OPTIMIZING DSP SCHEDULING VIA ADDRESS ASSIGNMENT WITH ARRAY AND LOOP TRANSFORMATION

Chun Xue, Zili Shao, Ying Chen, Edwin H.-M. Sha

Department of Computer Science
University of Texas at Dallas

ABSTRACT

Reducing address arithmetic instructions by optimization of address offset assignment greatly improves the performance of DSP applications. However, minimizing address operations alone may not directly reduce code size and schedule length for multiple functional units DSPs. In this paper, we exploit address assignment and scheduling for application with loops on multiple functional units DSPs. Array transformation is used in our approach to leverage the indirect addressing modes provided by most of the DSP architectures. An algorithm, Address Instruction Reduction Loop Scheduling (AIRLS), is proposed. The algorithm utilizes the techniques of rotation scheduling, address assignment and array transformation to minimize both address instructions and schedule length. Compared to the list scheduling, AIRLS shows an average reduction of 35.4% in schedule length and an average reduction of 38.3% in address instructions. Compared to the rotation scheduling, AIRLS shows an average reduction of 19.2% in schedule length and 39.5% in the number of address instructions.

1. INTRODUCTION

DSP processors generally provide dedicated address generation units (AGUs) that can perform address computations in parallel to the central data path. As a result, when we access data in register-indirect addressing mode, the address stored in the address register (AR) can be auto-incremented or auto-decremented without extra addressing instruction. If the address of the next variable could be reached by auto-increment or auto-decrement, the next instruction can be executed without additional address arithmetic instruction. Consequently, contrary to the traditional compilers, DSP compilers can carefully determine the relative location of data in memory and achieve compacted object code size and improved performance. Loops are the most critical sections in many computation-intensive DSP applications. An efficient loop scheduling can help reduce both the schedule length and code size. In this paper, we develop a scheme to exploit address assignment and scheduling for application with loops on multiple functional units DSPs.

Recently, a lot of research has been done to optimize the address assignment of variables to minimize the total number of address arithmetic instructions. The address assignment was first studied in [1, 2]. More research [3] has been done on address assignment with fixed scheduling on single functional unit architectures. Some work [4] has been done on combining scheduling and address assignment in code generation. These algorithms only target single functional unit and can not be directly applied to multiple functional units.

This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, and NSF CCR-0309461, USA.

This paper proposes an algorithm, Address Instruction Reduction Loop Scheduling (AIRLS), to minimize both address instructions and schedule length for loop applications with array transformation. In the AIRLS algorithm, the schedules are generated by repeatedly rotating down and re-allocating nodes with minimum address instructions based on rotation scheduling [5], and a best schedule is selected that has the minimum schedule length. During each step of the rotation, we generate a transformed array sequence based on the result from the address assignment algorithm. Then, this array sequence is used to determine the new location for each rotating node. AIRLS shows significant performance improvement. Compared to the list scheduling, the average reduction in schedule length is 35.4% and the average reduction in address instructions is 38.3%. Compared to the rotation scheduling, the average reduction in schedule length is 13.6% and the average reduction in address instruction is 38.3%. When the unfolding technique is applied with an unfolding factor of 2, the average reduction in schedule length is increased to 24.8% and the average reduction in address instructions is increased to 40.7%.

The remainder of this paper is organized as follows. Section 2 provides a motivating example. Section 3 introduces the basic concepts and the architecture model. The algorithm is discussed in Section 4. Experimental results and concluding remarks are provided in Section 5 and 6, respectively.

2. MOTIVATING EXAMPLES

In this section, we provide a motivating example based on the loop in Figure 1(a). The DFG for the loop is shown in Figure 1(b). A schedule generated by list scheduling for the DFG is shown in Figure 1(c) when there are three functional units. The retimed graphs and schedules after the first and second rotation are shown in Figure 2(a) and Figure 2(b) respectively, which are based on the original schedule.

