A Transformation-Based Approach for Storage Optimization*

Wei-Kai Cheng   Youn-Long Lin
Department of Computer Science, Tsing Hua University
Hsin-Chu, Taiwan 30043, R.O.C.
E-mail: {dr808309, ylin}@cs.nthu.edu.tw

Abstract

High-level synthesis (HLS) has been successfully targeted towards the digital signal processing (DSP) domain. Both application-specific integrated circuits (ASICs) and application-specific instruction-set processor (ASIPs) have been frequently designed using the HLS approach. Since most ASIP and DSP processors provide multiple addressing modes, and, in addition to classical constraint on the number of function units, registers, and buses, there are many resource usage rules, special considerations need to be paid to the optimizing code generation problem. In this paper we propose three transformation techniques, data management, data ordering, *and* transformational retiming, *for storage optimization during code generation. With these transformations, some scheduling bottlenecks are eliminated, redundant instructions removed, and multiple operations mapped onto a single one. The proposed transformations have been implemented in a software system called Theda.MS. A set of benchmark programs has been used to evaluate the effectiveness of Theda.MS. Measurement on the synthesized codes targeted towards the TI-TMS320C40 DSP processor shows that the proposed approach is indeed very effective.*

1 Introduction

For both ASIP and programmable DSP processor, the code generation problem is very important as it significantly affects the overall performance. High-level synthesis has been successfully targeted towards the DSP domain. In Paulin's survey [1] regarding future CAD requirements by a group of designers, code generation and microcode synthesis are among the most urgent needs in high-level synthesis for DSP applications. It is estimated that about four fifths of the DSP applications are implemented with either programmable DSP processors or ASIP.

There have been several systems targeted towards the code generation and microcode synthesis of DSP applications. For example, Ptolemy [3], FlexWare [1],and Cathedral II [4]are all well known systems. Beyond operation scheduling and register-for-variable allocation, there are active researches on the transformation techniques for further improving the results of synthesis or microcode generation [2] [5] [6] [7] [8] [9].

Since storage (memory or register) synthesis is very important to code generation. In this paper, we propose three transformation techniques – *data management*, *data ordering*, and *transformational retiming* – targeted towards programmable DSP processors or ASIP with non load/store architecture. *Data management* reduces the number of violations against the constraint on the number of memory and register accesses when multiple instructions are scheduled to executed simultaneously. It also eliminates redundant load/store instructions. *Data ordering* rearranges instruction execution order to minimize the dependence caused by the usage of indirect memory addressing mode. It also balances as much as possible the data flow graph and, hence, minimizes the critical path length. *Transformational retiming* combines some addressing mode transformation techniques and some retiming techniques together to eliminate the violations against addressing mode constraints, and merges multiple instructions into more inclusive one.

The rest of this paper is organized as follows. The basic ideas are first introduced in section 2. Section 3 gives an overview of the software system that implementing the proposed transformations. Detailed transformation algorithms are described in section 4. In section 5, some experiment results are presented. Finally, in section 6 we draw a conclusion and speculate on possible future research.

2 Basic Ideas

2.1 Data Management

Data management determines the best storage location (register or memory) for each operand. Many DSP processors are non-load/store architecture. There is usually register access constraint when multiple instructions are executed simultaneously. Therefore, it would be better for certain data to be kept in the memory rather than the register. We use the *data management* technique to determine where to store the data. It is different from code spilling, which temporarily stores data into memory while running out of registers.

