << 4;
d = c - e;
g = a + d;
```

(a)

abbcacffecedadg

(b)



(c)

Fig. 2. (a) A fragment of C code. (b) The access sequence of this fragment. (c) The corresponding access graph.

3. RELATED WORK

The OA problem was first studied by Bartley [1992]. In that paper, Bartley proposed an approach to SOA problem, in which only one address register was available in the architecture. Bartley's technique is based on finding a *maximum-weight path cover* (MWPC). Later, Liao et al. [1996] showed that the graph problem MWPC (known to be NP-Complete) can be reduced to SOA, thus proving that SOA is NP-Hard. In addition, Liao et al. extended SOA to the GOA problem, a variation of SOA for more than one address register. A large number of heuristics to solve SOA and GOA have followed [Leupers and Marwedel 1996; Leupers and David 1998; Rao and Pande 1999; Atri et al. 2001; Udayanarayanan and Chakrabarti 2001; Zhuang et al. 2003; Ottoni et al. 2003; Li and Gupta 2003], making it one of the most studied problems in code generation for DSPs and embedded processors, in general.

Liao et al. [1996] solution to SOA uses a heuristic based on the *Kruskal minimum spanning tree* algorithm [West 2001]. Liao et al. [1996] call *access sequence* the sequence, in a basic block, used by a program to access variables at run-time. For example, the access sequence for instruction `a = b` *op* `c` is `bca`. By using the concept of access sequence, Liao et al. define a weighted graph $G(V, E)$ (*access graph*), where V is the set of basic block variables and E an edge set. Each edge $e = (u, v) \in E$ is labeled with a weight $w(e)$, meaning that the access sequence has $w(e)$ consecutive accesses from variable u to v (or v to u). If two variables u and v are never accessed consecutively, then $(u, v) \notin E$. In its global form, access graphs can cross basic blocks boundaries. In this case, the relevant edge weights should reflect the profiling information associated to the blocks execution time. After constructing the access graph, Liao et al. tried to find a set of maximum weighted paths, called *assignment*, which define the variables' layout in memory. The *cost* of an assignment to memory is the addition of the weights of all edges that connect variables in nonadjacent memory positions, as only autoincrement (decrement) by one is available.

Consider, for example, the C code fragment of Figure 2a, and its corresponding access sequence (Figure 2b) and access graph (Figure 2c). Liao et al.'s heuristic is a greedy algorithm that, at each step, selects the edge with the greatest weight. It takes care not to choose an edge that can lead to a vertex with degree greater than 2, and not an edge that forms a cycle with the previously selected edges. The assignment resulting from the access graph of Figure 2c is `fecadgb` (highlighted), and has an offset cost of four. In other words, four update instructions (corresponding to the nonhighlighted edges), are required by the program when that memory layout is used.

An extension to Liao et al.'s heuristic was proposed by [Leupers and Marwedel 1996]. It is called tie-break and its goal is to decide which edge to choose when edges with the same weight are available in the access graph. Leupers and David [1998] proposed a genetic algorithm to solve SOA. Instead of using the access sequence, they compute the offset assignment directly by simulating an evolutionary process. Later, Rao and Pande [1999] described a technique that considers the order of the accesses. This technique uses algebraic transformations in the expression tree to optimize the access sequence.

Sudarsanam et al. [1997] used graph-coloring techniques to coalesce variables before SOA. Their goal was to reduce memory utilization, but they have not shown that coalescing can improve the offset cost. Section 6 presents experimental results of Sudarsanam et al.'s heuristic and compares them with CSOA.

Zhuang et al. [2003], like us in [Ottoni et al. 2003], also uses liveness information in their technique to solve SOA. First, Zhuang et al. coalesce variables, and then use path covering to solve the SOA problem. This method differs from our CSOA algorithm in the way that we perform coalescing simultaneously while choosing the best path covering. Section 6 shows that CSOA produces, on average, 15.5% fewer update instructions when comparing to the results of SOA–Zhuang. Li and Gupta [2003] proposed a new approach to SOA that tries to assign to the same memory slot several variables that are smaller than one memory word.

The GOA problem is NP-Complete, so many heuristics have been proposed to solve it. Most of these use the same approach to solve GOA: break the GOA problem into n SOA problems, with n address registers available, and solve each SOA problem with the already known techniques. The GOA solution will be the concatenation of all SOA solutions. The breaking, called partitioning, is made by assigning each variable to an address register. All variables assigned to the same AR will form a SOA problem. This type of solution was introduced by Liao et al. [1996] and then used in many other works [Leupers and Marwedel 1996; Zhuang et al. 2003].

Zhuang et al. [2003] proposed a GOA heuristic that first tries to partition the variables by coloring them using register coloring [Muchnick 1997]. If more then $2k$ colors are needed, where k is the number of ARs, then a heuristic similar to the partitioning described in Leupers and Marwedel [1996] is used instead of register coloring. This partitioning heuristic uses SOA–Zhuang [Zhuang et al. 2003] to decide in which partition to put each variable. After partitioning, the SOA–Zhuang algorithm is applied to each resulting partition.

Many generalizations of the OA problem have been proposed. Some important generalization are: the use of modify registers [Leupers and Marwedel 1996; Leupers and David 1998], autoincrement (decrement) ranges [Sudarsanam et al. 1997], instruction scheduling coupled with offset assignment [Choi and Kim 2002] and procedure-level offset assignment [Eckstein and Krall 1999].

Another problem related to OA is the problem known as *array reference allocation* (ARA), which optimizes the access to array variables instead of scalar variables. This problem was originally studied by Araujo et al. [1996], and later extended by Leupers et al. [1998], Ottoni et al. [2001], Cintra and Araujo [2000], and Ottoni and Araujo [2003].