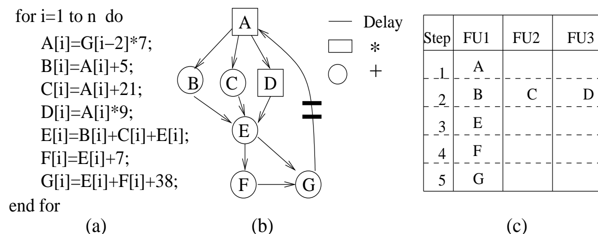


Fig. 1. A loop, its corresponding DFG and a static schedule.

We compare the schedule length based on the traditional rotation scheduling, and the schedule length after apply array transformation and address assignment to the traditional rotation scheduling. The detail schedule shown in Figure 3 is based on the node

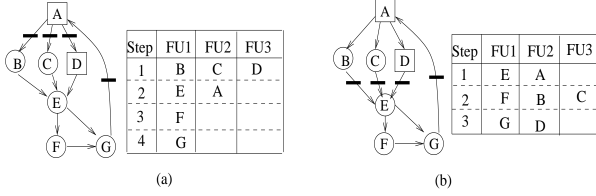
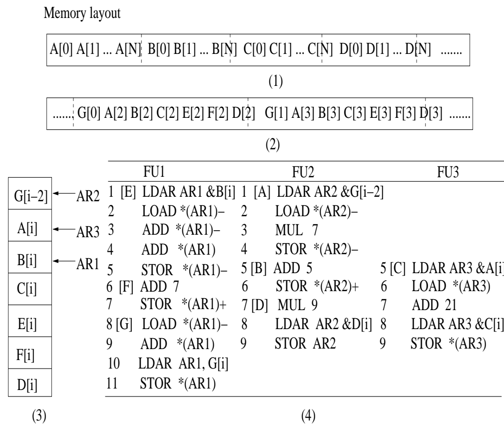


Fig. 2. The retimed graph and the schedule after (a) the first rotation. (b) the second rotation.

level schedule obtained by the traditional rotation scheduling from Figure 2(b). In this detail schedule, each node is expanded into assembly level codes. In our architecture model, we assume there is only one address register available, and no other register is available to each accumulator. Under this constraint, and with the traditional array layout, it is very difficult to leverage the indirect addressing modes. However, if we apply array transformation technique, we can leverage indirect addressing modes provided by most of the DSPs. For example, we transform the array data according to figure 4(2). Figure 4(1) shows the traditional sequential memory layout for arrays. Figure 4(3) denotes the array transformation in the 4(2), which is used as our notation in this paper for array data transformation memory layout. With this array layout, we obtain a new detail schedule as figure 4(4). We can see that the schedule length has been reduced from 17 to 11, and number of address instructions has been reduced from 14 to 5.

FU1	FU2	FU3
1 [E] LDAR AR1 &B[i]	1 [A] LDAR AR2 &G[i–2]	
2 LOAD *(AR1)	2 LOAD *(AR2)	
3 LDAR AR1 &C[i]	3 MUL 7	
4 ADD *(AR1)	4 LDAR AR2 &A[i]	
5 LDAR AR1 &E[i]	5 STOR *(AR2)	
6 ADD *(AR1)	6 [B] ADD 5	6 [C] LDAR AR3 &A[i]
7 STOR *(AR1)	7 LDAR AR2 &B[i]	7 LOAD *(AR3)
8 [F] ADD 7	8 STOR *(AR2)	8 ADD 21
9 LDAR AR1 &F[i]	9 [D] LDAR AR2 &A[i]	9 LDAR AR3 &C[i]
10 STOR *(AR1)	10 MUL 9	10 STOR *(AR3)
11 [G] LDAR AR1 &E[i]	11 LDAR AR2 &D[i]	
12 LOAD *(AR1)	12 STOR *(AR2)	
13 LDAR AR1 &F[i]		
14 ADD *(AR1)		
15 ADD 38		
16 LDAR AR1 &G[i]		
17 STOR *(AR1)		

