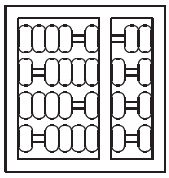


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Improving Offset Assignment through Variable Coalescing

Desiree Ottoni *Guilherme Ottoni*
Guido Araujo *Rainer Leupers*

Technical Report - IC-03-05 - Relatório Técnico

April - 2003 - Abril

Improving Offset Assignment through Variable Coalescing

Desiree Ottoni* Guilherme Ottoni Guido Araujo Rainer Leupers

April 2003

Abstract

Efficient address code optimization is a central problem in code generation for processors with restricted addressing modes, like Digital Signal Processors (DSPs). This paper proposes a new heuristic to solve the Simple Offset Assignment (SOA) problem, the problem of allocating scalar variables to memory so as to minimize addressing code. This new approach, called Coalescing SOA (CSOA), performs variable memory slot coalescing simultaneously to offset assignment computation. Experimental results, based on the MediaBench benchmark, reveal a very significant improvement over all known solutions to SOA. In fact, CSOA produces, on average, 66.5% fewer update instructions, and reduces by 71.3% the stack space for local variables, when comparing with the best known solution to SOA.

1 Introduction

The growth of the DSP market and the increasing demand for new and complex applications running on these processors have brought a strong interest to compilers capable of generating efficient DSP code. However, as DSPs have very irregular architectures, traditional compiling techniques designed for general-purpose processors [1, 17] are not capable of generating efficient code for DSPs [10]. As a result, new techniques tailored for these processors have been proposed and intensively studied. Due to their instruction size and performance constraints, DSPs traditionally have no offset addressing mode, containing only indirect addressing, and a few general-purpose registers. In addition, DSPs have specialized *Address Generation Units (AGU)*, that provide address computation in parallel to datapath computation. AGUs perform *auto-increment (decrement)* in address registers (AR) by some fixed values[1]. For different values, the program is required to have an explicit *update instruction* (prior to the memory access) that uses datapath resources to compute the memory address. Therefore, in order to produce efficient code for such DSPs, it is important to use auto-increment (decrement) addressing modes effectively.

The optimization that tries to maximize the use of instructions with auto-increment (decrement) for local scalar variables is called *Offset Assignment* (OA). This optimization

*Research supported by FAPESP (Proc. 01/12762-5).

[1]Generally, the values of auto-increment (decrement) are one, but in some architectures these values can sometimes be larger than one.

finds a stack layout for these variables in such a manner that auto-increment (decrement) addressing modes are used whenever possible. The variation of the OA problem when there is only one address register and auto-increment (decrement) by 1 is called *Simple Offset Assignment (SOA)* [16] and is the focus of this paper.

In this paper we describe a new approach to the SOA optimization problem, called Coalescing SOA (CSOA). It uses liveness information [1, 17] to simultaneously coalesce variable memory slots while solving SOA optimization. The interference graph [17] is used to identify which pairs of variables can be coalesced. Only variables that do not interfere[2] can be coalesced during CSOA. We show that variable coalescing can lead to a large improvement in code quality (66.5% fewer update instructions) when comparing to the best algorithm in OffsetStone [11]. This result dismisses the first assumptions to this problem, as in Liao [15], that seemed to indicate the opposite. Moreover, as a side effect of CSOA, we also show that the proposed algorithm considerably reduces the final stack size to 28.7%, when comparing to other approaches that do not perform coalescing.

The remainder of this paper is organized as follows. Section 2 exhibits an example that illustrates how the use of coalescing can affect the number of update instructions. Section 3 lists the previous work on SOA. Section 4 describes our technique (CSOA) and section 5 shows a small example that demonstrates the workings of the algorithm. Finally, section 6 evaluates the results of CSOA, while Section 7 summarizes the main results.

