Address Code Generation for Digital Signal Processors

Sathishkumar Udayanarayanan
Arizona State University
Tempe, AZ
usathish@ieee.org

Chaitali Chakrabarti
Arizona State University
Tempe, AZ
chaitali@asu.edu

ABSTRACT

In this paper we propose a procedure to generate code with minimum number of addressing instructions. We analyze different methods of generating addressing code for scalar variables and quantify the improvements due to optimizations such as offset assignment, modify register optimization and address register assignment. We propose an offset assignment heuristic that uses k address registers, an optimal dynamic programming algorithm for modify register optimization, and an optimal formulation and a heuristic algorithm for the address register assignment problem.

1. INTRODUCTION

Traditionally DSP compilers have been unable to meet the very tight constraints of code size and performance and the critical DSP application modules have been hand-written. The use of conventional code generation techniques in DSPs results in very inefficient code due to limitations like non-homogeneous register sets, specialized registers, special functional units, and irregular datapaths. As a result, code generation and optimization specifically for DSPs has recently received a lot of attention.

Address code generation is a very important part of code generation since it can account for over 50% of all program bits and 1 out of every 6 instructions for a typical general-purpose processor [1]. In fact, for a set of programs in MediaBench [4] that was compiled for Motorola *DSP56000* family, we found that more than 55% of the instructions involve address registers. Thus optimizing the addressing code could lead to significant improvement in code size and performance. In this paper, we consider the address generation unit of the DSP and develop an address code generation methodology to utilize it efficiently.

The address generation unit (AGU) in a DSP typically consists of multiple address registers, modify registers, and an independent arithmetic unit. The address register can be incremented or decremented or modified by adding or subtracting the value in the modify register. This subsumption of address arithmetic allows improvement of both code size and performance. Liao [7] has modeled the problem of finding the offsets for the variables and suggested heuristics to solve it for the case of a single address register and multiple address registers. Sudarsanam et. al. [10] consider the performance benefits of having an auto-increment with increments varying from $-l$ to $+l$. Leupers and Marwedel [6] have addressed the problem of finding the values that the modify register should hold at different times in the program. Leupers and David [5] present a genetic algorithm based technique to handle arbitrary address register file sizes and auto-increment ranges. They also integrate the allocation of modify registers into offset assignment. The problem of generating addressing code given the offset assignment and the number of address registers has been modeled using network flow by Gebotys [3].

In this paper, we analyze different methods of generating addressing code for scalar variables and quantify the improvements due to the following optimizations: offset assignment, modify register optimization and address register assignment. Our address generation procedure tries to reduce the number of addressing instructions by reducing the number of jumps[1] during offset assignment and address register assignment. In this paper we present

- A faster optimal formulation to perform address register assignment given an offset assignment and the number of address registers.
- An optimal dynamic programming based algorithm to utilize the modify registers efficiently.
- Quantitative analysis of the different optimizations with respect to number of addressing instructions and computation time.

An analysis of the results on pseudo-random sequences as well as sequences from procedures used in the DSPStone benchmarks showed that use of these optimizations result in a 70-80% overall reduction in the number of addressing instructions compared to the case when optimization is not used. Furthermore, the modify register optimization is effective and computationally inexpensive, and should be included in any address generation procedure. Another interesting conclusion is that the address register assignment followed by modify register optimization makes it almost unnecessary to have an intelligent offset assignment algorithm.

[1]A jump occurs when auto-increment/auto-decrement cannot be used.

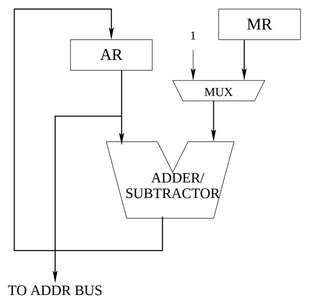

Figure 1: Address Generation Unit Model; AR stands for Address Register and MR stands for Modify Register

The rest of the paper is organized as follows: Section 2 introduces the AGU model that we will use throughout this paper and defines the notations and the problems that we consider; section 3 talks about simple offset assignment and general offset assignment problems; section 4 describes our dynamic programming algorithm for utilizing modify registers; section 5 presents the address register assignment problem and proposes an optimal formulation and a heuristic algorithm to solve it; section 6 considers five different configurations and compares their performance on a set of random access sequences and sequences obtained from real programs; section 7 concludes the paper.

