

Integrated offset assignment

Mattias Eriksson Christoph Kessler
Linköping University, Sweden

email: {mattias.eriksson|christoph.kessler}@liu.se

Abstract

One important part of generating code for DSP processors is to make good use of the address generation unit (AGU). In this paper we divide the code generation into three parts: (1) scheduling, (2) address register assignment, and (3) storage layout. The goal is to find out if solving these three subproblems as one big integrated problem gives better results compared to when scheduling or address register assignment is solved separately. We present optimal dynamic programming algorithms for both integrated and non-integrated code generation for DSP processors. In our experiments we find that integration is beneficial when the AGU has 1 or 2 address registers; for the other cases existing heuristics are near optimal. We also find that integrating address register assignment and storage layout gives slightly better results than integrating scheduling and storage layout. I.e. address register assignment is more important than scheduling.

1 Introduction

Digital signal processors often include an *address generation unit* (AGU) for calculating memory addresses. These AGUs have a register containing pointers into memory, and every time one of these pointers is read it can, at the same time, be incremented or decremented by a small value. Generating fast and compact code for such an architecture requires good utilization of the AGU.

The basic problem of finding the best storage layout given an access sequence and a single address register was first studied by Bartley [3]. He shows

how to transform the access sequence into an undirected graph where the nodes represent variables and an edge between two nodes is given a weight equal to the number of times the nodes are neighbors in the access sequence. Finding a storage layout that maximizes the utilization of auto-increment/decrement is equivalent to finding a maximal Hamiltonian path in the access graph. Finding a maximal Hamiltonian path is an NP-complete problem; Bartley presented a greedy heuristic that has since been improved by Liao et al. [9], Leupers and Marwedel [8] and by Ali et al. [1]. This basic problem with 1 address register and an increment/decrement range of $-1/+1$ is called the *simple offset assignment* problem (SOA).

A generalization to *general offset assignment* where there are multiple address registers in the AGU was presented by Liao et al. [9]. They propose a heuristic method to solve GOA by reducing it to one SOA instance per address register. Each of the SOA instances handles accesses to a disjoint subset of the variables in the access sequence. This heuristic was improved by Leupers and Marwedel [8] by finding better ways to partition the access sequence into subsequences.

Gebotys presents a fast *minimum cost circulation technique* (MCC) for optimal address register assignment for a given memory layout and access sequence [5]. She concludes that memory layout only has minor impact on the final quality of the generated code when address register assignment is solved optimally. This conclusion is not supported by the experimental evaluation by Huynh et al. where a range of GOA-heuristics are evaluated with the MCC-technique [6].

Leupers has presented a comparison of a range

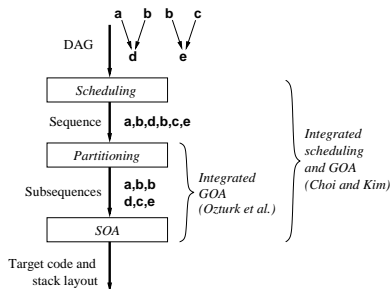


Figure 1: Choi and Kim have presented a heuristic algorithms for the fully integrated problem [4]. Özturk et al. have solved GOA optimally [10].

of different SOA-heuristics [7]. One of the conclusions is that the performance differences of the SOA-heuristics is surprisingly small for real-life problem instances. He also finds that comparisons of SOA-algorithms using random access sequences only give a coarse performance evaluation, and that the results are not always the same when real-life problem instances are used.

Udayanarayanan and Chakrabarti have presented a GOA-heuristic that performs modify register optimization as a final step in the code generation [13]. Their evaluation shows that, when a modify register is available, the impact of storage layout is not an important factor to the final quality of the solution.

Özturk et al. have presented an optimal integer linear programming formulation for GOA with modify registers. Their experiments show that all instances in their experiment are solved to optimality in a few minutes with a good ILP solver [10].

Atri et al. explore how commutativity of operations in the SOA-problem can be used to reduce the number of edges in the access graph, allowing for better SOA-solutions [2].

Rao and Pande have presented heuristic methods for SOA which uses commutativity, associativity and distributivity of operations to reorder the access sequence. The heuristics are also extended to GOA and experiments show a static code size reduction (of whole programs) of 2% on average compared to previous heuristics [11].