We illustrate the idea using the example depicted in Figure 1. Figure 1(a) and (b) are two code segments with different data locations for the same function. Their corresponding DFG are shown in Figure 1(c) and (d), respectively. Assume that there can be two memory reads and two register reads during an instruction cycle. In Figure 1(a), instructions 1 and 4 load data into registers 10 and 2, respectively, for later usage by instructions 2, 5, 7, and 9. In contrast, in Figure 1(b), no preloading has

*Supported in part by a grant from the National Science Council of R.O.C. under contract no. NSC84-2215-E-007-045

32nd ACM/IEEE Design Automation Conference ®
Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. © 1995 ACM 0-89791-756-1/95/0006 $3.50

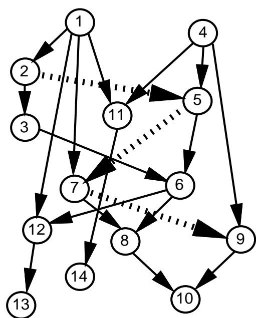
```
1   LDF    * AR4 , R10
2   MPYF   * AR2++ , R10 , R6
3   SUBF   R6 , R0
4   LDF    * AR5 , R2
5   MPYF   * AR2++ , R2 , R1
6   SUBF   R1 , R0 , R3
7   MPYF   * AR2++ , R10 , R5
8   ADDF   R3 , R5
9   MPYF   * AR2++ , R2 , R7
10  ADDF   R7 , R5
11  STOF   R10 , * AR5++
12  STOF   R3 , * AR4++
13  ADDI   1 , AR4
14  ADDI   1 , AR5
```

(a)

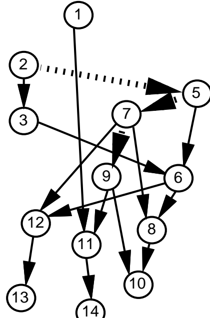
```
1   LDF    * AR4 , R10
2   MPYF   * AR2++ , * AR4 , R6
3   SUBF   R6 , R0

5   MPYF   * AR2++ , * AR5 , R1
6   SUBF   R1 , R0 , R3
7   MPYF   * AR2++ , * AR4 , R5
8   ADDF   R3 , R5
9   MPYF   * AR2++ , * AR5 , R7
10  ADDF   R7 , R5
11  STOF   R10 , * AR5++
12  STOF   R3 , * AR4++
13  ADDI   1 , AR4
14  ADDI   1 , AR5
```

(b)



(c)



(d)

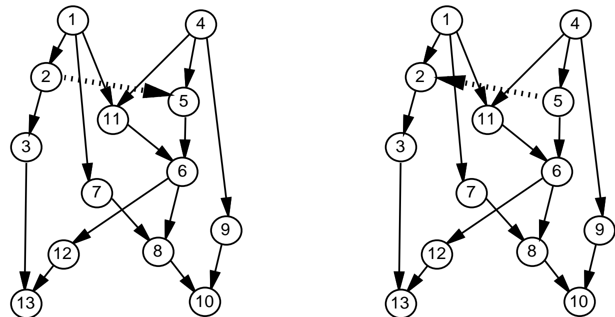
Figure 1: Effect of data management: (a) A code segment, (b) A code segment with different data locations for the same function as that of (a), (c) DFG for (a), (d) DFG for (b).

been performed and, hence, instructions 2, 5, 7, and 9 fetch data directly from the memory. For the later case, instructions 3 and 5 can be executed in parallel; While in the former case, this parallelism is not available because of too many register access during the same instruction cycle. As it has less violation against the register access constraint, the code in Figure 1(b) is better from the data management point of view.

2.2 Data Ordering

Data ordering minimizes the occurence of addressing dependences using addressing mode transformation. In many DSP application programs, the array structure is often used for storing data in the memory while instructions in the loop body usually access these data with indirect addressing mode. This results in a type of dependence which we called *addressing dependence.*

The occurrence of *addressing dependence* is also illustrated using Figure 1. In Figure 1(c) and (d), there are dashed lines between instructions 2 and 5, 5 and 7, and 7 and 9 denoting addressing dependence. These execution order constraints are imposed because the first operands of all four instruction are placed in successive memory locations and all referred to with addressing register AR2 using the *auto increasing* mode, where $*AR2{+}{+}$ denotes $addr = AR2, AR2 = AR2 + 1$. By minimizing the occurrence of *addressing dependence* via addressing mode transformation, the scheduling freedom can be increased. For example, in Figure 2 (a) the critical path length is 6; while if we change the execution order of instruction (2, 5) to (5, 2) as in Figure 2 (b), the critical path length becomes 5.