2. PRELIMINARIES

2.1 AGU Model

The Address Generation Unit (AGU) that we assume is shown in Figure 1. It consists of an address register(AR), a modify register (MR) and an adder/subtractor.The AGU is typical of many DSPs such as *Motorola DSP56000* and *NEC 7701*. While in Figure 1, we show just one address register-modify register pair, our model supports k such pairs. In our model, we do not consider support for circular addressing or bit reversed addressing. Our model can be easily extended to handle other DSPs such as *Analog Devices ADSP21xx and SHARC*, *Lucent DSP16xx and StarCore*, and *Texas Instruments TMS320C5x*.

2.2 Motivation

In address computation, knowledge of how the variables are accessed in the code can be utilized to intelligently assign addresses to variables, thereby generating fewer addressing instructions. The order in which variables are accessed in the program is referred to as the *access sequence*. We next describe an example that illustrates how placement of variables in memory affects the number of addressing instructions.

Example 2.1

Consider the access sequence shown in figure 2(a). Variables a and c are accessed consecutively in 4 places. The assignment shown in figure 2(b) is obtained on the basis of first-use of variables. The positions in the access sequence where auto-increment/auto-decrement cannot be used (we call such a situation a *jump*) are shown using arrows in figure 2(c). There are a total of 10 jumps which necessitate additional instructions. Now consider a more intelligent assignment shown in figure 2(d). Variables that are accessed

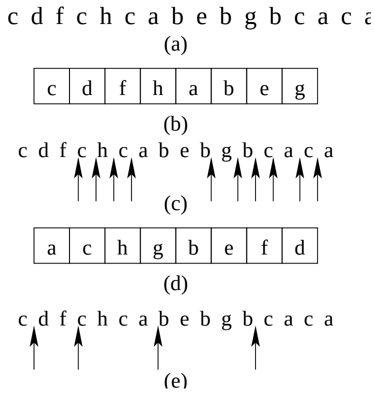
c d f c h c a b e b g b c a c a
(a)
c d f h a b e g
(b)
c d f c h c a b e b g b c a c a
(c)
a c h g b e f d
(d)
c d f c h c a b e b g b c a c a
(e)

Figure 2: Example 2.1: (a) Access Sequence. (b) An assignment based on first-use. (c) The places requiring a jump in address value for the assignment in (b). (d) An intelligent assignment. (e) The places requiring a jump in address value for the assignment in (d). (adapted from [8])

consecutively several times are now put in neighboring locations. As a result, there are now only 4 jumps as shown in Figure 2(e). Clearly, by placing the variables in proper locations, the number of address generation instructions can be reduced considerably. □

2.3 Overview

In this paper, we minimize the number of instructions required for addressing a sequence of variable accesses using $k \geq 1$ address registers. We assume that each variable is assigned to only one location and that we can allocate any relative location to a variable (there are no restrictions on the relative order.)

The input of our procedure is a sequence of variable accesses and the number of address registers. The output of our procedure gives (i) the address register to be used for each access, and whether to increment or decrement or modify its value, and (ii) whether to load the address register (and/or modify register) after each access and with what value(s).

There are two important problems in address code generation:

1. What is the address to be given to each variable (referred to as *offset assignment*)?
2. Which AR is to be used to address a particular access (referred to as *address register assignment*)?

Our aim is to minimize the number of instructions during offset assignment. This is a very complex problem and so initially we consider the problem of reducing the number of jumps. Similarly, during address register assignment, we minimize the number of addressing instructions by reducing the number of jumps. The modify registers enable us to implement the jumps with fewer addressing instructions.

