Software De-Pipelining Technique

Bogong Su [1] sub@wpunj.edu
Jian Wang [2] jiwang@nortelnetworks.com
Erh-Wen Hu [1] hue@wpunj.edu
Joseph Manzano [1] Josbry21@cs.com

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

Abstract

Software pipelining is a loop optimization technique used to speed up loop execution. It is widely implemented in optimizing compilers for VLIW and superscalar processors that supports instruction level parallelism. Software de-pipelining is the reverse of software pipelining; it restores the assembly code of a software-pipelined loop back to its semantically equivalent sequential form. Due to the non-sequential nature of the often optimized assembly code, it is very difficult to gain insight into the meaning of the code. Consequently, the task of de-pipelining the code of a software-pipelined loop is very complex and challenging. We present in this paper our de-pipelining algorithm with a formal description, proof, and a set of working examples. Experiments with loops taken from some practical DSP programs are conducted on popular VLIW digital signal processors to verify the algorithm. Some applications of software de-pipelining are discussed.

1. Introduction

Because of the practical importance of porting low-level code from one processor to another, decompilation has been studied for many years [1,3,5,9,14,20]. Yet few of these studies have dealt with source machine that supports instruction level parallelism or ILP [2]. De-compiling optimized code is difficult because the de-compiler must de-optimize the low-level code of the source machine [14]. It is even more so when the source machine supports ILP and the source code has been optimized by software pipelining.

Software pipelining [6,11,15,21] is a loop optimization technique used to speed up loop execution. It is widely implemented in optimizing compilers for VLIW and superscalar processors [8,13] such as IA-64, Texas Instruments' C6X and StarCore's SC140 DSP that support instruction level parallelism.

Software de-pipelining (de-pipelining hereafter) [16] is the reverse of software pipelining; it restores the assembly code of a software-pipelined loop back to its semantically equivalent sequential form.

The motivation for our study of de-pipelining is as follows. First, due to the transformation of the original sequential code, especially when the source machine has large branch delay and/or when it uses sophisticated optimization techniques such as prelude and postlude collapsing [7], the code of a software-pipelined loop is very difficult to comprehend, analyze, and debug. As an example, Figure 1.1 shows the assembly code segment of a software-pipelined loop optimized with both prelude and postlude collapsing for Texas Instruments' C62 (TIC62 hereafter) processor. The "||" symbol in the code segment means that the instruction in the current line is executed in parallel with the instruction in the previous line, and the set of instructions executed in parallel is referred to as an instruction group in this paper. Because TIC62 has long branch delay (6 clock cycles) and its compiler performs prelude and postlude collapsing on the software-pipelined loop in order to reduce code size, the instructions in the code segment have been so transformed that it is very difficult comprehend the meaning of the code and to determine if this code segment is a software-pipelined loop, let alone to identify the body, the prelude and the postlude of the software-pipelined loop.

```
                  MVK 57, A1
                  [A1] SUB A1,1,A1
                  ||  ZERO A7
                  ||  ZERO B7
                  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
                  ||  ZERO A6
                  ||  ZERO B6
                  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
                  ||  ZERO A2
                  ||  ZERO B2
                  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
                  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
                  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
LOOP:             LDW *A4++,A2
                  ||  LDW *B4++,B2
                  ||  [A1] SUB A1,1,A1
                  ||  [A1] B    LOOP
                  ||  MPY A2,B2,A6
                  ||  MPYH A2,B2,B6
                  ||  ADD A6,A7,A7
                  ||  ADD B6,B7,B7
                  ADD A7,B7,A4
```

Figure 1.1 An assembly code segment of TIC62

Second, code conversion between digital signal processors or DSPs is of practical importance. Yet it is often very difficult to port code from a VLIW DSP to other processors, even in cases where the new target processor are in the same series, largely due to the complexity introduced by software-pipelined loops in source machine code. Third, although a software-pipelined loop is more efficient in terms of execution time, it is often inefficient in terms of memory usage. For memory critical applications, it may be necessary to de-pipeline a software-pipelined loop.

We first investigated the de-pipelining technique in 2003 [16]. Since then we have extended our studies by taking into considerations of prelude and postlude collapsing recovery. We have also developed new algorithms to construct data dependence graph or *DDG* and to compute the loop count. In section 2, we provide a formal description and proof of our de-pipelining procedure. We provide in section 3 the detailed de-pipelining algorithm and in section 4 a working example. Section 5 shows our experimental results. In the concluding section 6, we introduce some applications of de-pipelining. A summary discussion is given in section 7.

2. Formal description and proof of de-pipelining

Before formally describing the software de-pipelining problem, we define loop's data dependence graph (*LDDG*) and loop schedule.