2 Motivation

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

Figure 1(a) shows a fragment of C code with the liveness information annotated at each program point. Figures 1(b) and (c) show two possible memory layouts for the variables, and the sequence in which the variables are accessed in memory. The arrows in Figures 1(b) and (c) indicate that an explicit address calculation instruction (i.e. update instruction) is required to make the address register point to the next variable, because the distance between the variables is greater than one. The layout showed in Figure 1(c) has one slot that is shared between two variables (b and g) which do not interfere at runtime. By sharing these variables, one less update instruction is required in the program. Clearly, coalescing variables increases the closeness between the variables on the stack, thus reducing the number of update instructions.

3 Related Work

The Simple Offset Assignment (SOA) problem was first studied by Bartley [5]. Later, Liao et al [16] showed that the graph problem *Maximum Weight Path Cover* (MWPC) (known to be NP-Complete) can be reduced to SOA, thus proving that SOA is NP-Hard. After

[2] Two variables interfere when they are simultaneously live.

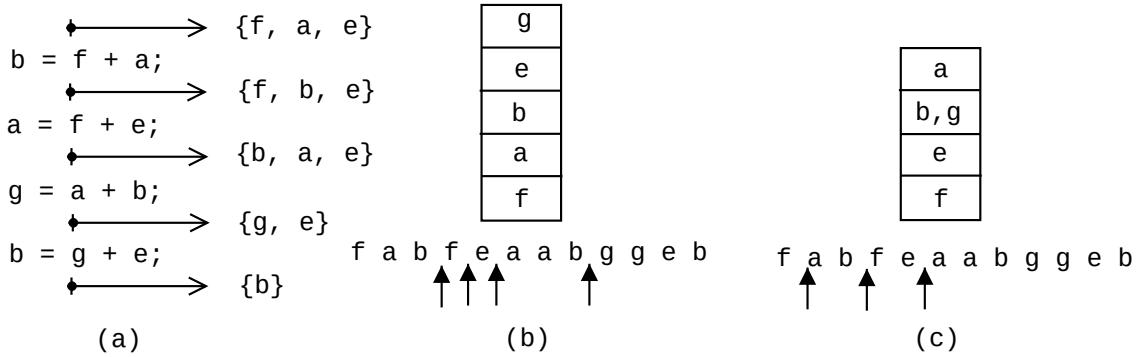


Figure 1: (a) A fragment of C code. (b) Memory layout with one slot per variable. (c) Memory layout with more than one variable per slot.

that, a large number of heuristic techniques have been proposed for SOA [16, 13, 4, 11, 19], making it one of the most studied problems in code generation for DSPs.

Liao et al [16] used a heuristic to solve SOA based on the *Kruskal Minimum Spanning Tree* algorithm [8]. Given a basic block, Liao et al [16] call *access sequence* the sequence used by the program to access variables during execution time. For example, in instruction `a = b` *op* `c`, the access sequence is `bca`. Based on the access sequence, Liao et al define an weighted graph $G(V, E)$, called *access graph*, where V is the set of variables in the basic block, and E is the set of edges. An edge $e = (u, v)$, with weight $w(e)$, indicates that there are $w(e)$ consecutive accesses to variables u and v (or v and u) in the access sequence. If two variables u and v are never accessed consecutively, then $(u, v) \notin E$. Once the access graph is constructed, Liao's algorithm tries to find a set of maximum weighted paths, called *assignment* that define the variable layout in memory. The *cost* of an assignment is the addition of the weights of all edges between variables in non-adjacent memory positions, as only auto-increment (decrement) by one is available. To illustrate these concepts consider Figure 2. Figure 2(a) shows a fragment of C code. Figure 2(b) shows the corresponding access sequence, and Figure 2(c) its associated access graph. Liao's heuristic is a greedy algorithm that, at each step, chooses the edge with the greatest weight, taking care not to select an edge that will stay with degree greater than 2 neither a edge that can form a cycle with the already selected edges. By using this heuristic, the assignment selected in the access graph of Figure 2(c) would be `fecadgb`, as highlighted in that figure. This choice results in an offset cost of four, i.e. four update instructions are required, corresponding to the non-highlighted edges.