Problem 1: Simple Offset Assignment (SOA)

Given an access sequence, $L = \{x_1, x_2, \ldots, x_l\}$, and a set of distinct variables of L, $Y = \{y_1, y_2, \ldots, y_n\}$, find a function f_A that maps the variables in Y to the offset addresses such that the number of jumps is minimized when using only one address register.

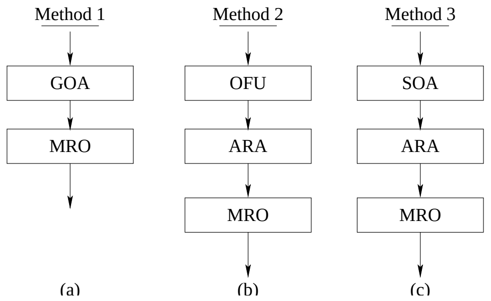

Figure 3: The approaches to be analyzed: (a) Method 1 (b) Method 2 and (c) Method 3

Problem 2: General Offset Assignment (GOA) Given an access sequence, L, find a function f_A such that the number of jumps is minimized when using $k > 1$ address registers.

Problem 3: Address Register Assignment (ARA) Given an access sequence, L, an address assignment, f_A, and the number of address registers, k, find a function f_{AR} that gives the address register that is to be used for each access such that the number of jumps is minimized. Note that for the case of $k = 1$, the function is constant, R_1.

Problem 4: Modify Register Optimization (MRO) Given a sequence a jumps [2], S_j, find a function f_{MR} that specifies the value in the modify register during each jump such that the difference between the number of jumps and the number of instructions after MRO is maximized.

Approaches Considered: For DSPs with only one address register, we could solve SOA followed by MRO. We will show how important these are in section 3.1 and section 4 and the potential reduction in instructions. For DSPs with $k > 1$ address registers, we could take three approaches. First, we could solve GOA, followed by ARA separately for each R_i (which is a constant since $k = 1$), followed by MRO. In the second approach, we could use the order of first use (OFU), followed by ARA for k address registers, and then MRO. In the third approach we could solve SOA, followed by ARA, followed by MRO. We show the different approaches in figure 3. The following sections discuss the different optimizations and compare the performance of the three methods.

2.4 Experimental Setup

To quantify the effect of the optimizations, we need access sequences that are representative of those arising in DSP systems. We obtained 51 access sequences from 20 different procedures using SPAM [11]. 14 of the procedures were kernels from the DSPStone benchmarks and the remaining 6 were part of the ADPCM program from the same benchmark set. Here we show the results of 8 representative programs. The corresponding sequences are referred to as program access sequences.

To make the experiments unbiased and independent of the processor, pseudo random access sequences were also used as inputs. 20 random sequences are used for different variable set sizes (n) and access lengths (l) and the average values are given. The execution times of the optimizations were measured using functions in sys/time.h. The programs were run in a dual Pentium III server with 1GB RAM running Redhat Linux.

[2] A *jump* is a situation that requires an address register, R_i, to be modified by addition or subtraction of a value more than 1. If accesses x_i and x_j occur next to each other (in L, x_i and x_j may not occur next to each other but for a particular R_i, they could be next to each other), the value of the jump is computed as $|f_A(x_j) - f_A(x_i)|$.

3. OFFSET ASSIGNMENT

3.1 Simple Offset Assignment

We implemented the heuristics of Liao [7] and Leupers [6] and ran them on random access sequences and real program sequences. The additional address instructions are compared with that generated due to an assignment performed according to the order of first use (OFU). On the average, there is 20% reduction compared to OFU in the number of additional instructions when these heuristics are used. We will see in the next section that after modify register optimization, the difference between the performance of the three offset assignment algorithms reduce further.

3.2 General Offset Assignment

Liao [7] and Leupers [6] have developed heuristics for partitioning the variables into $l \leq k$ subsets, where k is the number of available address registers. We developed an algorithm that iteratively partitions the variables based on a very simple cost function [12]. For our experiments, the number of address registers, k, was set to 4. For random access sequences, Leupers' heuristic is better than our GOA by about 2% while for most (6 out of 8) program access sequences, our heuristic is better. For larger random sequences, our heuristic is faster than Leupers' without much compromise in results. In this paper, in Method 1, we implement Leupers' GOA.

