Optimizing Address Assignment and Scheduling for DSPs with Multiple Functional Units

Chun Xue, Zili Shao, Qingfeng Zhuge, Bin Xiao, Meilin Liu, and Edwin H.-M. Sha

***Abstract*— DSP processors provide dedicated address generation units (AGUs) that are capable of performing address arithmetic in parallel to the main data path. Address assignment, optimization of memory layout of program variables to reduce address arithmetic instructions by taking advantage of the capabilities of AGUs, has been studied extensively for single functional unit (FU) processors. In this paper, we exploit address assignment and scheduling for multiple-FU processors. We propose an efficient address assignment and scheduling algorithm for multiple-FU processors. Experimental results show that our algorithm can greatly reduce schedule length and address operations on multiple-FU processors compared with the previous work.**

***Index Terms*— address assignment, scheduling, multiple functional units, AGU, DSP.**

I. Introduction

To satisfy ever-growing requirements of high performance DSP (Digital Signal Processing), VLIW (Very Long Instruction Word) architecture is widely adopted in high-end DSP processors. In such multiple-FU (functional unit) architecture, several instructions can be processed simultaneously. It is a challenging problem to generate high-quality code for DSP applications on multiple-FU architectures with minimal schedule length and code size. Address assignment is an optimization technique that can minimize address arithmetic instructions through optimizing the memory layout of program variables; therefore, both the schedule length and code size of an application can be reduced. Although address assignment has been studied extensively for single-FU DSP processors, little research has been done for multiple-FU architectures.

Address assignment technique reduces the number of address arithmetic instructions by using AGUs (Address Generation Units) provided by almost all DSP processors, such as TI TMS320C2x/5x/6x [7], AT&T DSP 16xx [8], etc. An AGU is a dedicated address generation unit that is capable of performing auto-increment/decrement address arithmetic in parallel to the main data path. When auto-increment/decrement is used in an instruction, the value of the address register is modified in parallel with the instruction, hence the next instruction is ready to be executed without any extra instructions. With a careful placement of variables in memory, we can reduce the total number of the address arithmetic instructions of an application, and both the schedule length and code size can be improved.

Address assignment has been studied extensively for single-FU processors. Assuming that instruction scheduling has been done, to find an optimal memory layout for program variables has been studied in [2]–[5]. Considering AGUs that can also perform auto-increment/decrement based on modified registers, various problems have been studied in [13]–[17]. Address mode selection has been studied in [20]. Optimal address register live range merge is solved in [18]. An algorithm that allows data variables to share memory locations is proposed in [22]. Considering scheduling and address assignment together, various techniques have been proposed in [10]–[12]. Experimental results comparing different algorithms are presented in [19], [21]. The goal of all these work is to minimize address operations to achieve code size reduction and performance improvement. It works well on single-FU processors. However, as shown in Section II-B, minimizing address operations alone may not directly reduce code size and schedule length for multiple-FU architectures. In this paper, we exploit the address assignment problem with scheduling for multiple-FU architectures.

The basic idea is to construct an address assignment first and then perform scheduling. In this way, we can take full advantage of the obtained address assignment and significantly reduce code size and schedule length. An algorithm, MFSchAS, is proposed in this paper to generate both address assignment and schedule for multiple-FU processors. In MFSchAS algorithm, we first obtain an address assignment and then use bipartite matching to find the best schedule based on the address assignment. Compared to list scheduling, MFSchAS shows an average reduction of 16.9% in schedule length and an average reduction of 33.9% in the number of address operations. Compared to Solve-SOA [5], MFSchAS shows an average reduction of 9.0% in schedule length and an average reduction of 8.3% in the number of address operations.

The remainder of this paper is organized as follows. Section II introduces the basic models and provides a motivational example. The algorithm is discussed in Section III. Experimental results and concluding remarks are provided in Section IV and V, respectively.

II. Models and Examples

A. Basic Models

The processor model we use is given as follows: For each FU in a multiple-FU processor, i.e. FU_i, there is an

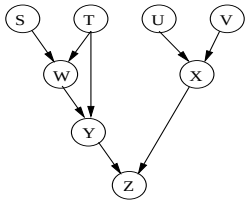
This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, USA, and HK POLYU A-PF86 and COMP 4-Z077, HK.