(a) (b)

Figure 2: (a) A DFG; (b) A DFG obtained from (a) with data ordering.

2.3 Transformational Retiming

The *transformational retiming* performs retiming followed by addressing mode transformation. Its basic idea is illustrated in Figure 3. In the loop body of the iir algorithm depicted in Figure 3(a), instructions 13 and 14 depend on instructions 1 and 4, respectively. If instructions 13 and 14 are retimed to the next iteration of the loop body as shown in Figure 3(b), new data dependences will be formed between instructions 13 and 14 and instructions 1 and 4, respectively. By means of addressing mode transformation, instructions 1 and 4 can be merged with instructions 13 and 14, respectively, to form new instructions using auto increasing mode as shown in Figure 3(c). This results in fewer instruction count.

3 System Overview

Our software system implementing the proposed ideas is called $Theda.MS$. Input to $Theda.MS$ is a data flow graph (DFG) generated by the compiler front end, the instruction set specification, and the addressing pattern transformation library. Output from $Theda.MS$ is an assembly code for the target DSP processor or ASIP. Its essential tasks include memory access and addressing mode transformation, instruction scheduling, and register allocation.

A typical DFG representation is shown in Figure 1(c). The DFG is similar to the traditional DFG with the addition of the *addressing dependence*(denoted by the dashed lines).

The instruction set specification describes the constraint of each instruction, including the instruction format, the legal addressing mode, and the accessible registers. It also specifies the restrictions on how multiple operations can be combined into an instruction word.

The addressing pattern transformation library provides the transformation pattern for *data ordering* and *transformational retiming* as shown in Figure 4. For *data ordering*, it provides patterns for changing the instruction execution order from that on the left hand side to that on the right hand side. For *transformational retiming*, it provides patterns that combines multiple opera-

1	LDF	* AR4 , R10
2	MPYF	* AR2++ , R10 , R0
3	SUBF	R0 , R9
4	LDF	* AR5 , R2
5	MPYF	* AR2++ , R2 , R3
6	SUBF	R3 , R9 , R3
7	MPYF	* AR2++ , R10 , R9
8	ADDF	R3 , R9
9	MPYF	* AR2++ , R2 , R0
10	ADDF	R0 , R9
11	STOF	R10 , * AR5++
12	STOF	R3 , * AR4++
13	ADDI	1 , AR4
14	ADDI	1 , AR5

(a)

13	ADDI	1 , AR4
14	ADDI	1 , AR5
1	LDF	* AR4 , R10
2	MPYF	* AR2++ , R10 , R0
3	SUBF	R0 , R9
4	LDF	* AR5 , R2
5	MPYF	* AR2++ , R2 , R3
6	SUBF	R3 , R9 , R3
7	MPYF	* AR2++ , R10 , R9
8	ADDF	R3 , R9
9	MPYF	* AR2++ , R2 , R0
10	ADDF	R0 , R9
11	STOF	R10 , * AR5++
12	STOF	R3 , * AR4++

(b)

1	LDF	*++ AR4 , R10
2	MPYF	* AR2++ , R10 , R0
3	SUBF	R0 , R9
4	LDF	*++ AR5 , R2
5	MPYF	* AR2++ , R2 , R3
6	SUBF	R3 , R9 , R3
7	MPYF	* AR2++ , R10 , R9
8	ADDF	R3 , R9
9	MPYF	* AR2++ , R2 , R0
10	ADDF	R0 , R9
11	STOF	R10 , * AR5++
12	STOF	R3 , * AR4++

(c)

Figure 3: A Transformational Retiming Example: (a) a loop body, (b) the retimed loop body, (c) the final loop body after addressing mode transformation.

tions into one operation as the first example shown, or some other transformations (the second example).