4. EXPLOITING MODIFY REGISTERS

The modify register can be used effectively to reduce the number of additional instructions. Instead of loading the address register with a new value, we can add/subtract the value that is already there in the modify register. We explain the procedure with an example.

Example 4.1
Consider the access sequence shown in figure 4(a). For the offset assignment given in figure 4(b), the jumps are shown in figure 4(c). For each jump, we have to change the value in the address register by the value of the jump, and each change to the address register necessitates an additional instruction. Thus if the modify register is not used, the number of additional instructions would be 6. Now, if the modify register is loaded with the value of 2 during the first jump, it could be used to change the address register for subsequent jumps of value 2. This additional operation can be subsumed during addressing as a post-modify operation, thus costing only the load instruction required to load the modify register. As a result, the number of additional instructions would be only 3. This example illustrates the potential of modify registers to reduce the number of addressing instructions. □

The sequence of values by which the address register needs to be modified is referred to as *jump sequence*, S_j. The distinct jump values are given in the set V_j. (For the example in figure 4, $S_j = \{2, 3, 2, 2, 3, 2\}$ and $V_j = \{2, 3\}$.) The problem is to find the values to be loaded into the modify register

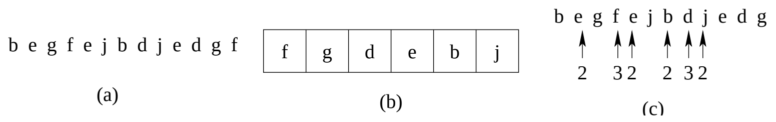

Figure 4: Example 4.1: (a) Access sequence. (b) Address assignment. (c) Jumps and jump values.

at specific times such that the number of additional instructions is minimized. Leupers and Marwedel [6] have provided a solution to the problem with worst-case complexity $O(p^2)$ where p is the length of the jump sequence. In the rest of this section, we describe a dynamic programming based solution that has a worst case complexity of $O(pq^2)$, where q is the size of V_j. Usually q is smaller than p, and the algorithm is almost linear in p.

Our solution utilizes the fact that there is an optimal substructure; if we consider the jump sequence without the last element, an optimal solution to that should be part of an optimal solution to the bigger problem. With this observation, we can reduce the problem (say P_n) to solving P_{n-1}, and then handling the additional variable. The subproblems share subsubproblems, for e.g., P_{n-1} and P_{n-2} both have P_{n-3} as the subsubproblem. This indicates that memoization will be useful. Based on these observations, we define a recursive cost function as follows:

$$\begin{aligned} C(k,i) &= \min_{m \epsilon V_j}\{C_{ind}(m,k,i)\} \text{ for } i > 1 \\ &= 1 \text{ for } i = 1 \text{ and } k = S_j(1) \\ &= 2 \text{ for } i = 1 \text{ and } k \neq S_j(1) \\ &\quad \text{where } 1 \leq i < |S_j| \text{ and } k \in V_j \qquad (1) \\ C_{ind}(m,k,i) &= C(m,i-1) + 2 \text{ if } S_j(i) \neq k \text{ AND } m \neq k \\ &= C(m,i-1) + 1 \text{ if } (S_j(i) \neq k \text{ AND } m = k) \\ &\quad \text{OR } (S_j(i) = k \text{ AND } m \neq k) \\ &= C(m,i-1) \text{ if } S_j(i) = k \text{ AND } m = k \\ &\quad \text{where } 1 \leq i < |S_j| \text{ and } m,k \in V_j \qquad (2) \end{aligned}$$

We add 2 to the cost when we have to change the modify register *and* the current jump value does not match the changed value in the modify register. We add 1 to the cost when we have to change the modify register *or* when the current jump value is not matching with the modify register value. Using this cost function, a table is constructed (memoization). The minimum cost changes to the modify register are found from the values in the table by following the values of m which gave the minimum value in equation 1.