Choi and Kim have described a heuristic algorithm

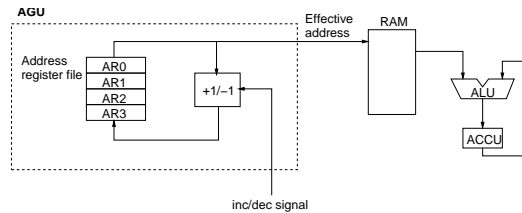


Figure 2: The architecture has an address generation unit with a number of address registers. The ALU has a single accumulator register.

that integrates scheduling and GOA. Their algorithm works in two steps: first an initial schedule and storage assignment is computed; second this solution is iteratively improved by reordering the access sequence and redoing the storage assignment [4]. This heuristic is compared to the one presented by Leupers and Marwedel [8] and shows good improvements. In Section 3 we compare our optimal algorithm to this heuristic by Choi and Kim.

In this paper we study the question of how important scheduling and address register assignment is for the offset assignment problem. We do this by comparing optimal solutions, in which scheduling and address register assignment are integrated, to solutions of non-integrated code generation. See Figure 1 for an illustration of different levels of integration. The two main contributions of this paper are: (1) A new optimal algorithm that solves instances of the integrated general offset assignment problem is presented (Section 2); (2) an experimental evaluation that compares the results of the optimal integrated method to the results of an optimal non-integrated method is presented (Section 3). This is interesting also from a theoretical perspective because it shows for which cases integration of subtasks is worth considering.

2 Integrating offset assignment and scheduling

In this section we will introduce the algorithms for integrated offset assignment based on dynamic programming.

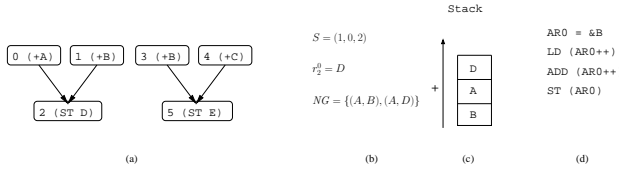


Figure 3: An example of a partial solution.

2.1 Partial solutions

We assume a simple accumulator-based processor, depicted in Figure 2, which has:

- one ALU with an accumulator register, and
- one AGU with $k \geq 1$ address registers; an address register can be incremented or decremented by 1 each time it is read.

This kind of setup is common in real-world DSPs; variations of it can be found in eg. TI-C2x [12].

A problem instance is a basic block for which we want to generate code. The basic block is represented by a directed acyclic graph (DAG) $G = (V, E)$. Each node $x \in V$ represents an operation. An edge $(u, v) \in E$ represents either true data dependencies or some other precedence constraint.

The goal of the code generation is to minimize the number of explicit assignments to address registers; we will use the term *cost* for this. Minimizing the *cost* will both minimize run time and code size of the produced code.

Assume that we have k address registers in the AGU and that we have n nodes to schedule. Then we can define what we mean by a partial solution:

Definition 1. A partial solution P of length l consists of:

- A schedule $S = (s_1, s_2, \dots, s_l)$, $l \leq n$, where s_t is the node that is scheduled at slot t , $1 \leq t \leq l$.
- Address register usage: r_t^i is the variable that address register i points to when it was last used (at slot t or earlier); if address register i has not been used yet, we can assume that r_t^i points to every variable, $0 \leq i < k$, $0 \leq t < n$.

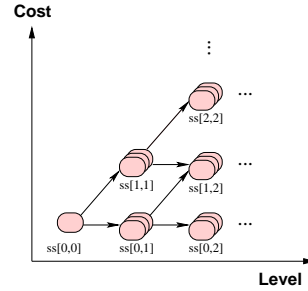


Figure 4: Structure of the solution space (ss) used in the DP-algorithm. The algorithm expands partial solutions with smallest cost first.

- A set NG of neighbour pairs, if $(v, w) \in NG$ the variables v and w are neighbours on the stack.

Example 2. Figure 3(a) shows a DAG representing the basic block:

$$D = A + B;$$

$$E = B + C;$$

Assuming that we have a single address register (ARO) Figure 3(b) shows a partial solution of length 3. In this partial solution the node sequence is $S = (1, 0, 2)$ which means that first B is loaded into the accumulator register, then A is added to the accumulator, and then the result is stored in D. The contents of the set NG implies that A is neighbor to both B and D, this is satisfied when the stack is laid out as in Figure 3(c). The corresponding assembler code for the partial solution is shown in Figure 3(d); note that the last instruction ST (ARO) is not yet finalized, a post-increment or decrement may be added as the partial solution is extended.