Finally, another important problem related to OA is *global register allocation for general-purpose registers*. This problem consists of determining which values will reside in general-purpose registers and which register will hold each of those values, considering multiple basic blocks. Most global allocators use the graph-coloring paradigm [Chaitin 1982; Briggs 1992; George and Appel 1996].

4. COALESCING SIMPLE OFFSET ASSIGNMENT

This section describes an approach that uses liveness information to optimize the solution to SOA. Our approach, called *coalescing simple offset assignment (CSOA)*, takes as input the access sequence and the variables' interference graph, and outputs an offset assignment for the variables in memory. This technique improves all the previous heuristics that solve SOA [Bartley 1992; Liao et al. 1996; Leupers and Marwedel 1996; Atri et al. 2001; Leupers 2003; Zhuang et al. 2003].

Although the use of coalescing can be applied to most previous SOA algorithms, we describe and evaluate a CSOA algorithm based on the SOA algorithm proposed by Liao et al. [1996], with the tie-break heuristic in Leupers and Marwedel [1996] to decide between edges with the same weight. Liao et al. try to form a maximum path in the access graph, sorting the edges of the access graph in decreasing order of their weights. After that, their algorithm iterates until all vertices are inserted onto the path or no other edge is available. At each iteration, Liao et al. choose the *valid edge* (i.e., one not already selected, which does not cause a cycle, and does not increase the degree of a vertex on the path to more than two) with maximum weight.

The pseudocode for CSOA is presented in Algorithm 1. Instead of always choosing an edge at each iteration, as in typical SOA solutions, Algorithm 1 considers another alternative: coalescing two vertices. Specifically, it chooses one of the two operations: (a) coalesce two vertices u and v in the access and interference graphs, if they do not interfere; (b) select a valid edge of maximum weight from the sorted list of edges (L in the Algorithm 1), as in Liao et al.'s approach.

Function *FindCandidatePair*, in Algorithm 1, tries to find the two candidate vertices for coalescing. It returns a quadruple $(coal, u, v, csave)$, where $coal$ is a flag that is set if there are two vertices u and v for coalescing, and $csave$ is the number of update instructions that are saved if u and v are coalesced.

In order to find the two candidates for coalescing, function *FindCandidatePair* (line 7) searches, in the interference graph, among all possible combinations of two vertices u and v, considering only the vertices that satisfy the following conditions:

1. $(u, v) \notin$ the interference graph;

Algorithm 1 Coalescing-Based SOA

```
Input: the access sequence L_AS,
        the interference graph G_I(V_I, E_I).
Output: the offset assignment.
G_A(V_A, E_A) ← BuildAccessGraph(L_AS);
L ← sorted list of the E_A;
coal ← false;
sel ← false;
repeat
        rebuild ← false;
        (coal, u, v, csave) ← FindCandidatePair(G_I, u, v);
        sel ← FindValidEdgeNotSel(L, e);
        if (coal && sel )
              if (csave ≥ w(e))
                    rebuild ← true;
              else
                    mark e as selected;
        else
              if (coal)
                    rebuild ← true;
              else if (sel)
                    mark e as selected;
        if (rebuild)
              RebuildAccessGraph(G_A, u, v);
              RebuildInterferenceGraph(G_I, u, v);
              RebuildL(L);
until (!(coal || sel))
returnBuildOffset(G_A);
```

2. Coalescing u and v creates no cycle in the access graph, when considering only the selected edges;
3. Coalescing u and v, in the access graph, does not result in a coalesced vertex with degree greater than two, when only the selected edges are considered.

These three conditions can be tested very efficiently. While checking conditions (1) and (3) is very straightforward, condition (2) is a bit trickier to be implemented efficiently. However, this can be done by using a Union-Find data structure to implement disjoint sets [Cormen et al. 1990], where each set corresponds to a connected component in the subgraph of the current access graph induced by the edges that have already been selected.

Algorithm 1 then picks, among all pairs of vertices that satisfy the above conditions, the pair u and v, whose coalescing produces the highest *csave*.

In order to determine *csave*, function *FindCandidatePair* computes the statements below, where $Adj_{sel}(y)$ is the set of vertices adjacent to y in the access graph. Here again, only the already selected edges are considered:

1. $\forall x \in (Adj_{sel}(u) - Adj_{sel}(v))$, add $w(x, v)$ to *csave*;
2. $\forall x \in (Adj_{sel}(v) - Adj_{sel}(u))$, add $w(x, u)$ to *csave*;
3. Add the weight of the edge between u and v, $w(u, v)$, to *csave*, if this edge has not been selected yet.

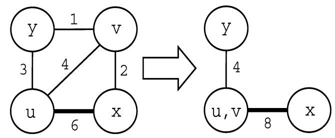


Fig. 3. One access graph and the resulting access graph after coalescing variables u and v.

To illustrate how the above statements work, consider the example in Figure 3. According to statements (1–3) and Figure 3, the value of *csave*, when u and v are coalesced, is the weight of edge (x, v) (since x is adjacent to u, edge (x, u) is selected, and edge (x, v) has not been selected) plus the weight of the nonselected edge (u, v). The value of *csave* becomes 6, 4 from edge (u, v), and 2 from edge (x, v).

The computation proceeds to line (8) of the Algorithm 1, where function *FindValidEdgeNotSel* searches for the valid edge e with maximum weight $w(e)$ in the sorted list of edges (L). Flag *sel* is set if e is found.