Fig. 3. Detail Schedule after Rotation Scheduling



FU1	FU2	FU3
1 [E] LDAR AR1 &B[i]	1 [A] LDAR AR2 &G[i–2]	
2 LOAD *(AR1)–	2 LOAD *(AR2)–	
3 ADD *(AR1)–	3 MUL 7	
4 ADD *(AR1)	4 STOR *(AR2)–	
5 STOR *(AR1)–	5 [B] ADD 5	5 [C] LDAR AR3 &A[i]
6 [F] ADD 7	6 STOR *(AR2)+	6 LOAD *(AR3)
7 STOR *(AR1)+	7 [D] MUL 9	7 ADD 21
8 [G] LOAD *(AR1)–	8 LDAR AR2 &D[i]	8 LDAR AR3 &C[i]
9 ADD *(AR1)	9 STOR AR2	9 STOR *(AR3)
10 LDAR AR1, G[i]		
11 STOR *(AR1)		

Fig. 4. (1) Traditional sequential memory layout (2) Array Transformation memory layout (3)Notation for array transformation in this paper (4) Detail Schedule after Rotation Scheduling based on array transformation memory layout

3. BASIC CONCEPTS AND MODELS

The **processor model** we use in this paper is given as follows. For each functional unit in a multiple functional units processor, there is an accumulator and an address register. Each operation involves the accumulator and, if any, another operand from the memory. Memory access can only occur indirectly via address registers, AR0 through ARk. Furthermore, if an instruction uses ARi for indirect addressing, then in the same instruction, ARi can be optionally post-incremented or post-decremented by one 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. In this paper, FNi is used to denote functional unit i, and ARi is used to denote the address register for FNi. We also use *(ARi), *(ARi)+, and *(ARi)- to denote indirect addressing through ARi, indirect addressing with post-increment, and indirect addressing with post-decrement, respectively. This processor model reflects addressing capabilities of most DSPs, and can be easily transformed into other architectures.

Data Flow Graph is used to model loops and is defined as follows. A *Data Flow Graph (DFG)* $G = \langle V, E, OP, d \rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of operation nodes, $E \subseteq V * V$ is the edge set that defines the precedence relations for all nodes in V, $OP(u)$ is a binary string associated with each node $u \in V$, $d(e)$ represents the number of delays for an edge e. Nodes in V can be various operations, such as addition, subtraction, multiplication, logic operation, etc.

In our case, a DFG can contain cycles. The intra-iteration precedence relation is represented by the edge without delay and the inter-iteration precedence relation is represented by the edge with delays. The *cycle period* of a DFG corresponds to the minimum schedule length of one iteration of the loop when there are no resource constraints.

An example is shown in Figure 1. The DFG in Figure 1(b) models the loop in Figure 1(a). In this example, there are two kinds of operations: multiplication and addition. They are denoted by the rectangle and circle as shown in Figure 1(b).

A static schedule of a cyclic DFG is a repeated pattern of an execution of the corresponding loop. In our work, a schedule implies both control step assignment, and functional unit allocation. A static schedule must obey the precedence relations of the *directed acyclic graph (DAG)* portion of the respective DFG. The DAG is obtained by removing all edges with delays in the DFG. Figure 1(c) shows a static schedule for the DFG in Figure 1(b) when there are three FUs. The schedule is obtained by list scheduling. We use $[i, j]$ to denote the location of a node in a schedule, where i is the row (control step) and j is the column (FU). For example, location $[2, 1]$ in the schedule refers to node B scheduled at control step 2 and assigned to FU_1 in Figure 1(c).

Unfolding is also called unrolling or unwinding, is widely used in compiler design [6]. A schedule of unfolding factor f can be obtained by unfolding G f times. That is, a total of f iterations are scheduled together, and the schedule is repeated every f iterations.

Retiming [7] can be used to optimize the cycle period of a DFG by evenly distributing the delays in it. Given a DFG $G = \langle V, E, OP, d \rangle$, retiming r of G is a function from V to integers. For a node $u \in V$, the value of $r(u)$ is the number of delays drawn from each of its incoming edges of node u and pushed to all of its outgoing edges. Let $G_r = \langle V, E, OP, d_r \rangle$ denote the retimed

graph of G with retiming r, then $d_r(e) = d(e) + r(u) - r(v)$ for every edge $e(u \to v) \in V$ in G_r.