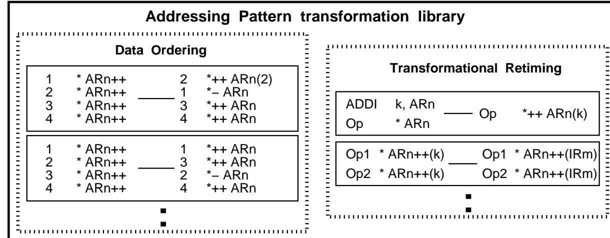


Figure 4: The addressing pattern transformation library.

The transformation of memory access and addressing mode utilizes all three transformation techniques. First, the *data ordering* transformation is invoked to break the addressing dependences, balance the DFG, and, possibly, reduce the critical path length. Then the *transformational retiming* transformation is applied to eliminate violations against addressing mode constraints and to combine multiple instructions into a single one. Finally, the *data management* transformation is applied to eliminate violations against register and memory access constraints and to remove redundant load/store instructions.

For instruction scheduling and register allocation, we apply the algorithm of [10]. The algorithm schedules operations into control steps, minimizes the pipeline conflicts, binds data and address values to registers, and allocates registers for loop counters and other special-purpose usage.

4 Algorithms

4.1 Data Management

Figure 5 gives a pseudo code description of the data management algorithm. The algorithm modifies a DFG in two phases. During the first phase, it determines whether an operand should be brought into the register before it is accessed. For each operand, the evaluation functions *evaluate_r* and *evaluate_m* are used to calculate the gains of fetching the operand directly from the memory and loading it into the register beforehand, respectively. If *evaluate_m* > *evaluate_r*, we will store the operand in the memory; otherwise, we will load it into the register before it is accessed. During the second phase, the algorithm calls the function *delete* to eliminate load/store instructions whose data will not be used. The DFG is then modified accordingly.

```
Algorithm Data_Management(S_DG)
  /** phase 1 **/
  Op_set = the set of operands for determining data location;
  I_set = the set of load instructions with operands in Op_set ;
  I_reg = ∅;
  for each operand op_i ∈ Op_set do
    gain_r = evaluate_r(op_i);
    gain_m = evaluate_m(op_i);
    if gain_m ≥ gain_r then
      operand op_i is fetched directly from the memory;
    else
      operand op_i is loaded into the register beforehand;
      let I_i be the instruction loading op_i;
      I_reg = I_reg ∪ I_i;
    endif
  endfor
  /** phase 2 **/
  I_del = I_set − I_reg;
  S_DG = delete(S_DG,I_del);
end Data_Management.
```

Figure 5: The data management algorithm.

We use Figure 6 to illustrate how the evaluation function works. Figure 6(a) and (b) show the ASAP scheduling and the ALAP scheduling results, respectively, of the DFG of Figure 1(c). In this example, operands of instructions 2, 5, 7, and 9 are evaluated. For instruction 9, it can only be scheduled into step 5 as determined by the ASAP and ALAP schedulings. Other instructions that might be scheduled into step 5 are 8, 11, 12, and 14. The possibility that these instructions be scheduled into step 5 is 1, $\frac{1}{4}$, 1, and $\frac{1}{4}$. If instruction 9 accesses its second operand from the memory, it can be executed in parallel with instruction 8. Among the remaining instructions, instructions 11 and 12 can also be executed in parallel. Therefore, the gain is $1 \times 1 + 1 \times \frac{1}{4} = 1\frac{1}{4}$. On the other hand, if instruction 9 accesses its second operand from the register, it can be executed in parallel with instruction 11 or 12, and leaves no more parallelism. Therefore,

the gain is $Max(1 \times 1, 1 \times \frac{1}{4}) = 1$. For instructions 2, 5, and 7, their appropriate data locations can also be determined similarily.

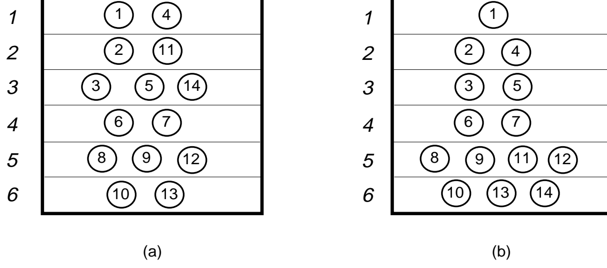


Figure 6: The ASAP scheduling (a) and ALAP scheduling (b) of the example in Figure 1 (c).