C. Xue, Q. Zhuge, M. Liu and E. H.-M. Sha are with the Department of Computer Science, the University of Texas at Dallas, USA. Email: {cxx016000, qfzhuge, meilin, edsha}@utdallas.edu. Z. Shao and B. Xiao are with Department of Computing, Hong Kong Polytechnic University, Hong Kong. Email: {cszshao,csbxiao}@comp.polyu.edu.hk.

accumulator and one or more address registers. Each operation involves the accumulator and another optional operand from the memory. Memory access can only occur indirectly via address registers, AR_{i0} through AR_{ik}. Furthermore, if an instruction uses AR_{ij} for indirect addressing, then in the same instruction, AR_{ij} can be post-incremented or post-decremented by one or by the value stored in the modify register (MR) without extra cost. If an address register does not point to the desired location, it may be changed by adding or subtracting a constant using the instructions ADAR and SBAR. LDAR is used to load address into address register, and ADD performs addition arithmetic. In this paper, AR_{ij} is used to denote the j_{th} AR for FU_i. For simplicity, AR_i is used in the examples to denote AR for FU_i when there is only one AR available for each FU. We use *(AR_{ij}), *(AR_{ij})+, and *(AR_{ij})- to denote indirect addressing through AR_{ij}, indirect addressing with post-increment, and indirect addressing with post-increment, respectively. This processor model reflects addressing capabilities of most DSPs and can be easily transformed into other architectures. The input of our algorithm is a DAG. A *Directed Acyclic Graph (DAG)*, $G = \langle V, E \rangle$, is a graph, where V is the node set in which each node represents a computation, and $E \subseteq V * V$ is the edge set where each edge denotes a dependency relation between two nodes.

B. Examples

In this section, we provide a motivating example. For a given DAG, we compare the schedule length and code size generated by list scheduling, Solve-SOA algorithm [5], and our algorithm.



(a)

Node	Computation
S	d = e + f
T	h = f + a
U	b = b + f
V	g = a + b
W	a = d + h
X	b = b + g
Y	e = a + h
Z	f = b + e

(b)

Fig. 1. (a) A given DAG (b) The computation in each node

The input DAG shown in Figure 1(a) is used throughout this paper. Each node in the DAG is a computation. For example, node Y denotes the computation of e = a + h. The list of nodes and computations is shown in Figure 1(b).

Assume we have two functional units in our system. Using the list scheduling that sets the priority of each node as the longest path from this node to a leaf node, we obtain the schedule shown in Figure 2(a). The address assignment is simply the alphabetical order as shown in Figure 2(b). The detailed assembly code for this schedule is shown in Figure 2(c). Each node in the schedule in 2(a) corresponds to several assembly instructions in 2(c) to complete the computation denoted by this node. For example, node S in 2(a) corresponds to assembly code from line 1 to line 5 of FU_1 in 2(c) that computes d = e + f. In this assembly code, we first load the address of variable e into address register AR_1, i.e. LDAR AR_1,&e. Then we load the value pointed by AR_1 into the accumulator of FU_1, i.e. LOAD *(AR_1)-. In this instruction, auto-decrement addressing mode, (*AR_1)-, is used to make AR_1 point to variable f. Then the value of f is added to the accumulator, i.e. ADD *(AR_1). And since the distance between d and f in the address assignment in Figure 2(b) is 2, we move AR_1 from f to d by adding 2 to it, i.e. ADAR AR_1,2. Finally, we store the result to d, i.e. STOR *(AR_1). The schedule length is 25 as shown in Figure 2(c).