<u>Definition 1</u>: The data dependence between the instructions in a loop program can be represented by a doubly weighted data dependence graph, *G(O,E,d,t)***,** which is called the Loop Data Dependence Graph or *LDDG*. In the expression, *O* is the set of the instructions in the loop; *E* is the set of dependence edges; *d* is dependence distance; and *t* is the delay. Both *d* and *t* are nonnegative integers and each *(d,t)* pair is associated with an edge. For example, for an edge *e(op1,op2)* that specifies the relationship between instructions *op1* and *op2*, *op2* can be issued for execution only *t(e)* cycles after the start of the instruction *op1* of the *d(e)*th previous iteration. A data dependence is called a loop-independent dependence if its dependence distance is 0. A data dependence is called a loop-carried dependence if its dependence distance is greater than 0.

As an example, Figure 2.1(b) is the *LDDG* of the loop shown in Figure 2.1(a), where all delays are assumed to be a single cycle. The data dependence edge from instruction 4 to instruction 2 is a loop-carried dependence because its dependence distance is one. All other dependence edges are loop-independent dependence edges because their dependence distances are zero.

<u>Definition 2</u>: For a given loop and its *LDDG(O,E,d,t)*, a loop schedule σ is a mapping from *O* x *N* to *N*, where *N* is the set of nonnegative integers. $\sigma(op,i)$ denotes the cycle number in which the instance of instruction *op* of the *i*th iteration is issued for execution. σ is a valid loop schedule if and only if the following three constraints are satisfied:

1. Resource constraint: in each cycle, there is no hardware resource conflict.
2. Data dependence constraint: for any edge *e(op1, op2)* and for any $j > 0$,
   $$\sigma(op1, j) + t(e) <= \sigma(op2, j + d(e));$$
3. Cycle constraint: σ must be expressible in the form of a loop, that is, there is an integer *II*, for any instruction *op* in the loop and for any integer $j > 1$, $\sigma(op, j) = \sigma(op, j-1) + II*(j-1)$, where $\sigma(op, 1)$ denotes the cycle number at which the instance of an *op* of the first iteration is issued for execution. *II* is called the initiation interval.

```
     for (i=1;i<100;i++)
     {
1.        t1=a[i]
2.        t2=b[i-1]
3.        t3=t1+t2
4.        b[i]=t3
5.        t5=t3*2
6.        c[i]=t5
     }
```

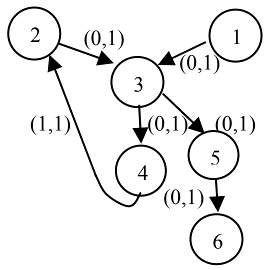


Figure 2.1 (a) A loop Figure 2.1 (b) LDDG

<u>Definition 3</u> The task of software pipelining is to find a valid loop schedule with minimum initiation interval *II*. A software-pipelined loop consists of three parts – the prelude, pipelined loop body, and the postlude. The pipelining depth is defined as the number of iterations from which the instructions are overlapped in the pipelined loop body.

```
1 (1)                           }
2 (1)                           }  prelude
3 (1)                           }
4 (1)     1 (2)                 }
+-----------------------------+
|          2                  |   software-pipelined
|5         3                  |   loop body
|6         4          1       |   (repeat 97 times)
+-----------------------------+
          5 (98)     2 (99)   }
          6 (98)     3 (99)   }
                     4 (99)   }  postlude
                     5 (99)   }
                     6 (99)   }
```

Figure 2.2 Software-pipelined Loop

Figure 2.2 is the software-pipelined loop of Figure 2.1(a). In the figure, *op(n)* represents the instruction instance of op from the n^{th} iteration. Here we have assumed that the processor has three memory load/store units, one addition unit, and one multiplication unit. It is noted that all the three constraints in Definition 2 are satisfied and the initiation interval of this pipelined loop is three cycles.

Definition 4: Given a loop and its *LDDG(O,E,d,t)*, the task of de-pipelining is to find a valid loop schedule *dp* that satisfies the following two conditions:

1. For any instruction *op* and any integer $j > 0$,
   $dp(op, j) + t(op) <= dp(op, j+1)$, and
1. For any two instructions, *op1* and *op2*, and any integer $j > 0$,
   $dp(op1, j) + t(op1) <= dp(op2, j+1)$.

In general, in a software-pipelined loop, instructions from different iterations are overlapped for execution. On the other hand, in a de-pipelined loop, instructions from different iterations must not overlap for execution and the loop-carried dependence is automatically satisfied.

Figure 2.3(a) is the de-pipelined loop directly derived from the software-pipelined loop shown in Figure 2.2. It is unnecessary that the de-pipelined loop be exactly the same as the original loop (Figure 2.1(a)) although they must be semantically equivalent.

```
    for (i=1;i<100;i++)
    {
1.        t1=a[i]
2.        t2=b[i-1]
3.        t3=t1+t2
4.        t5=t3*2
5.        c[i]=t5
6.        b[i]=t3
    }
```

Figure 2.3(a)
A de-pipelined loop

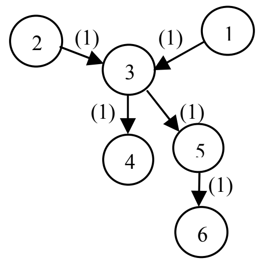


Figure 2.3 (b) LBDDG

Definition 5: Given a loop and its *LDDG(O,E,d,t)*, we define the loop body *DDG* or *LBDDG(O, E_0, t)*, where E_0 is a subset of *E* that contains only all loop-independent edges. Each edge of E_0 is only associated with one non-negative integer *t* which denotes the delay.