Eventually, if both *coal* and *sel* are true (line 9), the algorithm chooses (line 10) the operation (coalescing versus edge selection) that produces the largest reduction in the number of update instructions.

Whenever two vertices u and v are coalesced, parts of the access and interference graphs need to be rebuilt to reflect the operation. This is achieved in lines (19–22) of Algorithm 1. Hence, all the old adjacencies of u and v, in the old access graph, must be redirected to the coalesced vertex (uv) in the new graph. In the new interference graph, the coalesced vertex must interfere with all vertices that were adjacent to either u or v in the old interference graph.

Function *RebuildL*, line (22) of Algorithm 1, is used to reconstruct the sorted list of edges (i.e., L) from the new access graph. Algorithm 1 ends when there are no more valid edges that can be chosen and no more vertices to coalesce. This condition is tested by using flags *sel* and *coal* in line (23) of Algorithm 1.

4.1 Complexity Analysis of CSOA

In this section, the worst-case time-complexity of CSOA is analyzed. Let m be the length of the access sequence and n the number of variables considered for CSOA. In Algorithm 1, the complexity of *BuildAccessGraph* is $O(m + n^2)$. The sorting operation in line (2) takes $O(n^2 \log n)$. After that, the *repeat-until* loop can be executed at most $2(n - 1)$ times (at most $n - 1$ edges are selected and $n - 1$ coalescing operations are performed). The *repeat-until* loop is dominated by the *RebuildL* function, which is $O(n^2 \log n)$. Thus, this loop has complexity $O(n^3 \log n)$. Finally, the *BuildOffset* function takes $O(n^2)$ time. Therefore, CSOA has time-complexity $O(m + n^3 \log n)$. Notice that this is a worst-case analysis. In practice, the CSOA run-time is considerably better.

4.2 Example of Coalescing SOA

In order to illustrate how CSOA works, consider the code fragment of Figure 4a. In that code, each program point has the set of live variables (assuming that

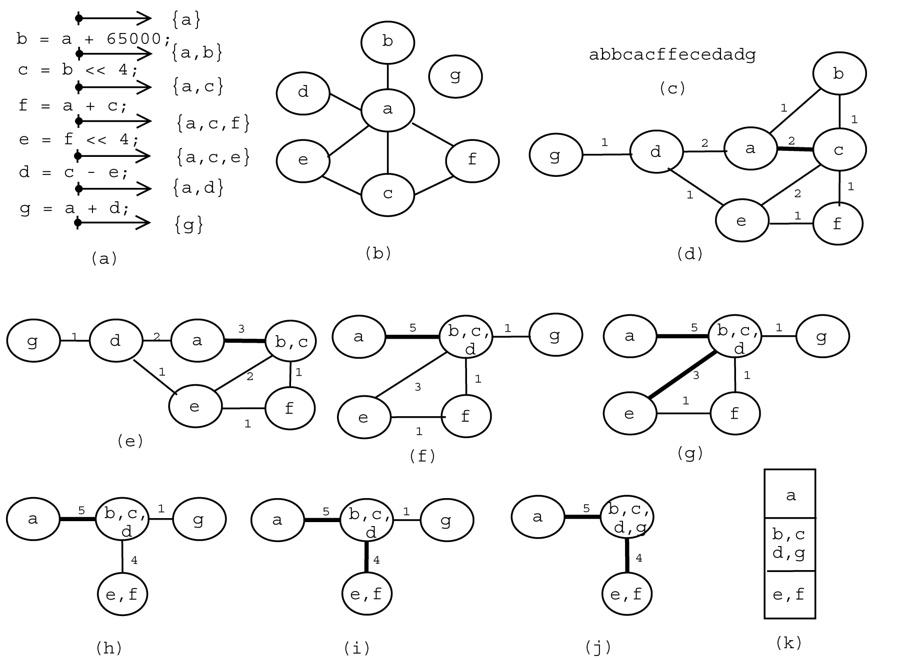


Fig. 4. (a) A fragment of C code with liveness information at each point. (b) The interference graph of the variables. (c) The access sequence of this fragment. (d–j) The access graphs resulting after each iteration of the algorithm. (k) The resulting memory layout. Selected edges are highlighted.

only g is live at the exit of the fragment). When Algorithm 1 is applied to this example, it takes as input the interference graph shown in Figure 4b and the access sequence in Figure 4c.

As the algorithm runs, it produces, at each iteration, the access graphs shown in Figures 4d–j. At each step, the edges selected during the assignments are highlighted. The final memory assignment is defined by the selected paths on the last access graph. The resulting memory layout is shown in Figure 4k. The reader should remember that whenever two vertices in the access graph are coalesced, these vertices are also coalesced in the interference graph (not shown in the figures).

Running CSOA in the example code of Figure 4a will result in the following execution. In the first iteration (Figure 4d), edge (a, c) is selected, as no pair of vertices can be coalesced to produce savings as high as 2. In the next iteration, the best choice is to coalesce vertices b and c, given that it results in a saving of 2 (corresponding to the edges (a, b) and (b, c)). The new vertex (bc) becomes adjacent to the vertices that were adjacent to b or c in the previous access graph, i.e., a, e, and f. Notice that the weight of the edge between a and (bc) becomes 3, which is the sum of the weights of edges (a, b) and (a, c) in the previous graph. In the next steps, the algorithm has to decide between coalescing two vertices or selecting an edge, until no more operations are possible. The resulting covered access graph is shown in Figure 4j.

As all edges in the final access graph are selected, the final cost of applying CSOA to the example is zero. Notice that this example is the same as the one in Section 3, for which Liao et al.'s algorithm produces a final cost of four.