Rotation Scheduling presented in [5] is a scheduling technique used to optimize a loop schedule with resource constraints. It transforms a schedule to a more compact one iteratively. In most cases, the node level minimal schedule length can be obtained in polynomial time by rotation scheduling. In each step of rotation, nodes in the first row of the schedule are rotated down. By doing so, the nodes in the first row are rescheduled to the earliest possible available locations. From retiming point of view, each node gets retimed once by drawing one delay from each of incoming edges of the node and adding one delay to each of its outgoing edges in the DFG. The new location of the node in the schedule must also obey the precedence relation in the new retimed graph. The retimed graphs and schedules after the first and second rotation are shown in Figure 2(a) and Figure 2(b) respectively, which is based on the original schedule in Figure 1(c). The node level minimal schedule length is obtained by the schedule in Figure 2(b).

4. THE AIRLS ALGORITHM

In this section, an algorithm, Address Instruction Reduction Loop Scheduling (AIRLS), is designed to reduce address operations based on loop unrolling and rotation scheduling. The basic idea is to first unroll the loop, then generate the schedules by repeatedly rotating down and re-allocating nodes with minimum schedule length and address operations based on Rotation Scheduling, and then select a best schedule that has the minimal schedule length. The AIRLS algorithm is shown in Algorithm 4.1.

We use a function mSOA()[8] in the AIRLS algorithm, which is a modified version of the original Solve-SOA algorithm [2] so that it can handle partial access sequence, and also handle variables from the same array. In the original Solve-SOA algorithm, an edge is not selected if any node has a degree > 2 or if the edge causes a cycle. In mSOA(), we add one more condition. An edge is not selected if any node belongs to an array that is already selected f times, where f is the loop unrolling factor. For example, if f = 2 and both A[i-1] and A[i] is already selected, we will not include any edge that has A[i+1] or A[i+2].

In the AIRLS algorithm, we first put all nodes in the first row of S into set R. Then we delete the first row of S and shift S up by one control step. Variable L is used to record the schedule length of S. After that, we retime each node $u \in R$ such that $r(u) \leftarrow r(u) + 1$. After the retiming, the computation within each node is updated as well. For example, after we retime node B, the computation is change from B[i]=A[i]+5 into B[i+1]=A[i+1]+5. With this updated computation, we will generate a new array transformation assignment based on the modified Solve-SOA algorithm, mSOA(). Then, based on the precedence relation in the retimed graph G_r and the array transformation assignment from mSOA(), we rotate each node $u \in R$ by putting u into the location with the minimum address operation among all available empty locations in T, where T is the set containing all available locations of u.

We obtain the best location for a rotated node by the following strategy. For a location $[i, j] \in T$, we define a function, $Address_Location(u, [i, j])$, to compute the address operation if u is assigned to location $[i, j]$. Assume that u' is the node in the first non-empty location above $[i, j]$ and u'' is the node in the first non-empty location below $[i, j]$ both in column j of S, then $Address_Location(u, [i, j]) = AD(Last_V ar(u'), First_v ar(u)) + AD(last_V ar(u), First_V ar(u''))$, where $AD(x, y)$ represents the number of address operations between x and y. It is defined as follows:

$$AD(x, y) = \begin{cases} 0 & \text{the distance between x and y} = 0 \\ 1 & \text{the distance between x and y} = 1 \\ 2 & \text{Otherwise} \end{cases}$$

When computing T, the available locations from row 1 to row L are considered first. If there is no available locations in this field, we assign the node to the locations in row $L + 1$. Using this strategy, the schedule length is minimized in the first priority. After all nodes in R are scheduled, the schedule S and the retiming r are recorded. $AIRLS$ will repeat the above procedure N times, where N is a user specified amount. A best schedule is selected from the N generated schedules, which has the minimum schedule length and the minimum number of address instructions.