Figure 2.3(b) is the *LBDDG* of the loop shown in Figure 2.1(a). *LBDDG* is similar to *LDDG* except that in *LBDDG*, only loop-independent edges are included.

Now the question is, given a software-pipelined loop, how to generate the original loop's LBDDG?

Theorem 1: Given the body of a software-pipelined loop and its prelude, the *LBDDG* can be constructed by scanning the prelude and the pipelined loop body if the prelude is neither compacted nor collapsed.

In practice, the prelude is sometimes compacted or collapsed for the purpose of reducing code size. It is noted that if the prelude is either compacted or collapsed, it may not satisfy the cycle constraint of a valid loop schedule as discussed in Definition 2. The cycle constraint is a pre-condition that must be satisfied in order to correctly construct the *LBDDG*. For this reason, we have developed an algorithm to recover a collapsed prelude, which is discussed in section 3. We now prove Theorem 1 below.

Proof: We now present an algorithm to construct the *LBDDG*, then prove that the *LBDDG* thus obtained is the *LBDDG* of the original loop.

Algorithm:

```
1  Denote the prelude and the pipelined loop body as a sequence of VLIW instructions, $L_1, L_2,..., L_k$, where $L_1$ is the first VLIW instruction in the prelude and $L_k$ is the last VLIW instruction in the pipelined loop body.
2  Set DDG = empty.
3  For $i$ = 1 to k; find $L_i'$, where $L_i' = L_i$ - {those instructions which have been selected in DDG}; identify all read-write data dependences among the instructions in $L_i'$; find out all write-read, read-write and write-write data dependencies from DDG's instructions on the instructions of $L_i'$; add all the instructions of $L_i'$ to DDG; update DDG;
4  The DDG built in step 3 is the LBDDG; return;
```

```
L1:     1 (1)
L2:     2 (1)
L3:     3 (1)
L4:     4 (1)  ||   1 (2)
L5:     2 (2)
L6:     5 (1)  ||   3 (2)
L7:     6 (1)  ||   4 (2)  ||  1 (3)
```

Figure 2.4(a) The prelude and software-pipelined loop body

Proof:

1 Because the prelude is neither compacted nor collapsed, the software-pipelined loop satisfies the cycle constraint;

2 From the cycle constraint, $\sigma(op, i) < \sigma(op, i+1)$ because $II >= 1$;

3 Therefore, the selected instructions of the *DDG* built in step 3 of the algorithm come from the first iteration of the software-pipelined loop.

4 Because all the instructions come from the same iteration, the data dependence edges built in step 3 of the algorithm only cover all the loop-independent dependence edges of the original sequential loop.

Using the software-pipelined loop shown in Figure 2.2 as an example, Figure 2.4 illustrates how the *LBDDG* algorithm works. First, we find out the VLIW sequence of the prelude and the pipelined loop body, as shown in Figure 2.4(a). We then follow the algorithm to construct the *LBDDG*, which is shown in Figure 2.4(b).

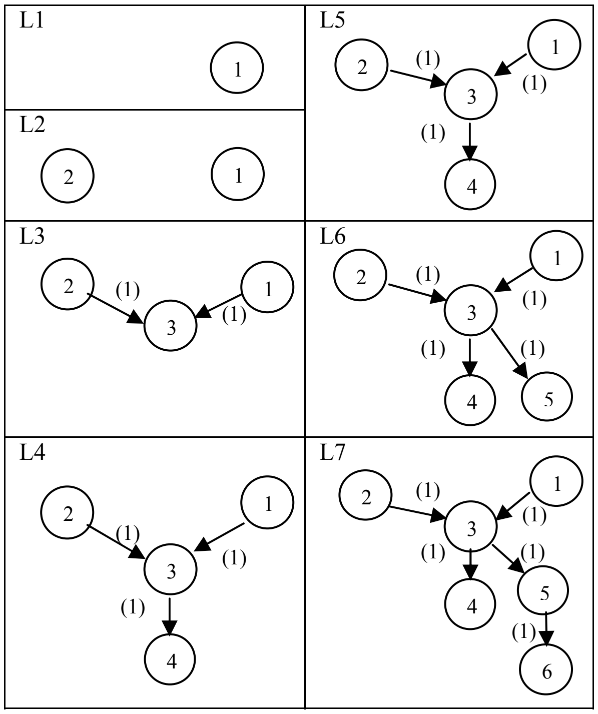


Figure 2.4(b) Construction of LBDDG

Theorem 2: Given a loop and its *LBDDG(O,E$_0$,t)*, a loop schedule σ is valid if the following conditions are satisfied:

1. The two conditions specified in Definition 4.
2. The resource constraints and the cycle constraints specified in Definition 2.
3. For each edge *e* of E_0, *e(op1, op2),*

   $\sigma(op1, j) + t(op1) <= \sigma(op2, j)$.