Sudarsanam et al [20] used graph coloring to coalesce variables before SOA, but their goal was to reduce memory utilization, and they have not shown that this would improve the offset cost.

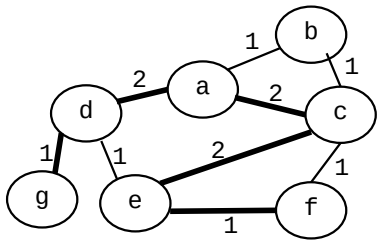
Leupers and Marwedel [14] proposed an extension to Liao's heuristic, called tie-break, that decides what edge to choose when there are edges with the same weight. Rao and Pande [19] described a technique that considers the order of the accesses. This technique optimizes the access sequence through algebraic transformations in the expression tree. In [13], Leupers and David proposed a genetic algorithm to solve SOA. Instead of using the

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

(a)

abbcacffecedadg

(b)



(c)

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

access sequence, they computed the offset assignment directly by a simulation of a natural evolution process.

Another problem related to SOA 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 in [3], and later extended by other researchers [12, 18, 2, 6].

4 Coalescing Simple Offset Assignment

This section describes an optimization for offset assignment that is based on variable liveness information. Our approach, called *Coalescing Simple Offset Assignment (CSOA)*, receives as input the access sequence and the interference graph of the variables. Its output is an offset assignment for the variables in memory.

Our technique is an extension to most of the previous heuristics that solve SOA [5, 16, 14, 4, 11]. For the purpose of testing CSOA, we use the algorithm proposed by Liao et al [16] with the tie-break heuristic in [14] 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 step of the iteration, Liao et al choose the *valid edge*, (i.e. one not already selected, that does not cause a cycle, and does not increase the degree of a vertex on the path to more than two) with maximum weight.

Algorithm 1 presents pseudo-code for CSOA. At each iteration step, instead of always choosing an edge, as in typical SOA solutions, it considers another alternative: coalescing two vertices. Specifically, we do one of two operations: (a) coalesce two vertices u and v in the access graph, if they do not interfere; (b) pick a valid edge of maximum weight from the sorted list of edges, L in the Algorithm 1, as in Liao's approach.

In Algorithm 1, function *FindCandidatePair* tries to find the two candidates for coalescing. This function 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 among all possible combinations of two vertices u and v, in the interference graph, considering only the vertices that satisfy the following conditions:

1. $(u, v) \notin$ the interference graph;
2. Coalescing u and v does not create a cycle, considering only the selected edges;
3. Coalescing u and v, does not cause the coalesced vertex to have degree greater than two, considering only the selected edges.

Algorithm 1 Coalescing-Based SOA

Input: the access sequence L_{AS},
the interference graph $G_I(V_I, E_I)$.
Output: the offset assignment.

```
(1)  G_A(V_A, E_A) ← BuildAccessGraph(L_AS);
(2)  L = sorted list of the E_A;
(3)  coal ← false;
(4)  sel ← false;
(5)  repeat
(6)      rebuild ← false;
(7)      (coal, u, v, csave) ← FindCandidatePair(G_I, u, v);
(8)      sel ← FindEdgeValidNotSel(L, e);
(9)      if (coal && sel )
(10)         if (csave ≥ w(e))
(11)             rebuild ← true;
(12)         else
(13)             mark e as selected;
(14)     else
(15)         if (coal)
(16)             rebuild ← true;
(17)         else if (sel)
(18)             mark e as selected;
(19)     if (rebuild)
(20)         RebuildAccessGraph(G_A, u, v);
(21)         RebuildInterferenceGraph(G_I, u, v);
(22)         RebuildL(L);
(23) until (!(coal || sel))
(24) return BuildOffset(G_A);
```

Then, it picks, among all pairs of vertices that satisfy the above conditions, the pair u and v whose coalescing results in the highest *csave*.