5. COALESCING GENERAL OFFSET ASSIGNMENT

The technique described in this section is a solution to the GOA problem that was designed to be used with CSOA (described in Section 4). CGOA, like most methods for GOA, first partitions the set of variables, and then applies CSOA to each partition. The main objective of the partitioning of CGOA is to increase the opportunities of variable coalescing for the CSOA method. The partitioning made by CGOA is based on variables' liveness information. It tries to put each variable v in the partition that has fewer variables that interfere with v. This partitioning is greedy and, at each step, it chooses the partition in which one variable will be inserted.

The algorithm that describes the CGOA partitioning is shown in Algorithm 2. This algorithm receives three inputs: access sequences, interference graph and number of address registers available. As output, it returns the set of variables' partitions. Initially, as shown in line (1) of Algorithm 2, the partition data structures are initialized, based on the number of ARs that are available. After that, in line (2), the number of interferences of each variable is computed, using the interference graph. Following this, as shown in line (3) of Algorithm 2, variables are sorted in decreasing order of their number of interferences. They are then assigned to the appropriate partition using this order, as shown in lines (5–10) of the algorithm. At each partition, Algorithm 2 computes the number of interferences of the variable under consideration with all the variables in the partition. This is shown in line (7) of Algorithm 2. The variable is assigned to the partition for which it has the fewest interferences (line 11 of the algorithm). After all variables have been assigned to a partition, the algorithm returns the set of partitions. There are two important details that are not shown in the algorithm. The first detail not shown is the tie-break criterion, which is used when one variable has the same minimum number of interferences in more than one partition. In such cases, the variable under consideration is assigned to the partition that has fewest variables, among the partitions with which it has the minimum number of interferences. The goal here is to minimize the creation of unbalanced partitions. The second detail (not shown) is that our CGOA algorithm only returns the partitioning if the addition of all SOA costs, resulting from this partitioning, is less than the CSOA cost. Otherwise, CGOA returns CSOA's solution corresponding to a single partition.

After partitioning, the CSOA algorithm is applied to each resulting partition. The solution for the GOA problem is the concatenation of the CSOA solutions of all partitions.

5.1 Complexity Analysis of CGOA

In this section, the worst-case time complexity analysis of Algorithm 2 is presented. Let *nars* denote the number of ARs available, m the length of the access sequence, and n the number of variables. First, the complexity analysis

Algorithm 2 CGOA Partitioning Algorithm

Input: L_{AS} access sequence,
 $G_I(V_I, E_I)$ interference graph,
 NARs number of address registers.
Output: *PartitionsSet* set of disjoint subsets of all variables.

(1) *PartitionsSet* ← InitSetofPartitions(*NARs*);
(2) *IgDegree* ← CalculateIgDegree($G_I(V_I, E_I)$);
(3) *VarsOrdByDegree* ← FillVarsReverseOrdedByDegree(*IgDegree*);
(4) for each $v \in V$, according to the *VarsOrdByDegree* order, do
(5) *MinNInt* ← *NVARs* + 1;
(6) for each *Partition* ∈ *PartitionsSet* do
(7)   *NInt* ← CalculateNInterferences(*VarsOrdByDegree*, *Partition*);
(8)   if *NInt* < *MinNInt* then
(9)   *MinNInt* ← *NInt*;
(10)   *NPartMinNInt* ← *Partition*;
(11) PutVarInPartition(*VarsOrdByDegree*, *NPartMinNInt*, *PartitionsSet*);
(12) return *PartitionsSet*;

of Algorithm 2 is performed. Function *InitSetofPartitions*, line (1) of Algorithm 2, takes $O(nars)$ time. Function *CalculateIGDegree*, line (2), traverses the interference graph in order to count the number of interferences of each variable. The resulting complexity for this part is, thus, $O(n^2)$. In line (3) of Algorithm 2, function *FillVarsReverseOrderedByDegree* time complexity is $O(n \log n)$, since it has to sort the variables in decreasing order of their number of interferences. Finally, the loop shown in lines (4–11) has time complexity $O(n\,(n + nars))$, given that function *CalculateNInterferences*, which is $O(n)$, is invoked inside this loop (line 7). Function *PutVarInPartition*, line (11), takes constant time. Thus, Algorithm 2 complexity is $O(n\,(n + nars))$.

After partitioning, CGOA executes CSOA for each partition. This part of the CGOA algorithm is $O(nars\,(m + n^3\, log\, n))$, since it runs the *nars* times CSOA algorithm, and has complexity $O(m + n^3 \log n)$. Therefore, the CGOA algorithm is $O(nars\,(m + n^3 \log n))$.

As before, this is a worst-case time complexity, and, in practice, this algorithm runs more efficiently.

