DE-PIPELINE A SOFTWARE-PIPELINED LOOP

Bogong Su [1] sub@wpunj.edu
Jian Wang [2] jiwang@nortelnetworks.com
Erh-Wen Hu [1] hue@wpunj.edu
Joseph Manzano [1] manzanoj@student.wpunj.edu

Abstract
Software pipelining is a loop optimization technique that has been widely implemented in modern optimizing compilers. In order to fully utilize the instruction level parallelism of the recent VLIW DSP processors, DSP programs have to be optimized by software pipelining. However, because of the transformation of the original sequential code, a software-pipelined loop is often difficult to understand, test, and debug. It is also very difficult to re-use and port a software-pipelined loop to other processors especially when the original sequential code is unavailable. In this paper we propose a de-pipelining technique, which converts the optimized assembly code of a software-pipelined loop back to a semantically equivalent sequential counterpart. Preliminary experiments on 20 assembly programs verifies the validity of the proposed de-pipelining algorithm.

I. INTRODUCTION

Digital Signal Processing (DSP) industry has been growing rapidly over the past few years [3]. Due to the constant need to improve the performance and to address a wide range of applications, a new breed of DSP processors based on Very Long Instruction Words (VLIW) have been introduced to the market by most major manufacturers. In order to fully utilize the instruction level parallelism of these VLIW DSP processors, DSP applications have to be optimized by software pipelining.

Software pipelining has been studied for many years [2,4,8]. It is a loop optimization technique widely implemented in optimizing compilers in order to speed up the execution of loops on processors with instruction level parallelism. Such DSP processors include Texas Instruments' C6X and StarCore's SC140. As an example, Figure 1 shows the code of a software-pipelined loop of a dot-product function. The code is written in the assembly language of TIC62 processor [7].

De-pipelining is the reverse of pipelining operation; it restores the assembly code of a software-pipelined loop back to its semantically equivalent sequential form. That is, given the code of a software-pipelined loop shown in Figure 1, de-pipelining will convert it to its semantically equivalent sequential loop shown in Figure 2.

The motivation for our study of de-pipelining is as follows. First, due to the transformation of the original sequential code, the code of a software-pipelined loop is very difficult to understand, test, and debug. Second, because of the "binary compatibility" issue, it is a very difficult to reuse

```
        MVK 0X32,B0
        MVC CSR,B8          ZERO A4             ZERO B5
        AND -2,B8,B4
        MVC B4,CSR          SUB B0,5,B0         MVK 0X2,A1
L1:     [B0] B  L2
        LDH *++A0(4),A3     LDH *++B7(4),B4
        LDH *+A0(2),A3      LDH *+B7(2),B4      [B0] B  L2
        LDH *++A0(4),A3     LDH *++B7(4),B4
L2:     LDH *+A0(2),A3      LDH *+B7(2),B4      [B0] B  L2          MPY B4,A3,B6    [!A1] ADD B6,B5,B5
        LDH *++A0(4),A3     LDH *++B7(4),B4     [B0]SUB B0,1,B0     MPY B4,A3,A5    [!A1] ADD A5,A4,A4  [A1] SUB A1,1,A1
L3:     LDH *+A0(2),A3      LDH *+B7(2),B4      MPY B4,A3,B6        ADD B6,B5,B5
                                                MPY B4,A3,A5        ADD A5,A4,A4
                                                MPY B4,A3,B6        ADD B6,B5,B5
                                                MPY B4,A3,A5        ADD A5,A4,A4
                                                MPY B4,A3,B6        ADD B6,B5,B5
                                                MPY B4,A3,A5        ADD A5,A4,A4
                                                                    ADD B6,B5,B5
                                                                    ADD A5,A4,A4
```

Figure 1 Software-pipelined loop of dot-product function in TIC62 assembly code

[1] Dept. of Computer Science, The William Paterson University of New Jersey, Wayne, NJ 07470, USA
[2] Wireless Speech and Data Processing, Nortel Networks, Montreal, QC, Canada, H3E 1H6

and port a software-pipelined loop to other processors. Third, although a software-pipelined loop is efficient in term of CPU execution time, it may be inefficient in terms of memory usage. It may not be suitable for certain applications with limited memory space. Finally, we note that most DSP applications have been optimized by software pipelining and well tested, but their semantically equivalent sequential loop code may not be available. To our best knowledge, there is no published report that addresses de-pipelining problems.