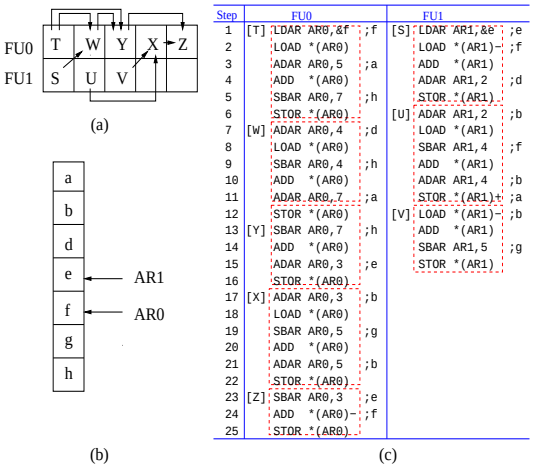


Fig. 2. The schedule obtained by List Scheduling without address assignment (schedule length=25): (a) The node-level schedule. (b) Address Assignment. (c) The assembly-code-level schedule.

Based on the schedule from Figure 2(c), Solve-SOA algorithm [5] is applied to generate a better address assignment as shown in Figure 3(b). With this new address assignment, some address arithmetic operations are saved. We obtain a new schedule with a total schedule length of 21 as shown in Figure 3(c).

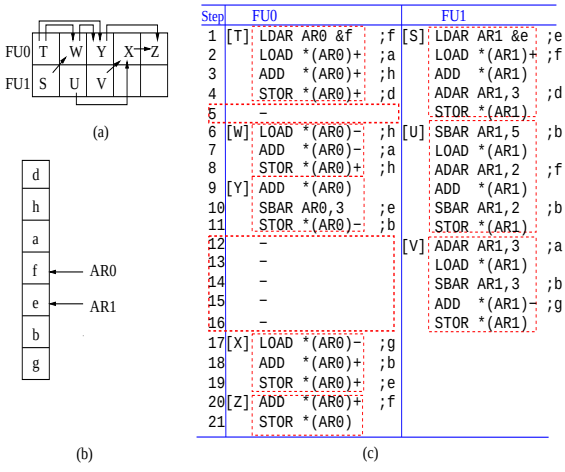


Fig. 3. The schedule obtained by List Scheduling without address assignment (schedule length=25): (a) The node-level schedule. (b) Address Assignment. (c) The assembly-code-level schedule.

From the schedule in Figure 3(c), we can see that the number of address operations (ADAR and SBAR) are reduced. However, the schedule length is not reduced as much as it could be. Even the address operations in one function unit can be saved, we may not reduce schedule length or code size because of the dependency constraints shown in the dashed boxes in Figure 3(c). This implies that we can not achieve the best result with a fixed schedule on a multiple-FU processor.

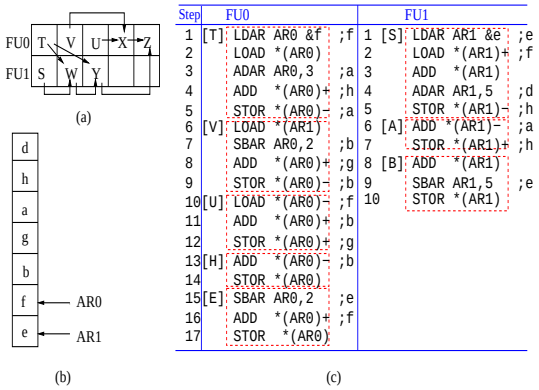


Fig. 4. The schedule obtained by our algorithm (schedule length=17): (a) The node-level schedule. (b) Address Assignment. (c) The assembly-code-level schedule.

The schedule generated by our algorithm is shown in Figure 4. With a different address assignment as shown in Figure 4(b), the schedule length is 17. In our schedule, both address operations and schedule length are reduced. Among the three schedules, the schedule generated by our algorithm has the minimal schedule length.

III. Address Assignment and Scheduling

As shown in Section II, minimizing address operations alone can not directly reduce schedule length and code size for multiple-FU processors. We use an approach that generates address assignment first and then performs scheduling based on the obtained address assignment to solve this problem. In this section, we first show how to generate a good address assignment and then propose an algorithm, MFSchAS, to minimize schedule length and code size for multiple-FU processors.