Complexity Analysis

The table is of size pq and filling up each entry takes $O(q)$ time. The complexity of the solution is $O(pq^2)$, where p is the length of S_j and q is the size of V_j.

Results:

The modify register optimization is performed after the SOA heuristic for both random sequences and real program sequences. The results are given in Table 1. For random sequences, the use of this optimization results in an average reduction of approximately 35% compared to the case when only OFU is done (i.e. no MRO). The reduction is approximately 30% for the case where the offset assignment is obtained using either Liao's or Leupers' heuristic.

For program sequences, modify register optimization results in a reduction of 30% when the assignment is obtained using OFU. The effect of modify register optimization for Liao's or Leupers' heuristic is less, at around 17%. This is because for program sequences, the offset assignment heuristics generated good results with small jump sequences. As a result, there is limited opportunity for reduction due to modify register optimization. Finally, after the modify register optimization step, the difference in the performance between OFU and the heuristics by Liao and Leupers is not as significant.

5. ADDRESS REGISTER ASSIGNMENT

The address register assignment problem is: Given an access sequence, offset assignment, and the number of address registers, assign an address register to each access such that the number of additional instructions is minimized. The offset assignment could be generated by the SOA or GOA algorithms described in the previous sections. The problem is formulated using min-cost circulation by Gebotys [3]. The complexity of the solution depends on the algorithm used for min-cost circulation. Orlin's algorithm [9] runs in $O(m(\log n)(m + n(\log n))$ time where m is the number of edges and n is the number of nodes. Let l be the length of the access sequence. Then, for the Gebotys' formulation, $n = O(l)$ and $m = O(l^2)$. The complexity in terms of length of the access sequence is $O(l^4 \log l)$. The complexity of this optimal formulation is high. This motivated us to look for alternate formulations. Our optimal formulation is based on Minimum Cost Perfect Matching (MCPM) and has a lower complexity of $O((l + k)^3)$ where k is the number of address registers.

The MCPM transformation utilizes the observation that if the same address register is used, say R_1, to access x_i and then x_j, the number of jumps increases only if $|f_A(x_j) - f_A(x_i)| > 1$. This information can be represented in a graph $G = (V, E)$ by adding nodes x_i and x_j, an edge e_{ij} between x_i and x_j for $j > i$, and a cost on the edge, c_e which is 1 if the number of jumps increases and 0, otherwise. A path in this graph corresponds to the sequence of nodes that each address register accesses. In terms of the graph G, the problem is then to select edges such that every node has only one incoming edge selected, only one outgoing edge selected, the number of disjoint paths is less than or equal to k and the number of selected edges with cost 1 is minimum. For more details, please refer to [12].

There exist efficient algorithms for solving Minimum Cost Perfect Matching on Bipartite graphs, with even the simpler ones running at $O(n^3)$ where n is the number of nodes [2]. The complexity of this approach in terms of the length of the access sequence, l and the number of registers k is $O((l + k)^3)$. Since $k \ll l$, this is better than the previous formulation. We implemented the Minimum Cost Perfect Matching using the Hungarian Method which in a direct implementation runs in $O(n^2 m)$ but with simple changes could be made to run in $O(n^3)$. Since even the MCPM-based algorithm takes a long time for large access sequences, we have

Table 1: Number of additional instructions using SOA followed by Modify Register Optimization for random access sequences and program access sequences: $|V|$ is the number of variables and $|AS|$ is the length of the access sequence. The values are normalized with respect to OFU values before MRO.

Random sequences					Program sequences			
$|V|$	$|AS|$	OFU	Liao	Leupers	Program	OFU	Liao	Leupers
5	20	0.57	0.45	0.44	matrix_multiply	0.60	0.60	0.80
5	50	0.45	0.37	0.37	2-D FIR	0.77	0.68	0.68
7	30	0.63	0.45	0.47	LMS	0.70	0.50	0.50
7	70	0.56	0.46	0.45	iir_biquad_N_sections	0.65	0.48	0.57
10	50	0.66	0.55	0.54	adpt_predict_1	0.75	0.50	0.50
10	100	0.65	0.55	0.55	adpt_predict_2	0.74	0.57	0.57
20	50	0.77	0.62	0.64	reset_states	0.68	0.64	0.68
20	100	0.77	0.64	0.64	speed_control_2	0.77	0.70	0.61

Table 2: Number of additional instructions for the case when offset assignment was obtained using OFU and address register assignment was obtained using either Optimal formulation or heuristic algorithm. The values are normalized with respect to the Optimal value before modify register optimization. The execution times are given in the last 2 columns with values normalized with respect to the execution time of the heuristic. The number of address registers, $k = 4$.