In general, a software-pipelined loop consists of three parts: the prelude, the loop kernel, and the postlude. As shown in Figure 1, the prelude part is from L1 to L2, the loop kernel is from L2 up to L3, and the postlude includes instructions after L3. There are very strict timing dependencies among the instructions in the prelude and in the loop kernel. For example, if instruction at L1 is issued at cycle t, then the three instructions between L1 and L2 must be issued at cycle (t+1), (t+2) and (t+3), respectively. Any delay will destroy the semantics of the program. These strict timing dependencies cannot be represented by the conventional control dependence and data dependence. Actually, it is difficult to understand the semantics of a software-pipelined loop before it is de-pipelined, for example, the construction of control flow graph of a software-pipelined loop is awkward due to the multi-cycle branch delay and the overlap of many iterations.

In this paper, we propose a de-pipelining algorithm. We first use the strict timing dependencies to identify the loop kernel. We then build the data dependence graph (DDG) of the software-pipelined loop with the help of the loop unrolling technique. Finally, we use the DDG to construct the semantically equivalent sequential loop.

In the following section, we demonstrate our de-pipelining algorithm with a working example.

```
        MVK   0X32,B0
        ZERO  A4
        ZERO  B5
LE:     LDH   *++A0(4),A3
        LDH   *++B7(4),B4
        LDH   *+A0(2),A3
        LDH   *+B7(2),B4
[B0]    SUB   B0,1,B0
[B0]    B     LE
        MPY   B4,A3,B6
        NOP
        MPY   B4,A3,A5
        ADD   B6,B5,B5
        ADD   A5,A4,A4
```

Figure 2 The semantically equivalent sequential code of a software-pipelined loop of dot-product function in TIC62 assembly code

II. DE-PIPELINING ALGORITHM

Our de-pipelining algorithm involves the following steps:

1. Loop detection.
2. Live variable analysis.
3. DDG construction.
4. Checking software-pipelined loop.
5. Finding parts of the prelude and the postlude.
6. Scheduling.
7. Loop count calculation.

Figure 3 shows a segment of TIC62 assembly code as a working example; the leftmost column is the line number and the || symbol means the instruction in the current line are executed in parallel with the instruction in previous line.

```
1                       MVK 0X32,B0
2                       ZERO A4
3       ||              ZERO B5
4                       SUB B0,5,B0
5       ||              MVK 0X2,A1
6               [B0]    B  L2
7                       LDH *++A0(4),A3
8       ||              LDH *++B7(4),B4
9                       LDH *+A0(2),A3
10      ||              LDH *+B7(2),B4
11      ||      [B0]    B  L2
12                      LDH *++A0(4),A3
13      ||              LDH *++B7(4),B4
14  L2:                 LDH *+A0(2),A3
15      ||              LDH *+B7(2),B4
16      ||      [B0]    B  L2
17      ||              MPY B4,A3,B6
18      ||      [!A1    ADD B6,B5,B5
19                      LDH *++A0(4),A3
20      ||              LDH *++B7(4),B4
21      ||      [B0]    SUB B0,1,B0
22      ||              MPY B4,A3,A5
23      ||      [!A1    ADD A5,A4,A4
24      ||      [A1]    SUB A1,1,A1
25                      LDH *+A0(2),A3
26      ||              LDH *+B7(2),B4
27      ||              MPY B4,A3,B6
28      ||              ADD B6,B5,B5
29                      MPY B4,A3,A5
30      ||              ADD A5,A4,A4
31                      MPY B4,A3,B6
32      ||              ADD B6,B5,B5
33                      MPY B4,A3,A5
34      ||              ADD A5,A4,A4
35                      MPY B4,A3,B6
36      ||              ADD B6,B5,B5
37                      MPY B4,A3,A5
38      ||              ADD A5,A4,A4
39                      ADD B6,B5,B5
40                      ADD A5,A4,A4
```

Figure 3 A segment of TIC62 assembly code

1. **Loop detection**: From the given segment of the assembly code, identify the body of a software-pipelined loop.

(1) Find the loop entry: If there is a backward branch instruction then the target of the branch instruction is the loop entry. (2) Find the length of the body of the software-pipelined loop: Define the pre-header as the code area just above the loop entry whose length equals the branch delay

slots pluses one. If there are some forward branch instructions within the pre-header area that has a loop entry as their target, then the length of the body of the software-pipelined loop is equal to the distance between the nearest forward branch and the loop entry; otherwise the length of loop body is equal to the distance between the backward branch and loop entry plus branch delay slots.