The proof is straightforward because the loop-carried dependences are automatically satisfied if the two conditions of Definition 4 are satisfied.

Given a software-pipelined loop, Theorem 2 provides the basis to generate the de-pipelined loop. That is, we can apply any list scheduling algorithm to schedule the loop under the constraints of hardware resource and the *LBDDG*.

3. De-pipelining Algorithms

De-Pipelining (as shown in Figure 3.1)

Input: A given segment of assembly code that includes software-pipelined loop.

Output: A sequential loop which is semantically equivalent to the given software-pipelined loop

Algorithm:

1. Loop detection. Call **detect_loop** function to find the software-pipelined loop body from the given segment of assembly code.
2. Software-pipelined loop checking. Call **check_spl** function to determine whether the given loop is software-pipelined.
3. Live variable analysis. Call **analyze_var** function to obtain all **last_instructions** from a given loop body
4. Identify prelude and postlude. Call **find_prepost** function to find the scopes of prelude and postlude of a software-pipelined-loop in the given segment of the assembly code.
5. Prelude recovery. Call **prelude_recovery** function to recover the prelude if the prelude has been collapsed.
6. Postlude recovery. Call **postlude_recovery** function to recover the postlude if the prelude has been collapsed.
7. Build *LBDDG*. Call **build_LBDDG** function to build the *LBDDG* of the software-pipelined loop
8. Scheduling. Call **scheduling** function to convert *LBDDG* to a sequential code.
9. Loop count calculation. Call **calculate_lcount** function to compute the loop count of the sequential code of the software-pipelined loop.

The **detect_loop** function**:**

Input: The given segment of assembly code

Output: The loop body within the given segment of the assembly code.

Algorithm:

If the loop's start and end are not specified do the following steps.

1. Find loop entry: if there is a backward branch instruction, then the target of this branch instruction is the loop entry point.

   If there are forward branch instructions within the pre-header area, which includes all the instructions from the loop entry point spanning upward a distance equal to the latency of the branch instruction.

   then: the length of the loop body equals the distance between the nearest forward branch and loop entry point plus the distance between the backward branch and loop entry.

   else: the length of the loop body equals the distance between the backward branch and loop entry point plus the latency of branch instruction minus 1.