Figure 1 again is used to illustrate how the function *delete* works. In Figure 1(c), the predecessors of instruction 12 are instructions 1 and 6. But in Figure 1(d), the predecessors of instrcution 12 becomes instructions 6 and 7 because the addressing register AR4 is used by both instructions 2 and 7. Similarly, for instruction 11, its predecessors in Figure 1(d) becomes instructions 1 and 9 instead of 1 and 4 in Figure 1(c). Since instructions 5 and 9 in Figure 1(b) fetch their operands directly from the memory, instruction 4 is of no use and can be removed from the code segment. Although instructions 2 and 7 also fetch their operands directly from the memory, instruction 1 is not redundant because R10 is used by instruction 11.

4.2 Data Ordering

Figure 7 gives a pseudo code description of the data ordering algorithm. During the first phase, the ALAP scheduling is performed on both the input DFG and the modified DFG while all addressing dependences are ignored. The ALAP function returns the execution order of the operations with addressing dependences. If the execution orders for the input DFG and the modified DFG are different, it proceeds with the second phase. During the second phase, it first calls the *pattern_matching*1 function to check whether there is applicable addressing mode transformation in the addressing patterns library for this data ordering. If there exists a matched pattern, it calls the function *pattern_transform*1 to apply the pattern and return the transformed DFG without actually reordering the data in the memory. On the other hand, if there exists no matched pattern, then it calls the function *data_exchange* to reorder the data in the memory according to the new execution order without changing the addressing mode.

For the example in Figure 2 (a), if the addressing dependence between instructions 2, and 5 is ignored during ALAP scheduling, the execution order will become (5, 2). Suppose they all use $*ARn++$ for indirect addressing. If there exists a matched transformation pattern $*+AR2$ and $*AR2++(2)$, where $*+AR2$ denotes $addr = AR2+1, AR2 = AR2$, and $*AR2++(2)$ denotes $addr = AR2, AR2 = AR2+2$, in the addressing patterns library, we can change the usage of ARn by instructions 5, and 2 accordingling.

Algorithm *Data_Ordering*(S_{DG})
  /** phase 1 **/
  O_a = the set of operations with addressing dependences;
  E_a = the set of addressing dependence edges in S_{DG};
  S^a_{DG} = elimination of E_a from S_{DG};
  Exc_o = ALAP(S_{DG});
  Exc_a = ALAP(S^a_{DG});
  /** phase 2 **/
  if Exc_o and Exc_a are different **then**
    P_{trans} = pattern_matching1(Exc_o,Exc_a);
    if $P_{trans} \neq NULL$ **then**
      S_{DG} = pattern_transform1(S_{DG},P_{trans});
    else
      data_exchange(Exc_o,Exc_a);
    endif
  endif
end *Data_Ordering*.

Figure 7: The data ordering algorithm.

4.3 Transformational Retiming

Figure 8 gives a pseudo code description of the algorithm. The algorithm consists of two phases. During the first phase, it first uses the function *pattern_matching*2 to traverse the DFG and find out all the instructions matching the transformation patterns defined in the addressing patterns library. Then it sorts the matched instructions according to the priority defined in the library. Finally, it transforms the matched instructions one at a time using the function *pattern_transform*2. There are two categories of transformation patterns: one for eliminating violations against addressing mode constraints and the other for combining multiple instructions into a single one. If a transformation lengthens the critical path of the DFG, it is discarded.

During the second phase, all successor instructions of all transformed instructions are retimed to the next iteration using the *retiming_f* function. Then the transformation process of the first phase is applied again to further compact the DFG. Finally, all instructions which have been retimed from the previous iteration but have not been matched during the current iteration are retimed back to the previous iteration using the function *retiming_back*.