In Figure 3, L2 is the loop entry because it is the target of a backward branch and the length of the software-pipelined loop body is 2.

2. **Live variable analysis**: From a given software-pipelined loop body, find all last_instructions.

We define last_instructions either as the instructions that write to registers which are live or the instructions that perform memory store operations. From the bottom of the loop area upward we perform a search for all last_instructions.

In Figure 3, ADD B6,B5,B5 and ADD A5,A4,A4 are last_instructions.

3. **DDG construction**: To build the DDG of a software-pipelined loop.

(1) Unroll loop body k - 1 times, where k equals the maximum number of instructions in instruction groups in the loop body. An instruction group is defined as a group of instructions that are executed in parallel. (2) Starting from last_instructions, build the DDG bottom up by using the height-first search algorithm.

Figure 4 shows the DDG of the software-pipelined loop of the dot-product function.

4. **Software-pipelined loop checking**: To check whether a given loop is software-pipelined.

If there exist two instructions Ii and Ij in the loop body such that their distance in loop body is less than their distance in the DDG, then the loop body is the result of software pipelining.

From Figures 3 and 4, we can confirm that the loop identified in Figure 3 is a software-pipelined loop.

5. **Find parts of the prelude and the postlude**: To find the prelude and the postlude of a software-pipelined loop in the given assembly code segment.

(1) Starting from the loop entry, search upward until reaching the top boundary of the software-pipelined loop to find all instruction groups which contain those instructions that exist in loop body. The highest instruction group is the upper boundary of the prelude. (2) Starting from the bottom of the body of the software-pipelined loop, search downward until reaching the bottom boundary of the software-pipelined loop to find all instruction groups which contain those instructions that exist in the loop body. The lowest instruction group is the lower boundary of the postlude.

In Figure 3, the prelude is from line No. 6 through 13, and the postlude is from line No. 25 through 40.

6. **Scheduling**: To convert the DDG to sequential code.

(1) From last_instructions, proceeds bottom up with list-scheduling algorithm to arrange the partial order list of the critical path of the DDG. It is necessary to satisfy the latencies of all instructions and insert NOPs as necessary. (2) Insert all instructions in non-critical paths to scheduled critical path. (3) Delete all instructions in the prelude and in the postlude which have the same instructions in the loop body.

7. **Loop count calculation**: To figure out the loop count of the sequential code of a software-pipelined loop.

Besides the initial value of the loop count in the given software-pipelined loop, one must consider many other factors such as the number of SUB instructions for decreasing loop counter in the prelude, the number of branch instructions in the prelude whose target is the loop entry, the number of last_instructions in the postlude, the relative position between the backward branch and the loop count decrement instructions in the given software-