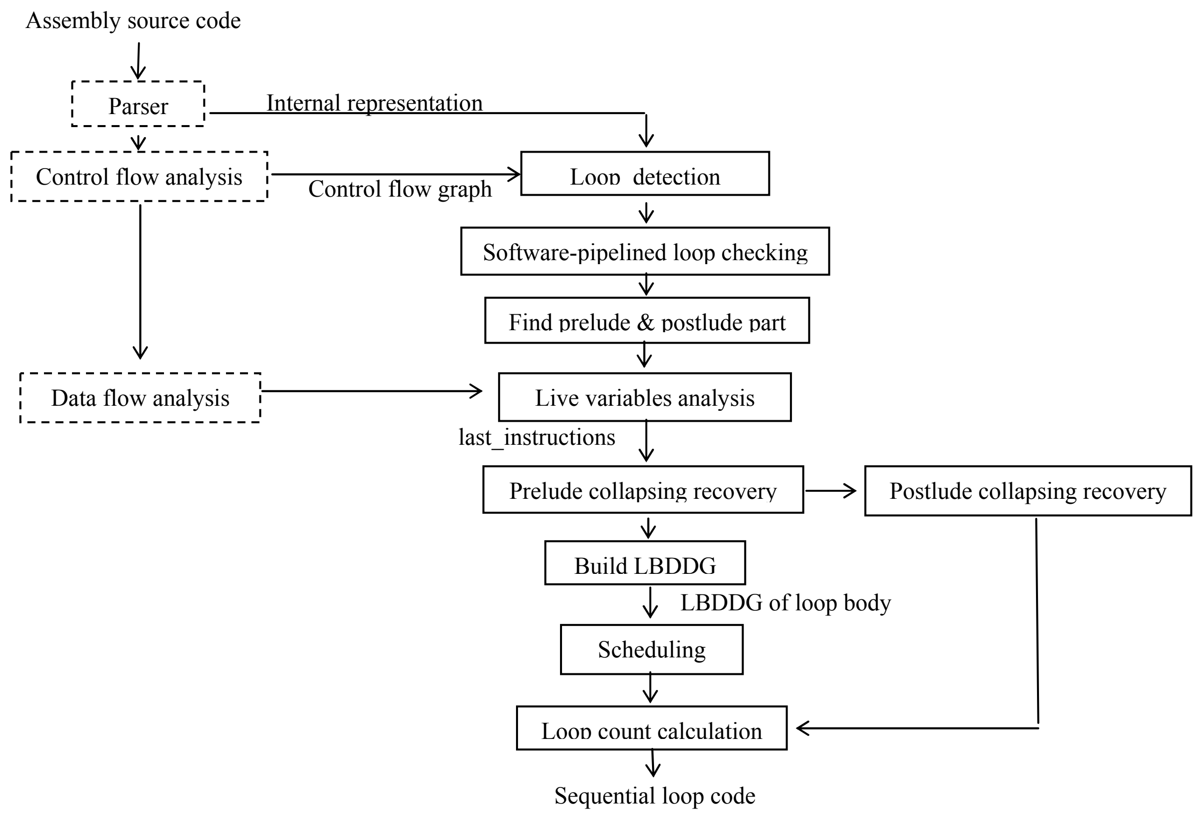


Figure 3.1 Framework of De-pipelining

The **check_spl** function**:**
Input: The output from **detect_loop** function -- the detected loop body in the given segment of the assembly code
Output: A logical value determining whether the loop body is software-pipelined.
Algorithm:
In the given loop body, for any pair of instructions *op_i* and *op_j*, assume *op_i* writes to a variable (register) which is to be read by *op_j* and *op_i* is located not earlier than *op_i* in the loop body, if latency of *op_i* is greater than the distance from *op_i* to *op_i*, then this loop is software-pipelined because *op_i* and *op_i* cannot be in the same iteration.

The **analyze_var** function:
Input: the software-pipelined loop area of the assembly code segment.
Output: all last_instructions.
Definition: all instructions belong to following two categories are defined as last_instructions.
1. The registers written by those instructions are live variables (they will be used after loop exit).
2. All memory store instructions.

Algorithm:
Perform a bottom-up search of the loop for all last_instructions.

The **find_prepost** function:
Input: The software-pipelined loop body and its *DDG*.
Output: The prelude and the postlude of software-pipelined loop.
Algorithm:
1. Starting from loop entry, search upward until reaching the top boundary of the software-pipelined loop (top boundary could be a branch instruction other than forwarding branch instruction to the loop entry or the top of a function code) to find all instruction groups that contain those instructions existing in loop body. The highest instruction group is the upper boundary of the prelude.
2. The lower boundary of the postlude is obtained in a similar manner [19].

The **pre_recov** function**:**
Input: The software-pipelined loop body and its prelude and postlude.
Output: The recovered prelude, if the given prelude has been collapsed.

Algorithm:

1. Assume $I_1,I_2,...I_k$ are VLIW instructions (including NOPs) in the prelude, i.e., prelude = $\{I_1,I_2,...I_k\}$. If the prelude is fully collapsed, then k=0.
2. Unroll the pipelined loop once, denote the instruction group sequence of unrolled loop body as $L_1,L_2,...,L_m$.
3. For i = 1 to m, check every instruction op in L_i. The *op* is a "collapsed instruction" if it can neither cause an exception nor modify any part of the machine state that is "live-in" to its successors (i.e. read within its successors before being written).
4. If no "collapsed instruction" is found in step 3, go to step 5; else, define $LL = \{L_1,L_2,...,L_m\}$ - {all "collapsed instructions"}; update prelude = prelude + LL, go to step 2.
5. In the prelude, delete the dead instructions and the branches caused by "prelude collapsing" if any, and adjust the loop count.

The **post_recov** function**:**
Input: The software-pipelined loop body and its prelude and postlude.
Output: The recovered postlude, if the given postlude has been collapsed.
Algorithm: Similar to **pre_recov** function [19].

The **build_LBDDG** function**:**
Input: The software-pipelined loop body and its recovered prelude.
Output: The ***LBDDG*** of a sequential loop, which is semantic equivalent to the given software- pipelined loop.
Pre-conditions:

1. The loop has been software-pipelined and the prelude is not compacted and/or collapsed.
2. The prelude and the pipelined loop body have been identified.

Algorithm:

1. Denote the prelude and the pipelined loop body as a sequence of instruction groups, $L_1, L_2, L_3, ..., L_k$, where L_1 is the first instruction group in the prelude and L_k is the last instruction group in the body of the software-pipelined loop.
2. Set *DDG* = empty;
3. For i = 1 to k, obtain $L_i' = L_i$ - {those instructions that have been selected in *DDG*}; identify all read-write data dependences among the instructions in L_i' and all write-read, read-write, and write-write data dependencies of the instruction in the *DDG* on the instructions in L_i'; add all the instructions of L_i' to *DDG*; update *DDG*;
4. The *DDG* built in step 3 is the *LBDDG* of a sequential loop that is semantically equivalent to the given software-pipelined loop; return;

The **scheduling** function**:**
Input: The body of a software-pipelined loop and its *LBDDG*
Output: A sequential loop that is semantically equivalent to the given software-pipelined loop
Algorithm:

1. From last_instructions using list scheduling, arrange the partial order list of the critical path of *LBDDG* in a bottom-up manner. The latencies of the instructions must be satisfied; insert NOPs as necessary.
2. Delete all conditional instructions of instructions in loop body except the loop count increment/decrement instructions and the branch instruction that branches to the loop entry point.
3. Insert the rest of instructions in the non-critical paths to the scheduled critical path.
4. Delete all instructions in the prelude and the postlude, which are the same as that in the loop body.