2.2 Integrating GOA and scheduling

The dynamic programming algorithm in Figure 5 works by enumerating the partial solutions in a two dimensional solution space shown in Figure 4. The first dimension is *cost* and second dimension (*level*) is the number of scheduled nodes. Initially, the solution space (ss) contains a single empty partial solution with *cost* 0 and *level* 0 (lines 0-1). The outer

Algorithm: DP-RS.

Method: Dynamic programming in two dimensions: *cost* and number of scheduled nodes (*level*).

Input: A DAG representing a basic block and the set of available address registers (\mathcal{AR}).

Output: Schedule, storage layout and address register assignment.

```

0: initialize  $ss[cost, level]$  to empty lists for all  $cost$  and  $level$ 
1: push ( $ss[0, 0]$ , an empty partial solution)
2: for  $cost$  from 0 to infinity do
3:   for  $level$  from 0 to infinity do
      // keep only one in every group of comparable solutions
4:   prune( $ss[cost, level]$ )
5:   while (  $ss[cost, level]$  not empty )
6:      $cur = \text{pop}(ss[cost, level])$ 
7:     if ( $cur$  is a complete solution)
8:       return  $cur$ 
9:     foreach ( $n \in$  selectable nodes in  $cur$ )
10:       $n_v =$  the variable of node  $n$ 
11:      Case 1: A  $reg \in \mathcal{AR}$  points to  $n_v$ :
12:        push ( $ss[cost, level+1]$ , extend( $cur, reg, n_v$ ))
13:      Case 2: No  $reg \in \mathcal{AR}$  points to  $n_v$ :
14:        foreach ( $reg \in \mathcal{AR}$ )
15:          Case 2A:  $reg$  points to a neighbor of  $n_v$ :
16:            push ( $ss[cost, level+1]$  extend( $cur, reg, n_v$ ))
17:          Case 2B:  $reg$  points to  $v$  that is not a neighbor of  $n_v$ :
18:             $new = \text{extend}(cur, reg, n_v)$ 
19:            if (make_neighbor( $new, v, n_v$ ))
20:              push ( $ss[cost, level+1]$ ,  $new$ )
21:            push ( $ss[cost+1, level+1]$ , extend( $cur, reg, n_v$ ))

```

Figure 5: Dynamic programming algorithm.

loop over *cost* progresses along the cost axis of the solution space. And the loop over *level* progresses over number of scheduled nodes in the DAG.

The algorithm proceeds by, for each partial solution at the current cost and level, expanding this partial solution by selecting an unscheduled node from the DAG. The node that is selected must be ready, i.e. all its predecessors must have already been selected (line 9). If the algorithm was to enumerate all possible solutions it would be necessary to create a new partial solution at the next level, by using each of the available address registers, however if an address register already points to the variables of the selected node, then it is enough to consider this address register only, see Lemma 3.

The function $\text{extend}(cur, reg, v)$ creates a copy of the current partial solution (*cur*) and then extends it

by using register *reg* for accessing the variable *v*.

In Case 2 (line 13) all possibilities for extending the current partial solution are exhausted.

The function make_neighbor (line 19) tries to make *v* and n_v neighbors in *new*. The function returns true if it is successful. The reason for a failure is either that one of the variables already has two neighbors, or that making the variables neighbors would create a cycle.

The algorithm terminates when the current partial solution (line 6) is a complete solution, i.e. all of the nodes in the DAG have been scheduled. Termination can be done earlier; if a new partial solution is created that is complete, and has the same cost as the current partial solution (i.e. on lines 12, 16, 18 or 20, but not on line 21) then this solution is optimal and the algorithm can exit. This early exit strategy is

not included in the presentation in Figure 5 to avoid cluttering the presentation.