$|V|$	$|AS|$	Number of addl. instrs.				Exec time	
		Before Modify		After Modify		After Modify	
		Opt	Heur	Opt	Heur	Opt	Heur
5	20	1	1.0	1.0	1.0	10	1
5	50	1	1.0	1.0	1.0	33	1
7	30	1	1.0	1.0	1.0	12	1
7	70	1	1.1	1.0	1.0	85	1
10	50	1	1.2	0.9	0.9	36	1
10	100	1	1.4	0.8	0.9	209	1
20	50	1	1.3	0.8	0.9	35	1
20	100	1	1.4	0.6	0.7	262	1

developed a heuristic algorithm which we describe next.

Heuristic algorithm for ARA:

The proposed heuristic greedily looks for an address register that could be assigned for each access. It reads the access sequence and stores information about the occurrences of different variables. It tries to assign to an address register that does not need a jump. If that is not possible, it assigns to the address register that requires the least amount of jump so that the jump sequence is biased towards a lower value. If two address registers are available to be assigned to an access, the occurrences list is consulted to see which one of the address registers will be needed first. The address register that is not needed immediately is used for this access.

Results

The heuristic was compared against the optimal formulation and the results are tabulated in Table 2 for random access sequences with the number of address registers, k, set to 4.. The number of instructions is normalized with the optimal value before modify. Though the heuristic performs well for smaller sequences, it does not do as well for larger sequences. However after modify register optimization, the results obtained by the heuristic comes closer to the optimal values. Since the execution time for the optimal algorithm is very high, the heuristic algorithm is a good alternate solution when used along with MRO.

6. ANALYSIS

First, we will present some of our main observations [12].

- Simple Offset assignment reduces the number of instructions, on the average, by 20% compared to OFU. The heuristics by Liao and Leupers produce similar results and have comparable computation times. GOA leads to around 60% reduction in the number of instructions with respect to OFU using $k = 4$.
- Modify Register Optimization (MRO) is a powerful kernel that results in a significant reduction in the number of instructions. For instance, when MRO is applied after Liao's SOA, the reduction in number of instructions due to MRO is approximately 55%. Similarly, when MRO is applied after Leupers' GOA, the reduction in number of instructions due to MRO is approximately 7%.
- Address Register Assignment can be thought of as another way to efficiently use the available k address registers. We have proposed an optimal formulation based on MCPM and a heuristic algorithm. For slightly large sequences, the heuristic is worse by 20-30% compared to the optimal formulation before modify register optimization. Interestingly, after modify register optimization, the heuristic is worse by only 8-13%. However, the optimal formulation is 10-200 times slower than the heuristic, depending on the length of the access sequence.

In section 2 (see figure 3) we proposed the use of three methods for address code generation. For each method, the subproblems could be solved with different algorithms. Based on our analysis, we have chosen to study the following five configurations more closely:
Configuration 1: Leupers' GOA followed by MRO,
Configuration 2.1: OFU followed by Optimal ARA followed by MRO,
Configuration 2.2: OFU followed by Heuristic ARA followed

Table 3: Number of instructions and execution times (in parentheses) for different configurations for random access sequences ($k = 4$).