The **calculate_lcount** function**:**
Input: A software-pipelined loop body, its ***LBDDG*** and last_instructions.
Output: The loop count of the semantically sequential loop
Definitions:

Npost:	the number of bottom_last_instruction's copies in postlude, where the bottom_last_instruction is the last_instruction at the bottom of the critical path of *LBDDG*
Nsub:	the number of SUB instructions for decreasing loop counter in prelude.
Nbr:	the number of branch instructions to the entry of the loop body in the prelude.
LCini:	the initial value of loop count in assembly code.
LCsq:	the loop count value in sequential code
LCadj:	the value of the loop count after prelude and/or postlude collapsing recovery
Gsb:	the gap between branch instruction and loop count decrement instruction in the loop body; Gsb =0 if the branch instruction is located not later than the loop count decrement instruction in the loop body; otherwise Gsb = 1.

Algorithm:
Use the following formula to calculate LCsq:

$$LCsq = LCadj - Nsub + Nbr + 1 - Gsb + Npost$$

```
                                                                    MVK 57, A1
                                                ZERO A7    ZERO B7    [A1] SUB A1,1,A1
                                                ZERO A6    ZERO B6    [A1] SUB A1,1,A1  [A1] B  LOOP
                                                ZERO A2    ZERO B2    [A1] SUB A1,1,A1  [A1] B  LOOP
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP
LOOP:  LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7   [A1] SUB A1,1,A1  [A1] B  LOOP
                                                ADD A7,B7,A4
```

Figure 4.1(a) An assembly code segment of TIC62

```
                                                                    MVK 57, A1
                                                ZERO A7    ZERO B7    [A1] SUB A1,1,A1                 }
                                                ZERO A6    ZERO B6    [A1] SUB A1,1,A1  [A1] B  LOOP   }
                                                ZERO A2    ZERO B2    [A1] SUB A1,1,A1  [A1] B  LOOP   }
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP   } Prelude
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP   } area
                                                                    [A1] SUB A1,1,A1  [A1] B  LOOP   }
LOOP:  LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7   [A1] SUB A1,1,A1  [A1] B  LOOP     Loop body
                                                ADD A7,B7,A4
```

Figure 4.1(b) After detect_loop & check_spl

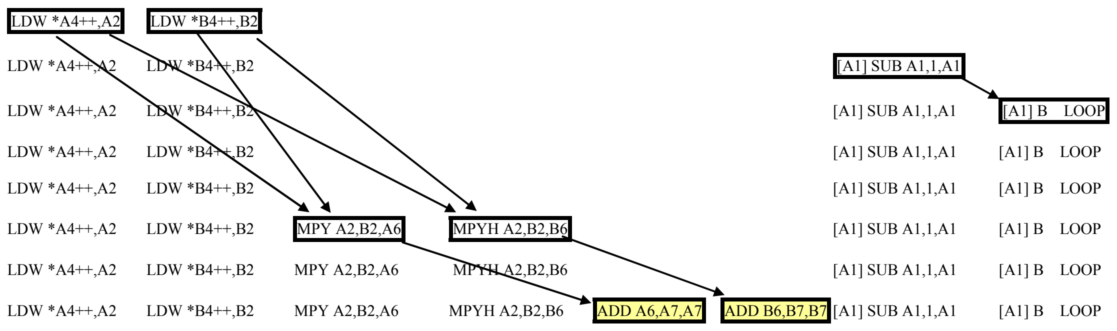
```
                                                ZERO A7    ZERO B7    MVK 50, A1
       LDW *A4++,A2   LDW *B4++,B2                                                                  }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1                }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B   LOOP  }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B   LOOP  } Prelude
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B   LOOP  }
       LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6    [A1] SUB A1,1,A1  [A1] B   LOOP  }
       LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6    [A1] SUB A1,1,A1  [A1] B   LOOP  }
LOOP:  LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7   [A1] SUB A1,1,A1  [A1] B   LOOP   Loop body
                                            ADD A7,B7,A4
```

Figure 4.1(c) After pre_recov

```
                                                ZERO A7    ZERO B7    MVK 43, A1
       LDW *A4++,A2   LDW *B4++,B2                                                                 }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1               }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B  LOOP  }
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B  LOOP  } Prelude
       LDW *A4++,A2   LDW *B4++,B2                                    [A1] SUB A1,1,A1  [A1] B  LOOP  }
       LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6    [A1] SUB A1,1,A1  [A1] B  LOOP  }
       LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6    [A1] SUB A1,1,A1  [A1] B  LOOP  }
LOOP:  LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7   [A1] SUB A1,1,A1  [A1] B  LOOP  Loop body
                                     MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7    }
                                     MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7    }
                                     MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7    }
                                     MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7    } Postlude
                                     MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7    }
                                                                    ADD A6,A7,A7   ADD B6,B7,B7    }
                                                                    ADD A6,A7,A7   ADD B6,B7,B7    }

                                                                    ADD A7,B7,A4
```