2.3 Pruning of the solution space

The partial solutions are constructed systematically in order of cost. This means that expanding less promising partial solutions is postponed as much as possible. However, even with this postponing it is obvious that any algorithm based on enumeration will suffer from a combinatorial explosion for large instances. We can delay the explosion of the number of partial solutions in the solution space if we can identify partial solutions, a and b , such that a is *at least as good as* b in the sense that a best final solution b^* to which b is a partial solution cannot be better than a best final solution a^* to which a is a partial solution. Then the partial solution b can be removed without affecting the optimality of the enumeration. If both a is at least as good as b and b is at least as good as a , we say that a and b are *comparable*.

One obvious case where two partial solutions are comparable is when all of the following are true:

- The last element in the schedules are equal.
- The same set of nodes has been scheduled.
- The set of variables that the address registers point to are the same.
- The NG sets of both partial solutions are equal.

Before nodes of a certain cost and level are expanded we can do a pruning step (see Figure 5, line 4). The list of partial solutions is first sorted and then the list is traversed and only one in each group of comparable solutions is kept. Another way in which we can delay the combinatorial explosion is to avoid doing a complete enumeration when it is not necessary. Consider a situation where we expand a partial solution in which an address register already points to the next variable in the access sequence (Case 1, line 11, in Figure 5), we can use this address register and do not need to make additional partial solutions where another register is used. Formally:

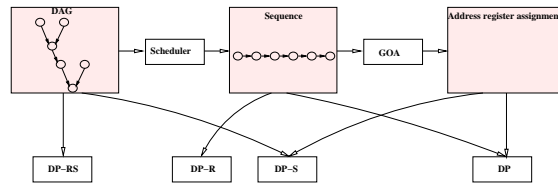


Figure 6: The algorithm flow.

Lemma 3. *Let P be a partial solution of length l in which $r_l^i = v$ and $r_l^j = w \neq v$. Assume that the next access is to variable v . Let P^i be a partial solution that uses address register i for this access to v and let P^j be a partial solution that uses address register j for this access to v . Then for any extension of P^j to a final solution there exists an extension to P^i that is at least as good.*

We omit the proof due to lack of space.

2.4 Optimal non-integrated GOA

One question that we want to answer in this paper is how much can be gained by integrating the parts of the code generation. To find the answer to this question we need to run experiments where we compare optimal integrated code generation to optimal non-integrated code generation. In this subsection we will present the non-integrated algorithms that we have used for the experiments: DP-R, which does scheduling separately, DP-S, which does address register assignment separately, and DP, which does all parts separately.

DP-R works exactly like DP-RS except that the nodes have already been scheduled. I.e. only one node is selectable on line 9 in Figure 5.

DP-S integrates scheduling, just like DP-RS does. But it forces all partial solutions to conform to the address register assignment found by `solve_goa` [8]. I.e. on lines 11, 13 and 14 in Figure 5 ' $reg \in \mathcal{AR}$ ' is replaced by ' $reg = ARX$ ', where ARX is the address register used for variable n_v in the solution of a `solve_goa` run. Note that since scheduling is integrated we can not simply partition the access sequence and solve k independent subproblems because the sub-sequences will not be independent.

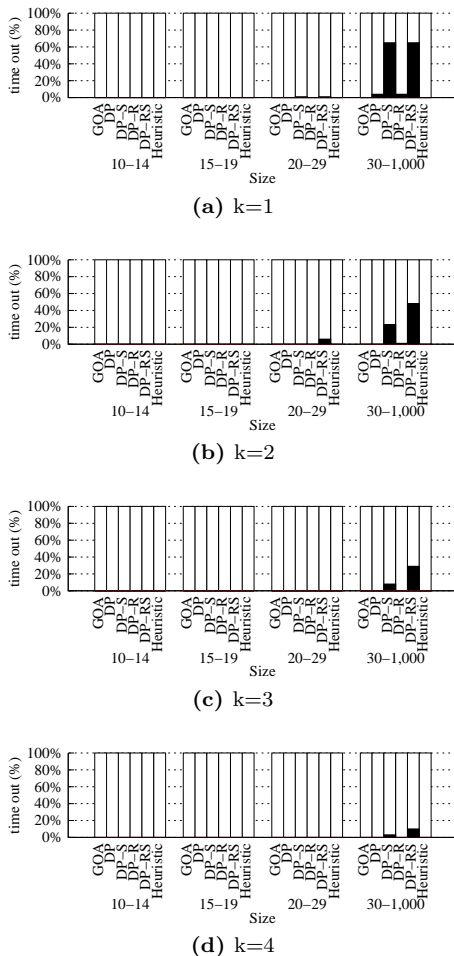


Figure 7: Bar charts showing how often our algorithms time out for $k = 1$ to 4 address registers.