A. Address Assignment before Scheduling

The input of Solve-SOA/GOA algorithm [5] is a complete access sequence based on a fixed schedule. In our algorithm, the schedule is not known yet. However, we can obtain a partial access sequence based on the access sequence within each node. In this section, we propose an algorithm, GetAS, that improves Solve-SOA/GOA algorithm [5] so that it can handle partial access sequences. The algorithm is shown in Algorithm III.1.

Algorithm III.1 GetAS(G,k)

Input: DAG $G = \langle V, E \rangle$, number of address registers k
Output: An address assignment
  for all $u \in V$ **do**
    access_sequence += get_access_sequence(u) + " | ";
  end for
  if k=1 **then**
    Solve-SOA(access_sequence);
  else
    Solve-GOA(access_sequence, k);
  end if

Algorithm III.1 has two steps. In step 1, a partial access sequence is obtained based on the access sequence within each node. Basically, a special symbol, " | ", is inserted between the access sequences of two neighbor nodes to denote that there is no relation between the two neighbor variables. For example, " e f d | f a h | b f b " is a partial access sequence, in which "d" and "f" have no relation. A partial access sequence for a DAG contains around 60% of the total access sequence information obtained based on a fixed schedule, assuming there are 3 variable accesses in each node.

In step 2, depends on the number of address registers that are available per FU, either Solve-SOA or Solve-GOA will be called with the access sequence created in step 1. Here Solve-SOA/GOA will be modified slightly in the calculation of edge weight. If there is a " | " symbol between u and v in the partial access sequence, it means u and v are not adjacent to each other, so it will not be counted in the weight w(e) of the edge e between u and v.

For a commutative operation such as d = e + f, the access sequence can be either " e f d " or " f e d ". To exploit commutativity or associativity, we can apply Commute2-SOA by Rao and Pande [11] in place of Solve-SOA [5] algorithm. When there are modify registers available in the system, we can apply the technique presented in [14] inside algorithm III.1 to take advantage of the modify registers.

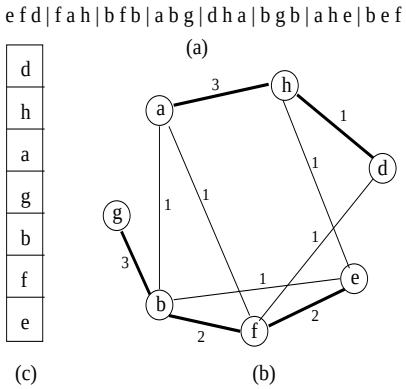


Fig. 5. (a)Access Sequence (b) Access Graph (c) Address Assignment

An example is presented in Figure 5. Given the DAG in Figure 1, a partial access sequence obtained by GetAS is shown in Figure 5(a), assuming only one address register is available per FU. For example, there is a " | " symbol between d and f since we do not know whether or not Node S (with internal access sequence " e f d ") will be scheduled in front of Node T (with internal access sequence " f a h "). Based on this partial access sequence, an access graph is constructed in Figure 5(b). A maximum weight path cover is shown in thick line in the access graph. The cost of the path cover is the sum of the weight of the edges that are not covered by the path cover, which also equals to the number of address arithmetic instructions we have to include in our schedule. For this example, the cost is 5.

B. Algorithm for Multiple-FU processors

In this section, we present an algorithm, MFSchAS, to minimize the schedule length and code size for multiple-FU processors. The basic idea is to find a matching between available functional units and ready nodes in such a way that the schedule based on this matching minimizes the total number of address operations in every scheduling step. MFSchAS algorithm is shown in Algorithm III.2.

As shown in Algorithm III.2, we first obtain an address assignment using GetAS(G,k) and then generate a schedule with minimum schedule length using weighted bipartite matching. In MFSchAS algorithm, we repeatedly create a weighted bipartite graph G_{BM} between the set of available functional units and the set of ready nodes in L_{RD}, and assign nodes based on the min-cost maximum bipartite matching M. In each scheduling step, the weighted bipartite graph, $G_{BM} =< V_{BM}, E_{BM}, W >$, is constructed as follows: $V_{BM} = FU_{ready} \cup L_{RD}$ where $FU_{ready} \subseteq \langle F_1, F_2, \cdots, F_N \rangle$ is the set of currently available FUs and L_{RD} is the set of ready nodes ; for each FU $F_i \in FU_{ready}$ and each node $u \in L_{RD}$, an edge $e(F_i, u)$ is added into E_{BM} and edge weight $W(F_i, u) = WCF(AList(F_i), First(u), Priority(u))$, where $AList(F_i)$ is the list of variables last accessed by each AR in FU F_i, $First(u)$ is the first variable that will be accessed by node u. Priority(u) is the longest path from node u to a leaf node. $WCF(AL, y, Z)$ is a weight function defined as follows:(AL is a list of variables; y is a variable in the address assignment; Z is the priority)