Figure 4.1(d) After post_recov



```
LDW *A4++,A2   LDW *B4++,B2
LDW *A4++,A2   LDW *B4++,B2                                                        [A1] SUB A1,1,A1
LDW *A4++,A2   LDW *B4++,B2                                                        [A1] SUB A1,1,A1   [A1] B   LOOP
LDW *A4++,A2   LDW *B4++,B2                                                        [A1] SUB A1,1,A1   [A1] B   LOOP
LDW *A4++,A2   LDW *B4++,B2                                                        [A1] SUB A1,1,A1   [A1] B   LOOP
LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6                         [A1] SUB A1,1,A1   [A1] B   LOOP
LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6                         [A1] SUB A1,1,A1   [A1] B   LOOP
LDW *A4++,A2   LDW *B4++,B2   MPY A2,B2,A6   MPYH A2,B2,B6   ADD A6,A7,A7   ADD B6,B7,B7   [A1] SUB A1,1,A1   [A1] B   LOOP
```

Figure 4.1(e) LBDDG

Figure 4.1 Procedures of the Working Example

4. Working example

We have taken a code segment of TIC62 shown in Figure 1.1 as a working example to demonstrate our de-pipelining technique. The reasons we chose TIC62 are as follows: (1) TIC62 can execute up to eight instructions in parallel. (2) It has a large branch delay. (3) Its compiler uses the sophisticated software pipelining and the prelude and postlude collapsing techniques. All these features make software-pipelined loops more complicated. To de-pipeline the software-pipelined loop, we first apply the **detect_loop** function and identify the instruction labeled with "LOOP" as the loop entry point because it is the target of a backward branch instruction. To explicitly express the instructions that are executed in parallel, we use a new format for the code segment shown in Figure 4.1(a). The initial value of loop count is 57, which is set by MVK 57, A1 instruction. By using the **check_spl** function, we find out that this loop is software-pipelined, because both the latencies of LDW *A4++,A2 and MPY A2,B2,A6 instructions are larger than 1, and these two instructions along with other instructions are within the same cycle. The length of the software-pipelined loop body is one, as shown in Figure 4.1(b).

We then apply the **pre_recov** function to obtain Figure 4.1(c). Based on Figure 4.1(c), we build the *LBDDG* topdown from the beginning of the prelude by using **build_LBDDG** function. Instructions ADD A6,A7,A7 and ADD B6,B7,B7 are the bottom_last_instructions. After applying **post_recov** function, we obtain Figure 4.1(d), which shows the prelude and the postlude of the software-pipelined loop. The value of the loop count after prelude and postlude collapsing recovery becomes 43, which is set by MVK 43,A1. Both bottom_last_instructions have seven copies in the postlude. There are six SUB A1,1,A1 instructions for loop count decreasing and five branch instructions in the prelude.

```
                MVK 50, A1
                ZERO A7
                ZERO B7
LOOP:           LDW *A4++,A2
                LDW *B4++,B2
                LDW *A4++,A2
                LDW *B4++,B2
                NOP
                [A1] B    LOOP
                MPY A2,B2,A6
                [A1] SUB A1,1,A1
                MPYH A2,B2,B6
                ADD A6,A7,A7
                ADD B6,B7,B7

                ADD A7,B7,A4
```

Figure 4.2   Sequential code

Finally we apply the **scheduling** and **calculate_lcount** algorithms to obtain the sequential code with loop count value 50, which is set by MVK 50, A1, as shown in Figure 4.2. The sequential code thus obtained is semantically equivalent to the code segment in Figure 4.1(a) and the original software pipelined code segment in Figure 1.1. By using a simulator we confirm that this is indeed the case.

5. Experiment

We have conducted experiment on 20 different segments of assembly code belonging in different applications for TIC62 and 6 different assembly code segments for SC140. These code segments have different loop lengths and various characteristics of their preludes and the postludes. The TIC62 assembly code segments are generated by either the compiler, or the linear assembler. First, we convert these assembly code segments to sequential code by using de-pipelining technique manually. We then use the simulators to run both the original assembly code and the converted sequential code. All computation results show that the two sets of results are identical thus confirming the validity of our de-pipelining algorithm. Table 5.1 summarizes the characteristics of the software-pipelined loops and the de-pipelining results of these 26 code segments.

6. Applications

6.1 Assembly Code Conversion between two VLIW DSP Processors

Assembly code conversion for DSP has been studied for many years; we have tackled the code conversion from the VLIW source machine. In 2003, we successfully applied our de-pipelining technique on code conversion of software-pipelined loop between two VLIW DSP processors [17]. Our conversion involves the following steps: (1) Using de-pipelining technique to convert the source assembly code of a software-pipelined loop to a semantically equivalent sequential loop; (2) Converting the sequential loop to a machine-independent high- level intermediate code; (3) Converting the machine-independent intermediate code to machine-dependent intermediate code of the target machine; (4) Feed the intermediate code of the target machine to the backend of the compiler of the target machine to obtain the optimized assembly code of the target DSP processor. To facilitate the verification of the correctness of our approach and to evaluate the performance of the converted code, we have chosen those DSP kernel programs whose C source code is available. The C source code is compiled to directly produce the source assembly code for comparison purpose. To verify the validity of the converted code, we have used simulators to compare the results of various steps in the conversion process. In our experiment, the source and target machines are TIC62 and Star*core SC140. Seven DSP kernel programs have been used for the experiments.