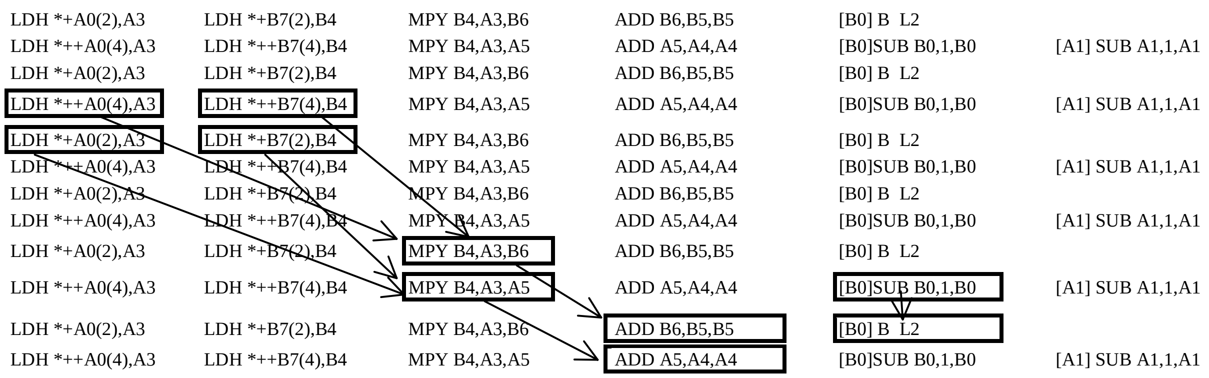

Figure 4 DDG of working example

Table 1 Experiment Result

Assembly code	Characteristics	Software-pipelined loop		De-pipelining result	
		Initial count	Body length	Count value	Body length
dot product_1[2]	Normal	43	1	50	14
dot product_2[2]	No postlude	50	1	50	14
dot product_3[2]	Sub & branch in prelude only, no postlude	57	1	50	14
dot product_4[2]	Branch in prelude only, no postlude	51	1	50	14
dot product_5[1]	Normal	50	1	50	14
FIR [1]	No postlude	32	3	32	15
FIRnorld [1]	No postlude	16	2	16	15
IIR [2]	No postlude	100	4	100	13
Codebook [2]	No postlude, conditional branch in loop body	32	2	32	9
Vec_mpy [1]	Normal	75	3	75	16
Latsynth [1]	Normal	200	5	200	25
WVS [2]	Normal	49	2	50	16
Add_test [1]	No postlude	5	2	5	6
Loop_test_1 [1]	Branch in prelude only	50	2	50	6
Loop_test_2 [1]	Branch in prelude only	100	2	100	7
Loop_test_3 [1]	Branch in prelude only	100	3	100	7
Loop_test_4 [1]	normal	50	5	50	11
Loop_test_8 [1]	No prelude	20	9	20	21
Loop_test_12 [1]	No prelude	25	23	25	27
Loop_test_16 [1]	No prelude	50	17	50	35

note: [1] generated by Compiler; [2] generated by Linear assembler

pipelined loop body, etc. From Step 6 and 7 we obtain the sequential code of the dot-production function as shown in Figure 2, which is semantically equivalent to the given loop in Figure 3.

IV. EXPERIMENT

We conducted experiment on 20 assembly code segments belonging in different applications with different loop length and various situations of the prelude and the postlude. The code segments are generated by TIC62 compiler either or linear assembler for hand crafting [7]. First, we convert these assembly code segments to sequential code by using de-pipelining technique manually. We then use the TIC62 simulator to run both original assembly code and the converted sequential code. All computation results show our de-pipelining algorithm is valid for these programs. Table 1 summarizes the characteristics of the software-pipelined loops and the de-pipelining results of these 20 programs.

V. SUMMARY

We present our de-pipelining technique and the experimental result. Our approach will be a very useful tool for DSP users to understand and debug software-pipelined assembly code. Furthermore, our de-pipelining technique can be extended to solve the “compatibility issue” that involves software-pipelined loops of VLIW computers. Although the “compatibility issue” was solved by using dynamic rescheduling [1], it does not address the code that involves software-pipelined loops. By using the de-pipelining technique, one can convert the software-pipelined code from a source VLIW processor to a set of semantically equivalent sequential code at an intermediate level. The intermediate code can then be fed into the compiler of the target VLIW processor. Finally, our approach can be adapted to convert the assembly code from one VLIW DSP processor to other DSP processors [6].

ACKNOWLEDGEMENT

Su and Manzano would like to thank the Center for Research, College of Science and Health, William Paterson University, for research support in the summer of 2002.

REFERENCES

[1] Conte T., and Sathaye S., Optimization of VLIW Compatibility Systems Employing Dynamic Rescheduling, *Journal of Parallel Programming*, vol.35, no.2, 1997

[2] Fisher J. and Rau R., "Instruction-Level Parallel Processing", *Science* vol.253, 1991.

[3] Strauss, Will; “Digital Signal Processing: The New Semiconductor Industry Technology Driver”*; IEEE Signal Processing Magazine*, March 2000

[4] Su B., Ding S., and Xia J., "URPR - An Extension of URCR for Software Pipelining", *Proc. of the 19th Microprogramming Workshop(MICRO-19)*, Oct. 1986,

[5] Su B., Wang J., and Hu E., “Code Migration from Conventional DSPs to VLIW DSPs”, *Proc. of ICPSAT2000*, Oct. 2000

[6] Su B., Wang J., Hu E., and Manzano J., Assembly Code Conversion Through pattern Mapping Between Two VLIW DSP Processors: A Case Study, *Proc. of ICSP'02*, Aug. 2002.

[7] *TMS320C62x/C67x Programmer’s Guide*

[8] Wang J., Eisenbeis C. Su B. & Jourdan M., "Decomposed Software Pipelining: A New Perspective and A New Approach". *International Journal on Parallel Processing*, Vol.22, No.3, 1994