To calculate *csave*, function *FindCandidatePair* computes the following statements, where $Adj_{sel}(y)$ is the set of vertices adjacent to y, considering only the already selected edges:

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 the edge was not selected yet.

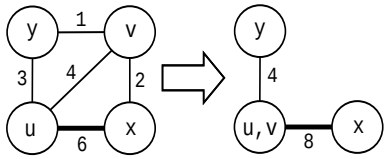


Figure 3: (a) One access graph. (b) The access graph after coalescing variables u and v.

For the sake of clarity, consider Figure 3. According to the statements above 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) is not selected) plus the weight of the non-selected edge (u, v). The value of *csave* becomes 6, 4 from edge (u, v) and 2 from edge (x, v).

After that, in line (8) of the Algorithm 1, function *FindEdgeValidNotSel* searches for the valid edge e with maximum weight $w(e)$ in the sorted list of edges L, and if it exists, flag *sel* is set.

Finally, if both *coal* and *sel* are true (line (9)), Algorithm 1 chooses (line (10)) the one that makes the best reduction in the number of update instructions.

When two vertices u and v are coalesced, parts of the access and the interference graphs need to be rebuilt in order to reflect the operation. This is performed in lines (19)-(22) of Algorithm 1. In the new access graph, all the old adjacencies of u and v must be redirected to the coalesced vertex (uv). 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.

Algorithm 1 uses function *RebuildL*, in line (22), 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.

5 Example of Coalescing SOA

To better illustrate CSOA, consider the code fragment of Figure 4(a). Each program point in the code shows the set of live variables (assuming that only `g` is live at the exit of the fragment). When Algorithm 1 is applied to this example, it receives as input the interference graph shown in Figure 4(b) and the access sequence (Figure 4(c)).