Let M be the number of functional units and n be the number of nodes in G. The number of nodes in a row of a schedule is at most M and the total number of empty locations is at most $M * (n - 1)$. Considering the rotation times N, the complexity of the AIRLS algorithm is $O(N * M * M * (n - 1)) = O(N * M^2 * n)$.

Algorithm 4.1 Address-Instruction-Reduction-Loop-Scheduling (AIRLS)

Require: DFG $G = \langle V, E, OP, d \rangle$, the unrolling factor f, the retiming r of G, the computation c of G, the rotation times N
Ensure: A schedule S and the retiming r
  $S \leftarrow$ Unroll G with factor f;
  for all k=1 to N **do**
    $R \leftarrow$ All nodes in the first row in S;
    Delete the first row from S;
    Shift S up by 1 control step;
    for all $u \in R$ **do**
      $r(u) \leftarrow r(u) + 1$;
      Update_computation(u);
    end for
    /*Generate a new address assignment based on updated computation */
    mSOA(G,V) ;
    for all $u \in R$ **do**
      $T \leftarrow$ All available locations of u from Row 1 to Row L in S based on the precedence relation in G_r;
      if $E = \emptyset$ **then**
        $T \leftarrow$ All available locations of u in Row $L + 1$ in S;
      end if
      $[a, b] \leftarrow$ The location with the minimum address operation among all locations in T;
      Put u into $[a, b]$;
    end for
    if $Schedule_length(S) < min_schedule_length$ **then**
      $min_schedule_length = Schedule_length(s)$;
      $S_m \leftarrow S$; $r_m \leftarrow r$;
    end if
  end for
  Output S_m and r_m;

5. EXPERIMENTS

In this section, we conduct experiments with the AIRLS algorithm on a set of benchmarks programs including 4-stage lattice filter, 8-stage lattice filter, differential equation solver, elliptic filter and voltera filter. The experiments are performed on a simulator with

Bench.	List		Rotation		AIRLS						AIRLS (Unfolding factor=2)					
	SL	AI	SL	AI	SL	%SL-L	%SL-R	AI	%AI-L	%AI-R	SL/2	%SL-L	%SL-R	AI/2	%AI-L	%AI-R
The number of FUs = 4																
iir	24	24	12	24	12	50.0%	0.0%	13	45.8%	45.8%	11	54.2%	8.3%	13	45.8%	45.8%
Voltera	69	75	69	75	56	18.8%	18.8%	57	24.0%	24.0%	56	18.8%	18.8%	55	26.7%	26.7%
4-Latt.	53	74	41	74	34	35.8%	17.1%	47	34.7%	34.7%	33	37.7%	2.9%	49	31.9%	31.9%
8-Latt.	102	125	66	125	53	48.0%	19.7%	69	44.8%	44.8%	53	48.0%	19.7%	69	44.8%	44.8%
Diff2	22	28	22	28	17	2.27%	22.7%	16	42.9%	42.9%	16	27.3%	27.3%	16	42.9%	42.9%
Ellip	84	101	78	101	58	31.0%	25.6%	56	44.6%	44.6%	56	33.3%	28.2%	56	44.6%	44.6%
Allpole	71	43	36	43	36	49.3%	0.0%	21	51.2%	51.2%	20	71.8%	44.4%	25	41.9%	41.9%
The number of FUs = 6																
iir	24	24	12	24	12	50.0%	0.0%	13	45.8%	45.8%	8	66.7%	33.3%	13	45.8%	45.8%
Voltera	69	75	69	75	56	18.8%	18.8%	55	26.7%	26.7%	56	18.8%	18.8%	55	26.7%	26.7%
4-Latt.	53	74	30	74	24	54.7%	20.0%	45	39.2%	39.2%	24	54.7%	20.0%	45	39.2%	39.2%
8-Latt.	102	125	42	125	42	58.8%	0.0%	72	42.4%	42.4%	35	65.7%	16.7%	65	48.0%	48.0%
Diff2	22	28	22	28	17	22.7%	22.7%	16	42.9%	42.9%	16	27.3%	27.3%	16	42.9%	42.9%
Ellip	84	101	78	101	55	34.5%	25.6%	56	44.6%	44.6%	54	35.7%	30.8%	56	44.6%	44.6%
Allpole	71	43	36	43	36	49.3%	0.0%	21	51.2%	51.2%	18	74.6%	50.0%	24	44.2%	44.2%
Average Reduction (%)						**35.4%**	**13.6%**	–	**38.3%**	**38.3%**	–	**45.3%**	**24.8%**	–	**40.7%**	**40.7%**