$\|V\|$	$\|AS\|$	Configurations			
		1	2.1	2.2	3
5	20	0.30(1)	0.33(9.80)	0.33(0.96)	0.33(1.02)
5	50	0.14(1)	0.14(81.14)	0.14(1.13)	0.14(1.20)
7	30	0.22(1)	0.23(13.70)	0.24(1.10)	0.24(1.01)
7	70	0.10(1)	0.10(100.3)	0.10(1.18)	0.10(1.20)
10	50	0.17(1)	0.18(26.03)	0.18(0.72)	0.18(0.94)
10	100	0.09(1)	0.10(170.7)	0.11(0.82)	0.11(0.95)
20	50	0.27(1)	0.28(2.36)	0.32(0.07)	0.34(0.10)
20	100	0.21(1)	0.22(19.55)	0.25(0.07)	0.25(0.12)

Table 4: Number of instructions and execution times (in parentheses) for different configurations for program access sequences ($k = 4$).

Program	Configurations			
	1	2.1	2.2	3
matrix_mult	0.60(1)	0.60(4.00)	0.60(0.64)	0.60(0.76)
2-D FIR	0.68(1)	0.50(3.44)	0.50(0.69)	0.50(0.83)
LMS	0.60(1)	0.60(3.91)	0.80(1.02)	0.60(1.11)
iir_biquad	0.39(1)	0.39(2.12)	0.39(0.17)	0.39(0.23)
adpt_pred_1	0.50(1)	0.50(2.42)	0.50(0.58)	0.50(0.68)
adpt_pred_2	0.57(1)	0.48(1.16)	0.52(0.17)	0.52(0.23)
reset_states	0.41(1)	0.36(1.14)	0.46(0.12)	0.41(0.18)
speed_cntl_2	0.54(1)	0.54(3.70)	0.54(0.59)	0.62(0.78)

by MRO,
Configuration 3: Leupers' SOA followed by Heuristic ARA followed by MRO

Tables 3 and 4 give the number of address instructions for the four configurations. The values are normalized with respect to the OFU values before Modify Register Optimization. Execution times are given in parenthesis and are normalized with respect to Configuration 1. We can see that for random access sequences, Configuration 1 is better in most of the cases. The values obtained from Configuration 2.1 are close to the other configurations considered. This is a useful observation, since in situations where delayed offset assignment is not possible, performing ARA followed by MRO gives a good reduction in the number of instructions. For the program access sequences, Configuration 2.1 is better than the others in most cases. If we perform Address Register Assignment, there is no need for a good offset assignment algorithm. Configuration 2.1 is computationally intensive primarily due to the complexity in solving the MCPM problem. Our implementation is not optimized, but even with a sophisticated implementation it may not be as fast as the others. Configurations 2.2 can be said to be good uniformly across sequences in terms of both the number of instructions and execution time.

7. CONCLUSION

Addressing instructions can be part of upto 50% of the final code and so optimization of address generation could lead to significant improvements in code size, performance and energy consumption. In this paper, we analyzed different methods of generating addressing code for scalar variables and quantified the improvements due to different optimizations. Analysis of the results show that modify register optimization is very inexpensive and effective and should become a part of any address code generation procedure. Furthermore, while intelligent offset assignment is important, address register assignment followed by modify register optimization makes it less significant.

The access sequence plays a very important role in determining the effectiveness of these optimizations. Since the access sequence is primarily determined by the schedule, it would be interesting to see if a scheduler could attempt to obtain a "good" access sequence.

8. REFERENCES

[1] G. Araujo. *Code Generation Algorithms for DSPs.* PhD thesis, Department of Electrical Engineering, Princeton University, 1997.

[2] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* John Wiley & Sons, Inc., 1998.

[3] C. Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer Aided Design*, pages 100–103, 1997.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[5] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings of the International Symposium on Systems Synthesis*, pages 3–8, 1998.

[6] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proceedings of the International Conference on Computer Aided Design*, pages 109–112, 1996.

[7] S. Liao. *Code Generation and Optimization for Embedded DSPs.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1996.

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, 1995.

[9] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 377–387, 1988.

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

[11] The SPAM Project. SPAM project home page. `http://www.ee.princeton.edu/spam`.

[12] S. Udayanarayanan. Energy efficient code generation for DSPs. Master's thesis, Department of Electrical Engineering, Arizona State University, 2000.