$$WCF(AL, y, Z) = \begin{cases} Z-2 & \text{y=x, x} \in \text{AL} \\ Z-1 & \text{y is a neighbor of x, x} \in \text{AL} \\ Z & \text{Otherwise} \end{cases}$$

In this way, the ready nodes with higher priority are considered first. Given the same priority, nodes with address operation savings have more advantage. ARs in use are preferred over unused ARs to save the initialization cost.

Algorithm III.2 MFSchAS(Multiple FU Scheduling with Address Assignment)

Input: DAG $G = \langle V, E \rangle$, FU set $\langle F_1, F_2, \cdots, F_N \rangle$, number of AR k
Output: A schedule with minimum schedule length
  Address_Assignment $\leftarrow GetAS(G, k)$;
  for each v $\in$ V **do**
    Priority$_v$ $\leftarrow$ longest_path_to_leaf(v);
  end for
  $L_{ready} \leftarrow$ get_ready_nodes(G);
  while $L_{ready} \neq \phi$ **do**
    $FU_{ready} \leftarrow$ Current available FUs ;
    Construct $G_{BM} = \langle V_{BM}, E, W \rangle$, where:
      $V_{BM} = FU_{ready} \cup L_{ready}$;
      $E = \{(F_i, u) | \forall F_i \in FU_{ready}, u \in L_{ready}\}$;
      $W(F_i, u)) = WCF(AList(F_i), First(u), Priority(u))$;
    $M \leftarrow Min\_Cost\_Bipartite\_Matching(G_{BM})$;
    for each edge $e(F_i, u) \in M$ **do**
      schedule u to F_i;
      update $AList(F_i)$;
    end for
    $L_{ready} \leftarrow$ get_ready_nodes(G);
  end while

An example is shown in Figure 6. Given the DAG in Figure 1(a), the scheduling in the second step by the MFSchAS algorithm is shown in Figure 6 when there are 2 FUs and 1 AR per FU, after nodes T and S have been scheduled to FU_0 and FU_1 in the first step. Address assignment generated using algorithm GetAS is shown in Figure 6(a). A weighted bipartite graph based on the set of available functional units and the set of ready nodes in L_{RD} is constructed in Figure 6(b). A min-cost maximum bipartite matching is shown in Figure 6(c). Applying this matching, we schedule node W to FU_1 and node V to FU_0.

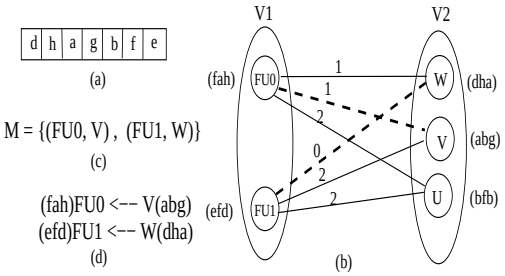


Fig. 6. (a) Address layout (b) Weighted Bipartite Graph (c) Min-cost Max Bipartite Matching (d) The schedule for second step

The technique proposed by Fredman and Tarjan [6] can be used to obtain a min-cost maximum bipartite matching in $O(n^2 \log n + nm)$, where n is the number of nodes and m is the number of edges of a bipartite graph. Let N be the number of functional units. In every scheduling step, we need at most $O(|V|^2 \log |V| + |V|^2)$ to find a minimum weight maximum bipartite matching, since the number of nodes is $N + |V|$ and the number of edges is $N * |V|$ in the bipartite graph. Thus, the complexity of MFSchAS is $O(|V|^3 \log V)$, since the scheduling step is at most $|V|$.