Table 1. The comparison of schedule length and address operation for rotation scheduling and AIRLS

the similar architecture as TI C6000 DSP. We compare our results with those from the traditional list scheduling and rotation algorithm. The experiments are performed on a PC with a P4 2.1 G processor and 512 MB memory running Red Hat Linux 9.0. In the experiments, the running time of AIRLS on each benchmark is less than one minute.

The experimental results for the list scheduling, the rotation scheduling and the AIRLS algorithm, are shown in Table 1 when the number of FUs is 4 and 6 respectively. Column "AI" presents the number of address instructions and Column "SL" presents the schedule length obtained from the three different scheduling algorithms: the list scheduling (Field "List"), the traditional rotation scheduling (Field "Rotation"), and our AIRLS algorithm (Field "AIRLS"). Field "AIRLS(Unfolding factor=2)" denotes the data obtained by AIRLS with each benchmarks unfolded by 2. Column "%SL-L" and "%SL-R" under "AIRLS" represent the percentage of reduction in schedule length compared with list scheduling and rotation scheduling respectively. Column "SL/2" and "AI/2" denotes the average schedule length and address instructions considering two loop iterations are processed at the same time. Column "%AI-L" and "%AI-R" under "AIRLS" represent the percentage of reduction in number of address instructions compared with list scheduling and rotation scheduling respectively.

Compared to the list scheduling, the reduction in schedule length is 35.4% and the reduction in address instructions is 38.3%. Compared to the rotation scheduling, the reduction in schedule length is 13.6% and the reduction in address instructions is 38.3%. When we apply unfolding technique with a unfolding factor of 2, the average reduction in schedule length and number of address instructions are both increased. Compared to the list scheduling, the reduction in schedule length become 45.3% and the reduction in address instructions become 40.7%. Compared to the rotation scheduling, the reduction in schedule length become 24.8% and the reduction in address instructions become 40.7%. In summary, we found that AIRLS can reduce both schedule length and the number of address instructions compared to the list scheduling and the rotation scheduling.

6. CONCLUSION

Loops are the most critical sections for DSP applications. Minimizing the schedule length and reducing code size for loops can significantly increase performance for computation-intensive DSP applications. In this paper, we proposed an algorithm, AIRLS, that utilizes array transformation, address assignment, and rotation scheduling techniques to reduce schedule length and address operations for loops on multiple functional units DSPs. AIRLS can significantly reduce schedule length and address instructions comparing to the previous work.

7. REFERENCES

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

[2] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang, "Storage assignment to decrease code size," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, pp. 235–253, May 1996.

[3] Rainer Leupers and Peter Marwedel, "Algorithm for address assignment in dsp code generation," in *IEEE/ACM International conference on Computer-aided design*, November 1996, pp. 109–112.

[4] Yoonseo Choi and Taewhan Kim, "Address assignment combined with scheduling in dsp code generation," in *ACM IEEE Design Automation Conference*, June 2002, pp. 225–230.

[5] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Trans. on Computer-Aided Design*, vol. 16, no. 3, pp. 229–239, March 1997.

[6] A. Aiken and A. Nicolau, "Optimal loop parallelization," *ACM Conference on Programming Language Design and Implementation*, pp. 308–317, 1988.

[7] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[8] Chun Xue, Zili Shao, Edwin Sha, and Bin Xiao, "Optimizing address assignment for scheduling embedded dsps," *International Conference Embedded and Ubiquitous Computing (EUC)*, pp. 64–73, 2004.