DP does not integrate any of the phases; all it does is calculate the optimal stack layout with a fixed schedule and a fixed address register assignment. See Figure 6 for a sketch of the different parts of DP, DP-R, DP-S and DP-RS.

2.5 Handling large instances

The large problem instances often do not succeed because the dynamic programming solver runs out of

memory. We can get around this problem by counting the current total number of partial solutions, and when we reach a maximal number of partial solutions we switch to an enumeration strategy that prioritizes extending partial solutions with higher levels, i.e. partial solutions that are nearer to completion. Then the program will not run out of memory, but obviously we may lose some opportunities for pruning comparable partial solutions. In other words the computation time increases but the memory usage is limited.

2.6 Integrated heuristic

For comparison we have implemented the heuristic algorithm Naive-it by Choi and Kim [4]. The algorithm is integrated and is based on iterative improvement. Internally it uses the function `solve_goa` from [8]. This algorithm is very fast compared to our dynamic programming algorithms and we will see in our experimental evaluation that it finds results that are not far from the optimum.

3 Experimental evaluation

The experiments were run on an Intel i7 3.06 GHz, each instance had a time limit of 1 hour, and a memory limit of 10GB. In total we have 400 instances divided by sequence length into 4 classes (100 instances in each class). The sequence lengths are 10-14, 15-19, 20-29 and 30+. The source of the instances is Offsetstone¹. However, the Offsetstone problems are already scheduled and the graphs are not available, we have therefore created DAGs of the sequences so that the Offsetstone sequence is one possible schedule for the DAG that we create. We have assumed that each operation is commutative and takes two operands. Obviously graphs generated in this way are most likely not equivalent to the original problems. But it is our hope that these DAGs are more realistic than randomly generated problems.

The algorithms included in the evaluation are:

- GOA, `solve_goa` from [8];

¹The Offsetstone benchmark suite is available from <http://address-code-optimization.org> (2010).

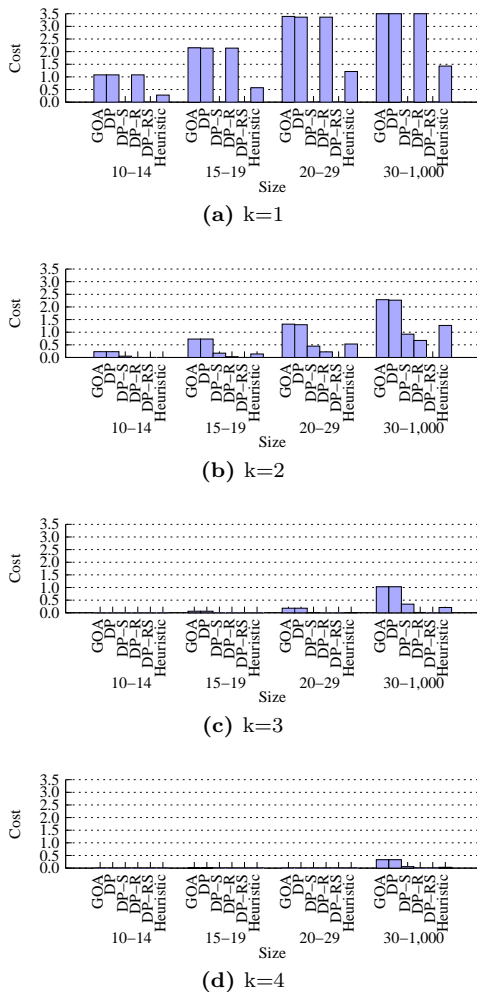


Figure 8: Number of extra explicit address register assignments compared to DP-RS (DP-S in (a)). DP-RS and DP-R are not included in (a) since register allocation is not relevant in this case.

- DP, DP-R and DP-S, described in Section 2.4;
- DP-RS, our fully integrated algorithm, described in Section 2.2;
- Heuristic, our implementation of Naive-it from [4].