5.2 Example of CGOA Partitioning

The example shown in Figure 5 illustrates how CGOA partitioning works. Figure 5a has an interference graph, and Figure 5b shows the variables sorted in decreasing order of their number of interferences. In this example, there are only two ARs. Figures 5c–h show the interference graphs of each partition while partitions are constructed. Figure 5i shows the interference graph for each partition after the partitioning algorithm is finished. To build the partitioning, variables are considered in the order shown in Figure 5b. Initially, variable 9 is arbitrarily placed into partition P0, since variable 9 has the same number of interferences with the variables in partitions P0 and P1 (Figure 5c). Variable 6 is placed into partition P1 (Figure 5d), because it has one interference with variables in P0 and none in P1. Variables are being placed into the

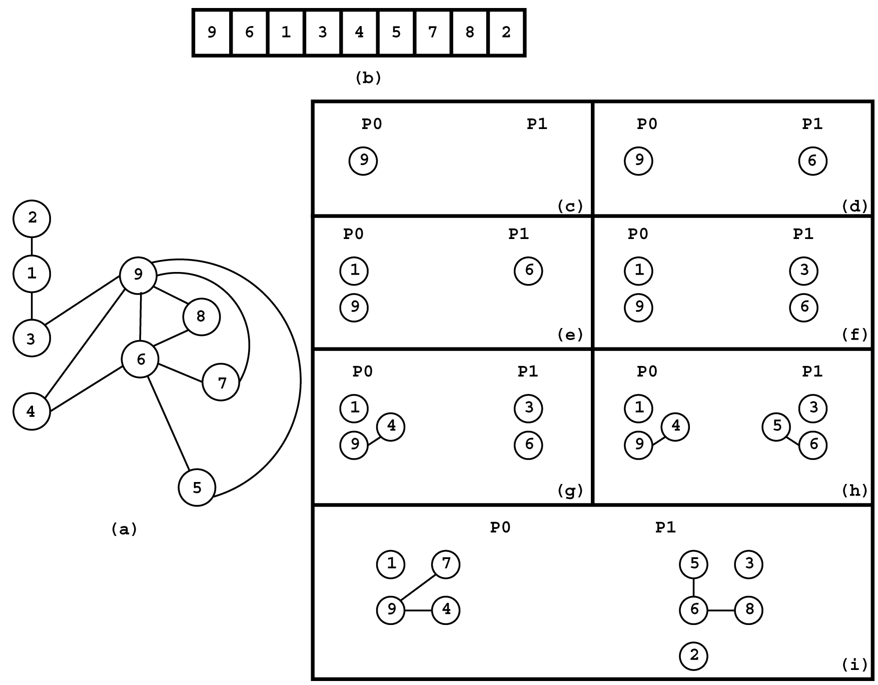


Fig. 5. (a) Interference graph; (b) variables sorted in decreasing order of their number of interferences; (c–h) interference graph of each partition during the process; (i) interference graphs resulting for each partition.

partition with fewer interferences, obeying the order shown in Figure 5b. The algorithm ends when each variable is assigned to a partition. In Figure 5h, the tie-break criterion is used, and variable 5 is placed into partition P1, as P1 has fewer variables than P0, and the number of interferences between variable 5 and variables in P0 and P1 is the same. Figure 5i shows the resulting partitions.

6. EXPERIMENTAL RESULTS

In this section, we compare CSOA to six other solutions to SOA, and CGOA to three other approaches to GOA. We use the MediaBench benchmark [Lee et al. 1997] to evaluate the heuristics.

The techniques proposed in this paper were implemented in OffsetStone [Leupers 2003], a toolset used to test and evaluate OA algorithms. Benchmark programs were compiled using the LANCE [Leupers 2001] compiler front-end, which translates the C source code into three-address code intermediate representation. The resulting intermediate representation code was then optimized through a combination of the following optimizations: constant folding, constant propagation, jump optimization, loop-invariant code motion, induction variable, elimination, global common subexpression elimination, dead code elimination and copy propagation. Notice that memory aliasing is not an issue for OA, as only scalar, local variables are rearranged on the stack. Because the

Table I. Offset Cost Relative to SOA–OFU

Benchmarks	Liao (%)	TB (%)	GA (%)	INC-TB(%)	CSOA (%)
adpcm	66.7	63.8	63.8	63.8	30.0
epic	72.3	70.0	69.8	69.8	36.3
g721	73.3	70.5	70.5	70.5	20.2
gsm	72.2	69.6	69.6	69.6	13.4
jpeg	68.8	66.7	66.5	66.5	21.9
mpeg2	71.5	69.6	69.4	69.5	24.5
pegwit	68.3	62.2	61.9	61.9	26.0
pgp	70.8	67.2	67.1	67.1	22.5
rasta	65.7	64.7	64.7	64.7	13.9
Average	69.9	67.1	67.0	67.0	25.6

programmer cannot assume any ordering among these variables on the stack, the compiler has freedom to perform OA safely.

Access sequences were then extracted from each basic block and basic block access graphs merged on a function basis. Variable live ranges were computed by doing liveness analysis [Aho et al. 1986] in the intermediate representation (after the optimizations described above) and the interference graph [Muchnick 1997] was then constructed.

CSOA is compared (Table I) to four other approaches that do not use variable coalescing. To evaluate the results, we measured the percentage of the number of update instructions inserted by each method. The baseline metric is the number of update instructions inserted by SOA–OFU, a trivial offset assignment algorithm where variables are assigned to offsets in the order of their first use in the code. The four other methods used in this comparison are: SOA-TB, the heuristic described in Leupers and Marwedel [1996]; SOA–GA, the heuristic described in Leupers and David [1998]; SOA–INC-TB [Leupers 2003], the combination of two SOA algorithms, SOA-incremental Atri et al. [2001], SOA–TB [Leupers and Marwedel 1996]; and SOA–Liao, the algorithm described in Liao et al. [1996].

Notice from Table I that CSOA reduces, on average, the number of update instructions to 25.6% of the SOA–OFU cost. This is a very significant improvement over the previous algorithms. The best of the other algorithms (SOA–INC–TB and SOA–GA) reduced the offset cost, on average, to 67.0% of the SOA–OFU cost.

Table II shows the numbers of update instructions necessary when SOA–OFU is used.