IV. Experiments

In this section, we experiment with our algorithm on a set of DSP benchmarks including IIR filter, IIR-UF2 (IIR filter with unfolding factor 2), IIR-UF3 (IIR filter with unfolding factor 3), 4-stage lattice filter, 8-stage lattice filter, differential equation solver, all-pole filter, elliptic filter and voltera filter. The algorithm is implemented in C language on Redhat 9 Linux.

Optimal solution for the address assignment and scheduling problem with multiple FU architecture can be computed by trying all possible schedules and applying Liao's branch-and-bound algorithm [9] for optimally solving SOA. We computed optimal solutions for several of the benchmark programs that are relatively small (With at most 16 variables), and compare with MFSchAS. This allows us to measure the quality of the proposed approach. The results are given in table I. For two of the benchmark programs, the MFSchAS achieves the same results as optimal solution. The average overhead is 2.1%.

Bench.	Optimum	MFSchAS	%Overhead
IIR	17	17	**0.0**
All-Pole	49	51	**4.1**
Diff. Eq.	16	16	**0.0**
4-stage	42	43	**2.4**
Voltera	53	55	**3.8**
Average Overhead			**2.1**

TABLE I

THE COMPARISON BETWEEN OPTIMUM AND MFSCHAS.

We compare the MFSchAS algorithm with list scheduling, and the algorithm that directly applies Solve-SOA [5] on multiple-FU processors. Tables II and III show the comparison for schedule length and for the number of address operations, respectively. In Tables II-III, the corresponding results are shown for list scheduling (Column "List"), Solve-SOA(Column "SOA") and MFSchAS

Bench.	List	SOA	MFSchAS	%LS	%SOA
The number of FUs = 3					
IIR	22	19	17	**22.7**	**10.5**
IIR-UF2	30	27	26	**13.3**	**3.7**
IIR-UF3	44	42	40	**10.0**	**4.8**
Diff. Eq.	19	19	16	**15.8**	**15.8**
All-Pole	67	54	51	**23.9**	**5.6**
4-stage	51	47	43	**15.7**	**8.5**
8-stage	94	89	83	**11.7**	**6.7**
Elliptic	83	75	72	**13.3**	**4**
Voltera	62	62	55	**11.3**	**11.3**
The number of FUs = 4					
IIR	22	18	14	**36.4**	**22.2**
IIR-UF2	22	21	20	**9.0**	**4.8**
IIR-UF3	38	35	32	**15.8**	**8.6**
Diff. Eq.	19	18	16	**15.8**	**11.1**
All-Pole	67	54	49	**26.7**	**9.3**
4-stage	49	46	39	**20.4**	**15.2**
8-stage	96	86	81	**15.6**	**5.8**
Elliptic	80	74	72	**10.0**	**2.7**
Voltera	62	58	52	**16.1**	**10.3**
Average Reduction				**16.9**	**9.0**

TABLE II

THE COMPARISON ON SCHEDULE LENGTH FOR LIST SCHEDULING, SOLVE-SOA, AND MFSCHAS.

Bench.	List	SOA	MFSchAS	%LS	%SOA
The number of FUs=3					
IIR	17	14	8	**52.9**	**42.9**
IIR-UF2	35	25	25	**28.6**	**0.0**
IIR-UF3	49	37	37	**24.5**	**0.0**
Diff. Eq.	25	16	15	**40.0**	**6.3**
All-Pole	39	26	23	**41.0**	**11.5**
4-stage	68	50	48	**29.4**	**4.0**
8-stage	114	84	84	**26.3**	**0.0**
Elliptic	95	65	65	**31.6**	**0.0**
Voltera	67	54	51	**23.9**	**5.7**
The number of FUs=4					
IIR	17	11	8	**52.9**	**27.3**
IIR-UF2	35	25	25	**28.6**	**0.0**
IIR-UF3	51	43	37	**27.5**	**14.0**
Diff. Eq.	25	16	15	**40.0**	**6.3**
All-Pole	39	26	22	**43.6**	**15.4**
4-stage	66	49	46	**30.3**	**6.1**
8-stage	114	85	85	**34.1**	**0.0**
Elliptic	96	65	65	**32.3**	**0.0**
Voltera	65	55	50	**23.1**	**9.1**
Average Reduction				**33.9**	**8.3**