Figure 7 shows the success rates of the algorithms. When we set the time limit to 1 hour all algorithms finds a solution. However, for some of the large instances the dynamic programming algorithms times out before a guaranteed optimal solution is found. This case is most common for the larger instances with the fewest number of address registers. The reason for this is that when the number of address registers is large it is easier to find a solution that has no extra explicit assignments of the address registers, i.e. the value of the *cost* variable is 0 in our algorithm. We can also see in this chart that integrating scheduling is much more expensive than integrating address register assignment.

Figure 8 shows the difference in number of clock cycles on average between each algorithm and the optimum for the cases where the optimum is known (i.e. when the fully integrated DP-RS algorithm terminates before the time out is reached). We find that the greatest difference between the integrated and non-integrated algorithms occurs when there is only a single address register (Figure 8(a)). In this case the optimum is found by DP-S since there is no address register assignment. For the largest basic blocks (more than 30 nodes) the average reduction in number of address code operations between the non-integrated DP to the integrated DP-S is 3.5.

Looking at the case with 2 address registers (Figure 8(b)) we see that the differences between the algorithms are smaller. Also it is clear that more is gained by integrating register assignment than is gained by integrating scheduling. But both DP-R and DP-S, on average, outperform the Naive-it heuristic for the larger instances but the difference is quite small.

When increase the number of address registers even more to (Figure 8(c) and (d)) we find that integrated algorithms do not give much better results than the non-integrated ones. I.e. the cheaper algorithms are already quite close to optimal.

4 Conclusions

The question of how to optimally use an address generation unit when compiling for DSP-machines is important. In this paper we have studied the impor-

tance of scheduling and address register assignment with regards to the outcome of the offset assignment. We have compared optimal algorithms for integrated offset assignment to both optimal and heuristic algorithms that are not fully integrated.

We have run experiments where we compare the solutions of our fully integrated method (DP-RS) with the solutions of our separated methods (DP-S, DP-R and DP). While it is true that finding optimal results requires a lot of time and computer memory, we think that the results are important because they show exactly how much that is gained, on average, by integrating the different parts of the code generation. The results of our experiments suggest that integrating the phases of the code generation has much potential for improving code when the number of address registers is 1 or 2, and if the number of address registers is more than that, only small improvements can be expected by the integration.

References

- [1] Hesham S. Ali, Hatem M. El-Boghdadi, and Samir I. Shaheen. A new heuristic for SOA problem based on effective tie break function. In *SCOPES '08: Proc. of the 11th int. workshop on Software & compilers for embedded systems*, pages 53–59, 2008.
- [2] Sunil Atri, J. Ramanujam, and Mahmut T. Kandemir. Improving offset assignment on embedded processors using transformations. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 367–374, London, UK, 2000. Springer-Verlag.
- [3] David H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exp.*, 22(2):101–110, 1992.
- [4] Yoonseo Choi and Taewhan Kim. Address assignment in DSP code generation - an integrated approach. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(8):976–984, 2003.
- [5] Catherine Gebotys. DSP address optimization using a minimum cost circulation technique. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 100–103, Washington, DC, USA, 1997. IEEE Comp. Society.
- [6] Johnny Huynh, José Amaral, Paul Berube, and Sid Touati. Evaluation of offset assignment heuristics. In *HiPEAC '07: Int. Conf. on High Performance Embedded Architectures and Compilers*, pages 261–275, Ghent, January 2007.
- [7] Rainer Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *proc. of the 12th int. conf. on compiler construction*, pages 290–302, 2003.
- [8] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *ICCAD '96: Proc. of the 1996 IEEE/ACM int. conf. on Computer-aided design*, pages 109–112, Washington, DC, USA, 1996. IEEE Computer Society.
- [9] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):235–253, 1996.
- [10] O. Ozturk, M. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In *Proc. 16th ACM Great Lakes symposium on VLSI*, pages 37–42, New York, USA, 2006. ACM.
- [11] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *PLDI*, pages 128–138, 1999.
- [12] Texas Instruments Incorporated, Upper Saddle River, NJ, USA. *Second-generation TMS320 user's guide*, 1988.
- [13] Sathishkumar Udayanarayanan and Chaitali Chakrabarti. Address code generation for digital signal processors. In *DAC '01: Proc. of the 38th annual Design Automation Conference*, pages 353–358, New York, USA, 2001. ACM.