Table III shows the results of CSOA when compared to the results of two other methods that also use variable coalescing. As in Table I, we measured the percentage of the number of update instructions inserted by each method, with respect to the number of update instructions inserted by SOA–OFU. The other two heuristics are: SOA–Color, the optimization described in Sudarsanam et al. [1997], and SOA–Zhuang, the algorithm described in Zhuang et al. [2003]. The table shows that, on average, CSOA reduces the number of update instructions to 25.6%, while SOA–Zhuang reduces to 30.3% and SOA–Color to 35.8% of the SOA–OFU cost. In other words, CSOA produces 15.5% fewer update

Table II. Offset Cost Returned by SOA–OFU

Benchmarks	SOA–OFU
adpcm	207
epic	6235
g721	718
gsm	1511
jpeg	10338
mpeg2	7981
pegwit	2249
pgp	7235
rasta	6626

Table III. Offset Cost Relative to SOA–OFU

Benchmarks	CSOA(%)	SOA–Color (%)	SOA–Zhuang (%)
adpcm	30.0	37.2	31.9
epic	36.3	53.7	39.9
g721	20.2	37.1	22.1
gsm	13.4	19.2	14.6
jpeg	21.9	36.2	26.6
mpeg2	24.5	43.1	26.8
pegwit	26.0	51.3	30.3
pgp	22.5	39.0	26.3
rasta	13.9	21.8	24.7
Average	25.6	35.8	30.3

instructions than SOA–Zhuang and 28.5% fewer update instructions than SOA–Color. We believe that this exceptional improvement is because of the fact that CSOA does not coalesce variables aggressively and indiscriminately, but tries to make adjacent in memory variables that have many consecutive accesses. This increases the closeness between variables that are accessed consecutively. CSOA, in opposition to other techniques that naively coalesce variable slots [Sudarsanam et al. 1997], wisely takes advantage of coalescing to reduce both the SOA cost and the memory requirement. This is achieved by simultaneously performing variable coalescing while solving SOA.

Another important result is the memory (stack) savings, when variable coalescing is used combined with the OA techniques. These results can be observed in Table IV. The results shown in Table IV are the percentages of memory size used by SOA–Color, CSOA, and SOA–Zhuang when compared to the memory size used by all other methods that do not use variable coalescing [Liao et al. 1996; Leupers and Marwedel 1996; Leupers and David 1998; Leupers 2003; Atri et al. 2001], and considering that only local statically allocated variables are in the stack.

From Table IV, one can observe that CSOA reduces the size of the stack used to store local variables to 28.6%, when compared to the other methods that do not perform variable coalescing, while SOA–Color reduces to 11.6 and SOA–Zhuang to 24. It is important to emphasize that, although CSOA makes fewer variable coalescing, the OA cost resulting from CSOA is the smallest cost,

Table IV. Stack Length Relative to Stack Length of the Methods Not Performing Variable Coalescing

Benchmarks	CSOA (%)	SOA–Color (%)	SOA–Zhuang (%)
adpcm	27.3	15.7	28.3
epic	27.0	11.6	26.6
g721	25.0	13.3	22.7
gsm	21.5	7.0	19.8
jpeg	34.7	14.6	25.7
mpeg2	31.8	12.2	21.9
pegwit	35.3	9.7	26.8
pgp	31.5	12.8	24.7
rasta	26.1	9.9	21.4
Average	28.6	11.6	24.0

Table V. Percentage of Temporary Variables, Considering as Temporary a Variable Alive in Only One Basic Block

Benchmarks	%Temporaries
adpcm	59.6
epic	48.1
g721	80.7
gsm	86.6
jpeg	65.2
mpeg2	65.6
pegwit	72.1
pgp	67.5
rasta	43.6
Average	64.1

compared with SOA–color and SOA–Zhuang. In addition, by achieving a better OA, CSOA reduces the code size more than the other techniques.

Finally, Table V shows the number of temporary variables (among those considered for SOA) in each program.[3] Observe through these numbers that, on average, 64.1% of the variables are temporaries. Memory-stored temporaries are very common in DSP and other ASIP architectures, given their reduced number of general-purpose registers. Thus, temporary allocation plays an important role in the final code performance, reinforcing our perception that there are many opportunities for CSOA to coalesce variables in DSP code, as shown by the experimental results.

Yet another experimental result revealed that in 43.9% of the instances of all benchmarks, CSOA resulted in zero cost. Thus, at least for this percentage of the instances, CSOA resulted in the optimal cost. We also believe that this percentage is significantly higher, in fact, as many of the other instances may have an optimal cost greater than zero.

Table VI shows the results obtained by the following methods:

- CGOA, the technique proposed in this paper, and described in Section 5;

[3]We consider here as temporaries those variables whose liveness are restricted to a single basic block.

Table VI. Offset Cost Relative to SOA–OFU Cost When Using Two ARs

Benchmarks	CGOA (%)	CGOA–Liao (%)	GOA–Leupers (%)
adpcm	7.3	44.4	23.2
epic	22.8	51.3	30.3
g721	6.8	47.1	27.3
gsm	3.0	32.4	36.0
jpeg	12.5	46.6	23.5
mpeg2	15.2	52.0	28.9
pegwit	10.8	41.8	19.1
pgp	11.4	44.9	25.2
rasta	10.4	54.1	12.0
Average	11.3	52.8	27.9

Table VII. Offset Cost Relative to SOA–OFU Cost When Using Four ARs

Benchmarks	CGOA (%)	CGOA–Liao (%)	GOA–Leupers (%)
adpcm	4.4	25.1	8.2
epic	10.5	31.5	8.7
g721	7.1	30.6	14.6
gsm	4.4	27.0	36.7
jpeg	9.2	34.5	15.2
mpeg2	10.1	38.2	13.7
pegwit	5.7	24.2	12.9
pgp	7.8	31.4	14.3
rasta	9.2	46.2	5.4
Average	8.4	36.4	14.6

- CGOA-Liao, a hybrid technique that performs our variable partition described in Section 5 and, after that, performs SOA–Liao [Liao et al. 1996] in each partition;
- GOA-Leupers, which is the method described in Leupers and Marwedel [1996].