We use the example depicted in Figure 9 to illustrate the transformational retiming algorithm. In Figure 9(a), suppose there are three set of instructions found and sorted into the order (2, 3, 7), (5, 10), and (5, 6). The (5, 10) set is to combine into a single instruction while the other two are to eliminate violations against addressing mode constraint. Firstly, instructions 2, 3, and 7 are transformed. Then, instructions 5 and 10 are merged into a single instruction as depicted in Figure 9(b). Because this merging increases the length of the critical path (from 4 to 6), we undo it and continue with the transformation for instructions 5 and 6 as depicted in Figure 9(c). During the second phase, we retimed instructions 8, 9, 10, 11, 12, and 13, which are successors of the transformed instructions 2, 3, 7, 5, and 6, to the next iteration as depicted in Figure 9(d). Now suppose the

```
Algorithm Transformational_Retiming(S_DG)
   /** phase 1 **/
   P_set = pattern_matching2(S_DG);
   Op_trans = the set of operations in P_set;
   P_set = sort(P_set);
   for each p_trans ∈ P_set do
      S_DG = pattern_transform2(S_DG,p_trans);
   endfor
   /** phase 2 **/
   Op_suc = the set of operations that are successors of Op_trans;
   S_DG = retiming_f(S_DG,Op_suc);
   P_set = pattern_matching2(S_DG);
   Op_no = the set of operations not in P_set;
   P_set = sort(P_set);
   for each p_trans ∈ P_set do
      S_DG = pattern_transform2(S_DG,p_trans);
   endfor
   Op_pre = Op_suc ∩ Op_no;
   S_DG = retiming_b(S_DG,Op_pre);
end Transformational_Retiming.
```

Figure 8: The transformational retiming algorithm.

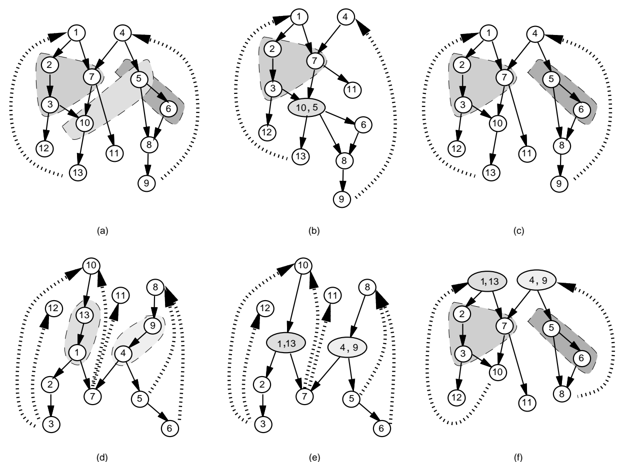


Figure 9: Transformational retiming example: (a) A DFG with three matched patterns, (b) A transformation lengthening the critical path, (c) Realization of two transformations, (d) Retimed DFG, (e) Realization of two transformations in the next iteration, (f) Back retiming.

new matched patterns are (13, 1) and (9, 4). Figure 9(e) shows the transformation result. Since the length of critical path is not increased, these two transformations are taken. Finally, instructions 8, 10, 11, and 12, which have not been transformed, are retimed back to the previous iternation as shown in Figure 9(f).

5 Experimental Results

We have implemented the proposed approach in a software system called *Theda.MS* using the C programming language. We have tested our system with a set of benchmark programs. We compare *Theda.MS* and the TI TMS320C40 C compiler in terms of the code efficiency targeted towards the TMS320C40 DSP processor.

Seven frequently used DSP algorithms in the C language are used as the benchmarks. The first four benchmarks come from [11]. They are a 6th order FIR filter (fir), an IIR filter (iir), a 16-point discrete fourier transform algorithm (dft), and a 32-by-32 two-dimensional convolution algorithm with a 3rd order Kaiser filter kernel (convolution). The latter three benchmarks are from [12]. They are an edge detection algorithm (edge) for 32-by-32 image, a 16-by-16 discrete consine transform algorithm (dct), and a 16-by-16 wiener filter frequency responance (wiener). We use the TI TMS320C40 simulator to simulate the codes and count the execution cycles.