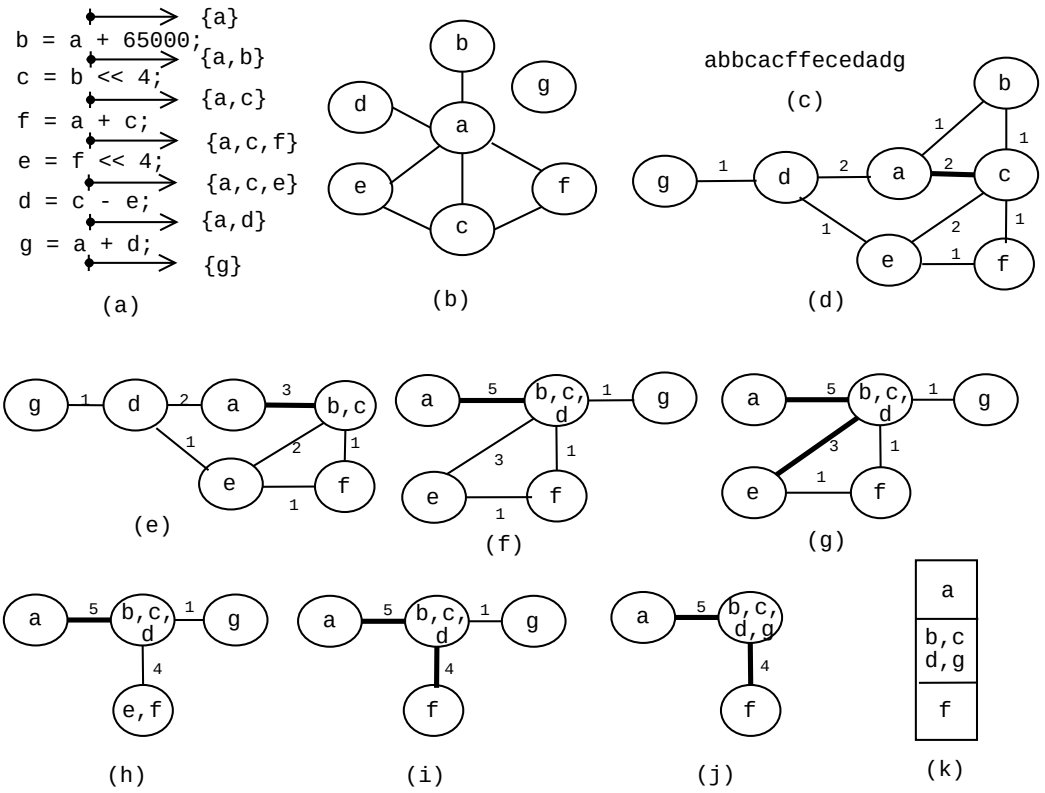


Figure 4: (a) A fragment of C code with liveness information at each point. (b) The interference graph of the variables. (c) The access sequence of this fragment. (d)-(j) The access graphs resulting after each iteration of the algorithm. (k) The memory layout. Selected edges are highlighted.

As the algorithm proceeds, it produces at each iteration the access graphs shown in Figures 4(d)-(j), after which it reaches the final memory assignment. The edges selected during the assignments are highlighted. The final memory layout is shown in Figure 4(k).

Although not illustrated, the reader should remember that, whenever two vertices in the access graph are coalesced, these vertices are also coalesced in the interference graph. In the first iteration (Figure 4(d)), edge (a, c) is selected, as no pair of vertices can be coalesced to produce saving as high as 2. In the next iteration, the best choice is to coalesce vertices b and c, given that this operation 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, that is, a, e and f. Notice that the weight of the edge between a and (bc) becomes 3, the summation of the weights of edges (a, b) and (a, c) in the previous graph. The algorithm proceeds, choosing between coalescing two vertices or selecting an edge, until no more operations are possible, thus resulting in Figure 4(j).

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

6 Experimental Results

In this section, we compare CSOA with five other approaches to SOA that are implemented in OffsetStone [11], a toolset used to test and evaluate OA algorithms. We use the MediaBench benchmark [9] to evaluate the six heuristics.

We implemented our approach using the OffsetStone [11] toolset. All Benchmark programs were compiled with the Lance [7] compiler frontend, which translates the C source code into three address code intermediate representation. Intermediate representation 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. Access sequences were then extracted from each basic block, and basic block access graphs merged on a function basis.

In Table 1, we compare CSOA with five other approaches available in OffsetStone. 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 Order First Use (SOA-OFU), which is a very simple technique that is used as the baseline in OffsetStone. The five methods used in the comparison are: SOA-OFU, a naive approach that assigns offsets to the variables in the order of their first use in the code; SOA-Liao, the algorithm described in [16]; SOA-TB, the heuristic described in [14]; SOA-GA, the heuristic described in [13]; SOA-INC-TB [11], the combination of two SOA algorithms, SOA-incremental [4] and SOA-TB [14].

Notice from Table 1 that CSOA reduces, on average, the number of update instructions to 22.4% of the SOA-OFU cost. This is a significant improvement over the previous algorithms. The best of the other algorithms (SOA-GA) reduced the offset cost, on average, to 67.0% of the SOA-OFU cost, meaning that CSOA produces 66.5% fewer update

Benchmarks	Liao	TB	GA	INC-TB	CSOA
adpcm	66.7%	63.8%	63.8%	63.8%	30.4%
epic	72.3%	70.0%	69.8%	69.8%	36.3%
g721	73.3%	70.5%	70.5%	70.5%	20.5%
gsm	72.2%	69.6%	69.6%	69.6%	14.1%
jpeg	68.8%	66.7%	66.5%	66.5%	22.2%
mpeg2	71.5%	69.6%	69.4%	69.5%	24.5%
pegwit	68.3%	62.2%	61.9%	61.9%	26.5%
pgp	70.8%	67.2%	67.1%	67.1%	22.8%
rasta	65.7%	64.7%	64.7%	64.7%	13.9%
Average	69.9%	67.1%	67.0%	67.0%	22.4%

Table 1: Offset costs relative to OFU cost.

instructions than SOA-GA, according to OffsetStone. We believe that this exceptional improvement is due to the increased closeness between the stack variables, resulting from the variable coalescing strategy. CSOA, in opposition to other techniques that naively coalesce variable slots [20], 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.