Only two address registers were used to obtain the results of Table VI. The results are the percentages of the number of update instructions inserted by each method, with respect to the number of update instructions inserted by SOA–OFU. Observe that, on average, for only two ARs, CGOA reduces the offset cost to 11.3%. CGOA–Liao and GOA–Leupers reduce the offset cost to, respectively, 52.8 and 27.9%.

Table VII shows the CGOA, CGOA–Liao and GOA–Leupers results when four address register are available. Again, the percentages are the offset cost for each method when compared to the SOA–OFU offset cost. The CGOA heuristic reduces the cost to 8.4%, CGOA-Liao to 36.4%, and GOA–Leupers to 14.6%. Again, CGOA produces the best results.

Table VIII shows the results of CGOA, CGOA–Liao, and GOA–Leupers, for eight address registers. Like in the previous tables, the percentages are the offset cost of each method compared to SOA–OFU cost. For all three methods, the results obtained when eight ARs are used were worse than the results achieved

Table VIII. Offset Cost Relative to SOA–OFU Cost When Using Eight ARs

Benchmarks	CGOA (%)	CGOA–Liao (%)	GOA–Leupers(%)
adpcm	10.1	23.7	16.9
epic	5.5	19.9	5.8
g721	13.1	29.7	27.3
gsm	7.1	45.3	55.7
jpeg	9.3	38.8	27.8
mpeg2	9.2	34.5	17.9
pegwit	9.4	31.5	26.6
pgp	8.2	34.4	24.0
rasta	9.4	39.7	6.8
Average	10.2	37.2	22.0

Table IX. Stack Size Result CGOA Relative to the Stack Size Resulting from the Methods that Do Not Performing Variable Coalescing

Benchmarks	2 ARs (%)	4 ARS (%)	8 ARS (%)
adpcm	28.8	44.5	55.6
epic	32.0	37.3	48.3
g721	30.6	42.7	57.3
gsm	27.3	36.7	46.2
jpeg	43.0	55.4	68.5
mpeg2	41.8	52.3	62.5
pegwit	42.5	54.5	65.7
pgp	53.2	53.4	68.5
rasta	23.1	47.8	56.1
Average	35.5	46.7	58.2

when four ARS are available. This is probably because of the cost to initialize four other ARs, which considerably impacts the final code quality. Besides that, in CGOA, more partitions decrease the variable coalescing opportunities, because only variables in the same partition can be coalesced. CGOA reduces the offset cost to 10.2%, CGOA–Liao to 37.2%, and GOA–Leupers technique to 22%.

Table IX shows the CGOA stack savings when two, four, and eight ARs are used. The results are in percentage of the stack size when compared to the methods that do not perform variable coalescing. CGOA reduces the stack size, on average, to 35.5, 46.7, and 58.2%, for two, four and eight ARs, respectively. These numbers confirm that more partitions imply fewer opportunities for variable coalescing, since less variables are assigned to each partition. Obviously, there is an ideal number of partitions, that should be between three and seven for the Mediabench benchmark. A simple technique to find the real number of ARs for each instance is to test all values between 1 and the total number of ARs in the processor.

Tables XI, XII, and XIII show the results obtained by CGOA and GOA–Zhuang [Zhuang et al. 2003], using, respectively, two, four, and eight ARs. Because of the high-time complexity of GOA–Zhuang and the large number of variables generated by the LANCE compiler [Leupers 2001] to some instances of Mediabench programs, we ran the GOA–Zhuang and CGOA algorithms only

Table X. Offset Cost Returned by SOA–OFU with Restriction on Number of Variables per Instance

Benchmarks	SOA–OFU
adpcm	207
epic	1560
g721	588
gsm	1105
jpeg	9247
mpeg2	5024
pegwit	1410
pgp	6160
rasta	1359

Table XI. Offset Cost Relative to SOA–OFU Cost when Two ARs Are Available

Benchmarks	CGOA (%)	GOA–Zhuang (%)
adpcm	7.3	8.7
epic	15.5	12.4
g721	6.8	8.5
gsm	3.4	9.4
jpeg	11.6	12.7
mpeg2	13.9	10.7
pegwit	9.7	10.3
pgp	10.7	10.8
rasta	12.8	12.3
Average	6.8	7.5

Table XII. Offset Cost Relative to SOA–OFU Cost When Four ARs Are Available

Benchmarks	CGOA (%)	GOA–Zhuang (%)
adpcm	4.3	12.1
epic	8.6	17.7
g721	8.2	17.5
gsm	5.4	25.4
jpeg	8.1	20.4
mpeg2	8.9	17.0
pegwit	8.0	22.1
pgp	7.1	18.7
rasta	8.3	19.4
Average	7.3	18.6

to the instances with less than 270 variables. It is important to emphasize that this is a limitation of GOA–Zhuang algorithm only and not of CGOA.

Table X shows the offset cost returned by SOA–OFU, considering only the instances with less than 270 variables.

Table XI shows that CGOA, on average, reduces the offset cost to 6.8% and GOA–Zhuang to 7.5%, using as baseline the SOA–OFU cost. In general, we want to stress that CGOA has many advantages, when comparing to GOA–Zhuang, namely: (1) CGOA produces a better offset cost than GOA–Zhuang; (2)

Table XIII. Offset Cost Relative to SOA–OFU When Eight ARs Are Available

Benchmarks	CGOA (%)	GOA–Zhuang (%)
adpcm	10.1	16.9
epic	9.6	26.0
g721	14.8	32.1
gsm	9.0	57.6
jpeg	9.0	33.9
mpeg2	9.3	27.1
pegwit	14.5	43.7
pgp	8.6	30.1
rasta	9.5	30.1
Average	10.3	31.4

it has a much smaller time complexity; and (3) CGOA is a much simpler algorithm when compared to GOA–Zhuang, and, thus, is much easier to implement.