The experiment studies the effectiveness of the proposed transformations. The first row of Table 1 shows the execution cycles of the codes generated by the TI TMS320C40 compiler with the highest level (Level 2) optimization. The second rows shows the result of our system with the transformation procedures deactivated. The next three rows show the results when *data ordering*, *transformational retiming*, and *data management* are applied, respectively. The last four rows show the results when *data ordering* followed by *transformational retiming*, *data ordering* followed by *data management*, *transformational retiming* followed by *data management*, and all all three transformations are applied, respectively. In all but the smallest examples, all transformations lead to significant reduction in the cycle counts.

Figure 10 shows the program listings for the iir benchmark. Figure 10(a) is the C source program from Page 174 of Reference [11]. Figure 10(b) is the TMS320C40 assembly code generated by the TI TMS320C40 compiler using Level 2 (the highest level) optimization. Figure 10(c) shows the assembly code generated by our system without the transformation procedures proposed in this paper. Figure 10(d) shows the assembly code generated by our system with the transformation procedures activated. Compared Figure 10(c) and (d), we can see that operand locations for instructions 5, 7, and 9 are different, hence, resulting in different instruction orderings. In Figure 10(c) we can pack three two-instruction microcodes, while in Figure 10(d) we can pack five two-instruction microcodes. The total number of instructions is also reduced.

6 Conclusion and Future Work

We have presented three transformation techniques for storage optimization during code generation targeted towards DSP processors and ASIPs with non load/store architecture. With these transformations applied before instruction scheduling and register allocation, the code compactness and execution speed can be greatly improved. Simple yet efficient algorithms have been designed based on cost functions taking into account practical considerations. Experimental results over a set of DSP kernel programs have demonstrated the effectiveness of the proposed transformations.

In the future, we would like to extend the proposed approach for a multiple-DSP-processor system. Such a system promises even higher level of performance. In addition to the proposed techniques, we will develop methods for partitioning and interface/communication design, and consider the hierarchal memory structure.

Table 1: Effect of the three transformation techniques targetted towards the TI's DSP processor TMS320C40 in terms of execution cycles. $Theda()$: our system without transformations applied. o: data ordering; r: transformational retiming; m: data management.

Compilers	fir	iir	dft	con.	edge	dct	wiener
TI	40	65	11456	126377	25565	10115	8805
$Theda()$	37	58	10918	126377	25565	9843	8805
$Theda(o.)$	37	58	10896	126377	25525	9843	8805
$Theda(r.)$	37	52	10642	118295	24953	9577	8805
$Theda(m.)$	37	57	10775	126377	22657	7842	7743
$Theda(o.r.)$	37	52	10642	118295	24953	9577	8805
$Theda(o.m.)$	37	57	10775	126377	22657	7842	7743
$Theda(r.m.)$	37	52	10388	118295	21968	7349	7743
$Theda(o.r.m.)$	37	52	10388	118295	21968	7349	7743
$\frac{Theda(o.r.m.)}{TI}$	0.93	0.80	0.91	0.94	0.86	0.73	0.88

```
for  ( i = 0 ; i < iir–>length ; i ++ )  {
      history1 = * hist1_ptr ;
      history2 = * hist2_ptr ;

      output = output – history1 * ( * coef_ptr ++ ) ;
      new_hist = output – history2 * ( * coef_ptr ++ );

      output = new_hist + history1 * ( * coef_ptr ++ ) ;
      output = output + history2 * ( * coef_ptr ++ ) ;

      * hist2_ptr ++ = * hist1_ptr ;
      * hist1_ptr ++ = new_hist ;
      hist1_ptr ++ ;
      hist2_ptr ++ ;
}
```