TABLE III

THE COMPARISON ON ADDRESS OPERATIONS FOR MFSCHAS, SOLVE-SOA AND LIST SCHEDULING

(Column "MFSchAS") when the number of functional units equal 3 and 4, respectively, and assuming only one AR is available for each FU. Column "%LS" denotes the percentage of reduction between list scheduling and MFSchAS. Column "%SOA" denotes the percentage of reduction between Solve-SOA and MFSchAS. Compared to list scheduling, MFSchAS experimental results show an average reduction of 16.9% in schedule length and an average reduction of 33.9% in the number of address operations. Compared to the algorithm that directly applies Solve-SOA [5], MFSchAS shows an average reduction of 9.0% in schedule length and an average reduction of 8.3% in the number of address operations.

V. CONCLUSION

In this paper, we show that we can improve both performance and code size when we combine scheduling with address assignment for multiple functional units DSP processors. Specifically, we can generate an address assignment, and then utilize this address assignment during the scheduling. Hence we can minimize the number of address operations needed and significantly reduce schedule length.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. January 1986.

[2] D. H. Bartley. *Optimizing stack frame accesses for processors with restricted addressing modes*. John Wiley & Sons, Inc., 1979.

[3] C. Gebotys. Dsp address optimization using a minimum cost circulation technique. In *IEEE/ACM International conference on Computer-aided Design*, pages 100–103, November 1997.

[4] R. Leupers and P. Marwedel. Algorithm for address assignment in dsp code generation. In *IEEE/ACM International conference on Computer-aided design*, pages 109–112, November 1996.

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

[6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization alg orithms. *Journal of the ACM*, 34(3):596–615, 1987.

[7] T. Instruments. *TMS320C54x DSP Reference Set: CPU and Peripherals*. March 2001.

[8] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals: Architectures and Features*. January 1997.

[9] S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors *Ph.D. thesis, MIT*, 1996.

[10] Y. Choi and T. Kim. Address assignment combined with scheduling in dsp code generation. In *ACM IEEE DAC*, June 2002.

[11] A. Rao and S. Pande. Storage assignment using expression tree transformation to generate compact and effi cient dsp code. In *ACM SIGPLAN on Programming Language design and impl.*, 1999.

[12] J. K. Sungtaek Lim and K. Choi. Scheduling-based code size reduction in processors with indirect addressing mode. In *Intl. Symp. on Hardware/software Codesign*, pages 165–169, April 2001.

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

[14] R. Leupers and F. David. A uniform optimization technique for offset assignment problem. In *International Symposium on System Synthesis*, pages 3–8, December 1998.

[15] Y. Zhang and J. Yang. Procedural level address offset assignment of dsp applications with loops. In *IEEE Intl. Conf. on Parallel Processing*, 2003.

[16] B. Wess and M. Gotschlich. Optimal DSP memory layout generation as a quadratic assignment problem. In *IEEE Int. Symp. on Circuits and Systems*, 1997.

[17] B. Wess and M. Gotschlich. Constructing memory layouts for address generation units supporting offset 2 access. In *IEEE Intl. Conference on Acoustics, Speech, and Signal Processing*, 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 *Intl. Conf. on Compiler Construction*, 2001.

[19] R. Leupers. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. In *Intl. Conf. on Compiler Construction*, 2003.

[20] E. Eckstein and B. Scholz. Address Mode Selection In *International Symposium on Code Generation and Optimization*, 2003.

[21] B. Wess. Mini. of data address computation overhead in DSP programs. In *Kluwer Design Auto. for Embedded Systems* , 1999.

[22] D. Ottoni and G. Ottoni and G. Araujo and R. Leupers. Improving Offset Assignment through Simultaneous Variable Coalescing In *Intl. Workshop on Software and Compilers for Embedded Systems*, 2003.