Table XII shows the results obtained by CGOA and GOA–Zhuang, when there are four ARs available. CGOA reduces offset cost to 7.3% and GOA-Zhuang to 18.6%, again comparing to the SOA–OFU cost. The solutions achieved by CGOA and GOA–Zhuang, considering four ARs are worse than considering two ARs, and even worse when considering eight ARs, as illustrated in Table XIII.

7. CONCLUSIONS

In this paper we described the CSOA heuristic, originally presented in Ottoni et al. [2003], and proposed a heuristic to solve the GOA problem based on coalescing memory variable slots. The experimental results show that our method (CGOA), for two address registers, eliminates, on average, 17.2% of the update instructions when compared to GOA–Zhuang, the best previous method to GOA. In addition, CGOA runs faster and is much simpler to implement. Another important side effect of our technique is the reduction in the size of the memory layout when compared to methods that do not perform variable coalescing. The large presence of temporaries in DSP programs and the increased closeness resulting from the coalescing technique justify these good results.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techinques and Tools*. Addison-Wesley, Reading, MA.

ARAUJO, G., SUDARSANAM, A., AND MALIK, S. 1996. Instruction set design and optimizations for address computation in dsp architectures. In *Proc. of the 9th. ACM/IEEE International Symposium on System Synthesis*. 102–107.

ATRI, S., RAMANUJAM, J., AND KANDEMIR, M. 2001. Improving offset assignment for embedded processors. *Lecture Notes in Computer Science 2017*.

BARTLEY, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Software—Practice and Experience 22,* 2, 101–110.

BRIGGS, P. 1992. Register allocation via graph coloring. Ph.D. thesis, Rice University.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*.

CHOI, Y. AND KIM, T. 2002. Address assignment combined with scheduling in DSP code generation. In *Proc. of the 39th Design Automation Conference, DAC 2002*.

CINTRA, M. AND ARAUJO, G. 2000. Array reference allocation using ssaform and live range growth. In *Proc. of the ACM SIGPLAN 2000 LCTES*. 26–33.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, Cambridge, MA and New York.

ECKSTEIN, E. AND KRALL, A. 1999. Minimizing cost of local variables access for DSP-processors. In *Proc. of the ACM SIGPLAN 1999 LCTES*.

GEORGE, L. AND APPEL, A. 1996. Iterated register coalescing. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages*. 208–218.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30)*.

LEUPERS, R. 2000. *Code Optimization for Embedded Processors*. Kluwer Academic Publ., Boston, MA.

LEUPERS, R. 2001. Lance: A c compiler platform for embedded processors. In *Embedded Systems/Embedded Intelligence*.

LEUPERS, R. 2003. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proceedings of the 12th International Conference on Compiler Construction*.

LEUPERS, R. AND DAVID, F. 1998. A uniform optimization technique for offset assignment problems. In *Proc. of the International Symposium on System Synthesis (ISSS)*. 3–8.

LEUPERS, R. AND MARWEDEL, P. 1996. Algorithms for address assignment in DSP code generation. In *International Conference on Computer-Aided Design (ICCAD)*. 109–112.

LEUPERS, R., BASU, A., AND MARWEDEL, P. 1998. Optimized array index computation in DSP programs. In *Proc. of the Asia South Pacific Design Automation Conference (ASP-DAC)*. IEEE.

LI, B. AND GUPTA, R. 2003. Simple offset assignment in precense of subword data. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*. ACM Press, New York. 12–23.

LIAO, S. 1996. Code generation and optimization for embedded digital signal processors. Ph.D. thesis, Massachusetts Institute of Technology.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1996. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems 18,* 3 (May), 235–253.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Mateo, CA.

OTTONI, G. AND ARAUJO, G. 2003. Address register allocation for arrays in loops of embedded programs. *Microelectronics Journal 34,* 11 (Oct.), 1009–1018.

OTTONI, G., RIGO, S., ARAUJO, G., RAJAGOPALAN, S., AND MALIK, S. 2001. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of the 10th International Conference on Compiler Construction, CC2001, LNCS 2027*. Springer, New York. 274–288.

OTTONI, D., OTTONI, G., ARAJO, G., AND LEUPERS, R. 2003. Offset assignment through simultaneous variable coalescing. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03)*. Vol. LNCS 2826. Springer Verlag, New York. 285–297.

RAO, A. AND PANDE, S. 1999. Storage assignment optimizations to gerenrate compact and efficient code on embedded dsps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 128–138.

SUDARSANAM, A., LIAO, S., AND DEVADAS, S. 1997. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Design Automation Conference*. 287–292.

UDAYANARAYANAN, S. AND CHAKRABARTI, C. 2001. Address code generation for digital signal processors. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'01)*. 155–164.

WEST, D. B. 2001. *Introduction to Graph Theory*, 2nd ed. Prentice Hall, Englewood Cliffs, NJ.

ZHUANG, X., LAU, C., AND PANDE, S. 2003. Storage assignment optimizations through variable coalescence for embedded processors. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Processors (LCTES'03)*. ACM Press, New York. 220–231.

Received January 2005; revised August 2005 and January 2006; accepted February 2006