Table 2 lists, for each benchmark program, the total number of variables, the final number of memory slots used by CSOA, and the ratio between these two quantities. Notice that we have not listed these data for all other methods. This is simply because the number of memory slots resulting from them is equal to the total number of scalar variables, given that none of them perform variable coalescing.

From Table 2, one can observe that our method reduces the size of stack used to store local variables to 28.7%, when comparing to the other five methods [16, 14, 13, 11, 4]. Table 2 also shows that CSOA packs 3.5 variables per memory slot, on average, instead of just one as in the previous approaches.

Finally, Table 3 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 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.

7 Conclusions and Future Work

In this paper we proposed a heuristic to solve the Simple Offset Assignment (SOA) problem based on coalescing memory variable slots. The experimental results show that our method

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

Benchmarks	#Variables	#Memory Slots	#Variables/slot
adpcm	198	55	3.6
epic	4163	1125	3.7
g721	1152	289	4.0
gsm	4817	1048	4.6
jpeg	13690	4778	2.9
mpeg2	8828	2815	3.2
pegwit	4122	1454	2.8
pgp	9451	2989	3.2
rasta	4040	1056	3.8
Average	3538.7	1016.3	3.5

Table 2: Memory savings using the CSOA.

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%

Table 3: Percentage of temporary variables, considering as temporary a variable that is alive only in one basic block.

(CSOA) eliminates, on average, 66.5% of the update instructions when comparing with the best approach to the problem (SOA-GA) from OffsetStone [11]. Another important side effect of our technique is the reduction in the size of the memory layout by 71.3% when comparing again with SOA-GA. The large presence of temporaries in DSP programs and the increased closeness resulting from the coalescing technique seem to explain well these exceptional numbers.

In this paper, we only addressed the SOA problem. We are currently investigating the use of coalescing to partition the access graph in the case of the General Offset Assignment (GOA) problem.

8 Acknowledgments

This work was supported by FAPESP (Proc. 01/12762-5).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techinques and Tools*. Addison-Wesley Publishing Company, 1986.

[2] G. Araujo, G. Ottoni, and M. Cintra. Global array reference allocation. *ACM Trans. on Design Automation of Electronic Systems*, 7(2):336–357, April 2002.

[3] G. Araujo, A. Sudarsanam, and S. Malik. Instruction set design and optimizations for address computation in DSP architectures. In *Proc. of the 9th. ACM/IEEE International Symposium on System Synthesis*, pages 102 – 107, November 1996.

[4] S. Atri, J. Ramanujam, and M. Kandemir. Improving offset assignment for embedded processors. *Lecture Notes in Computer Science*, 2017, 2001.

[5] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software - Practice and Experience*, 22(2):101–110, 1992.

[6] M. Cintra and G. Araujo. Array reference allocation using ssaform and live range growth. In *Proc. of the ACM SIGPLAN 2000 LCTES*, pages 26–33, June 2000.

[7] L. R. C. compiler. http://ls12-www.cs.uni-dortmund.de/lance/.

[8] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30)*, December 1997.

[10] R. Leupers. Code generation for embedded processors. In *International System Synthesis Symposium*, 2000.

[11] R. Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proceedings of the 12th International Conference on Compiler Construction*, April 2003 (to appear).

[12] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Proc. of the Asia South Pacific Design Automation Conference (ASP-DAC)*. IEEE, February 1998.

[13] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proc. of the International Symposium on System Synthesis (ISSS)*, pages 3–8, 1998.

[14] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *International Conference on Computer-Aided Design (ICCAD)*, pages 109–112, 1996.

[15] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors.* PhD thesis, Massachusetts Institute of Technology, 1996.

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

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[18] 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*, pages 274–288. Springer, April 2001.

[19] 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.

[20] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Design Automation Conference*, pages 287–292, 1997.