(a)

```
LDF     * AR4 , R10
MPYF    * AR2++ , R10 , R0
SUBF    R0 , R7
LDF     *AR5 , R2
MPYF    * AR2++ , R2 , R3
SUBF    R3 , R9 , R3
MPYF    * AR2++ , R10 , R9
ADDF    R3 , R9
MPYF    * AR2++ , R2 , R0
ADDF    R0 , R9
STF     R10 , *AR5++
STF     R3 , * AR4++
ADDI    1 , AR4
ADDI    1 , AR5
```

(b)

```
    LDF     * AR4 , R5
||  LDF     * AR5 , R4
    MPYF    * AR2++ , R5 , R6
||  STF     R5 , * AR5++
    SUBF    R6 , R9 , R3
    MPYF    * AR2++ , R4 , R0
    SUBF    R0 , R3 , R3
    MPYF    * AR2++ , R5 , R1
||  STF     R0 , * AR4++
    ADDF    R3 , R1 , R2
    MPYF    * AR2++ , R4 , R1
    ADDF    R1 , R2 , R2
    ADDI    1 , AR4
    ADDI    1 , AR5
```

(c)

```
    LDF     * ++AR4 , R5
||  LDF     * ++AR5 , R4
    MPYF    * AR2++ , R5 , R6
    SUBF    R6 , R7 , R3
||  MPYF    * AR2++ , * AR5 , R0
    SUBF    R0 , R3 , R3
||  MPYF    * AR2++ , *AR4 , R1
    ADDF    R3 , R1 , R2
||  MPYF    * AR2++ , * AR5 , R1
    STF     R5 , * AR5++
||  STF     R3 , * AR4++
    ADDF    R1 , R2 , R2
```

(d)

Figure 10: Program listings for the iir benchmark: (a) The C source code, (b) The assembly code generated by the TI compiler, (c) The assembly code generated by our system without transformations. (d) The assembly code generated by our system with transformations.

References

[1] Pieere G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala, "DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective." Journal of VLSI Signal Processing, 1994.

[2] Clifford Liem, Trevor May, and Pieere Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation." Proc. The European Design and Test Conference, pp. 31-37, 1994.

[3] Jose Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck, "Software Synthesis for DSP Using Ptolemy." Journal on VLSI Signal Processing, special issue on "Synthesis for DSP", 1993.

[4] Gert Goossens, Jan Rabaey, Joos Vandewalle, and Hugo De Man, "An Efficient Microcode Compiler for Application Specific DSP Processors." IEEE Transactions on Computer-Aided Design, pp. 925-937, Sep. 1990.

[5] Viraphol Chaiyakul, Daniel D. Gajski, and Loganath Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances." Proc. 30th Design Automation Conference, pp. 413-418, June 1993.

[6] Zia Iqbal, Miodrag Potkonjak, Sujit Dey, and Alice Parker, "Critical Path Minimization Using Retiming and Algebraic Speed-Up." Proc. 30th Design Automation Conference, pp. 573-577, June 1993.

[7] Shan-Hsi Huang, Jan M. Rabaey, "Maximizing the Throughput of High Performance DSP Applications Using Behavioral Transformations." Proc. The European Design and Test Conference, pp. 25-30, 1994.

[8] David J. Kolson, Alexandru Nicolau, Nikil Dutt, "Minimization of Memory Traffic in High-Level Synthesis." Proc. 31st Design Automation Conference, pp. 149-154, June 1994.

[9] Florin Balasa, Francky Catthoor, and Hugo De Man, "Dataflow-driven Memory Allocation for Multidimensional Signal Processing Systems." Proc. 1994 ICCAD.

[10] Wei-Kai Cheng, and Youn-Long Lin, "Code Generation for a DSP Processor." Proc. 7th International Symposium on High-Level Synthesis, pp. 82-87, 1994.

[11] Paul M. Embree, and Bruce Kimble, "C Language algorithms for Digital Signal Processing." Prentice-Hall, 1991.

[12] Ioannis Pitas, "Digital Image Processing Algorithms." Prentice-Hall, 1993.