The results in Table 6.1 show that the converted assembly code of target DSP processor have comparable performance to the assembly code directly generated by compiling the source code with the optimizing compiler of the target machine.

Table 5.1 Experiment Results

DSP	Program	Characteristics	Software-pipelined loop		After pre/postlude recovery					De-pipelining result	
			Body length	LCini	LCadj	Nsub	Nbr	Npost	Gsb	Body length	LCsq
TI C62	Dot Product_1[2]	Normal	1	43	43	6	5	7	0	14	50
	Dot Product_2[2]	No postlude	1	50	43	6	5	7	0	14	50
	Dot Product_3[2]	Sub & branch in prelude only, no postlude	1	57	43	6	5	7	0	14	50
	Dot Product_4[2]	Branch in prelude only, no postlude	1	51	43	6	5	7	0	14	50
	Dot Product_5[1]	Normal	2	50	50	7	2	4	0	14	50
	FIR[1]	No postlude	3	30	28	0	1	2	0	15	32
	FIRnorld[2]	No postlude	2	16	11	3	2	5	0	15	16
	IIR[2]	No postlude	4	100	97	1	1	3	1	13	100
	Codebook[2]	No postlude	2	32	29	3	2	3	0	9	32
	Vec_mply[1]	Normal	3	75	72	3	1	4	0	16	75
	Latsynth[1]	Normal	5	200	197	2	1	4	1	25	200
	Weighted Vector Sum[2]	No postlude	2	49	46	3	2	4	0	16	50
	Add_test[1]	No postlude	2	5	4	3	2	1	0	6	6
	Loop_test_1[1]	Branch in prelude only, no postlude	2	49	47	1	2	1	0	6	50
	Loop_test_2[1]	Branch in prelude only, no postlude	2	100	99	3	2	1	0	7	100
	Loop_test_3[1]	Branch in prelude only, no postlude	3	100	99	2	1	1	0	7	100
	Loop_test_4[1]	Branch in prelude only, no postlude	5	50	49	1	1	1	1	11	50
	Loop_test_8[1]	No prelude & postlude	9	20	18	-1	0	1	1	21	20
	Loop_test_12[1]	No prelude & postlude	23	25	23	-1	0	1	1	27	25
	Loop_test_16[1]	No prelude & postlude	17	50	48	-1	0	1	1	35	50
SC 140	FIR[1]	Normal	1	49	49	0	0	1	0	4	50
	FIRnorld[1]	Normal	2	15	15	0	0	1	0	11	16
	Vec_mpy[1]	Normal	4	149	149	0	0	1	0	10	150
	Loop_test_2[1]	Normal	2	9	9	0	0	1	0	6	10
	Loop_test_8[1]	Normal	7	24	24	0	0	1	0	12	25
	Loop_test_12[1]	Normal	11	24	24	0	0	1	0	16	25

Note: [1] generated by compiler, [2] generated by linear assembler

Table 6.1 Execution Times Comparison
(in clock cycles)

Program	TIC62 assembly code	Converted SC140 assembly code	Compiler generated SC140 assembly code
Dot product	74	56	53
FIR	12522	23303	38903
IIR	1217	1004	602
Vec_mpy	257	378	602
WVS	1070	204	154
Latsynth	570	1294	1194
Codebook	122	358	615

6.2 Providing Wider Trade-Off Space Between Cycle Count and Code-Size for DSP Applications

Even software pipelining can significantly reduce runtime, it expands code size due to the introduction of prelude and postlude. The size of prelude and postlude grows in proportion to the number of overlapped iterations, which can be large in VLIW DSP processors with many function units. However today's software development often requires an optimum balance between code size and cycle count, which in turn requires a much wider tradeoff space. Based on our de-pipelining technique we proposed a code-size-constraint software pipelining approach [18,19], and demonstrated that the tradeoff space between execution time and memory space can be widened to provide more flexibility for software developers.

7. Conclusion

We present our de-pipelining technique and experimental results. Our approach can be a very useful tool for DSP users to gain insight into the meaning of otherwise very complex software-pipelined code. Furthermore de-pipelining technique can be used to the compatibility issue of VLIW computers. Although this key problem can be solved by using dynamic rescheduling [4], however it cannot solve software-pipelined code. By using our de-pipelining technique we can convert the software-pipelined code of source VLIW processor to sequential code at certain level, then feed into the compiler of other target DSP processor to complete the assembly code conversion.

We are working with more sophistic algorithm to release the restriction (in Theorem 1) of de-pipelining technique. Even our current work is more DSP-oriented, we believe that our de-pipelining technique can be extended to general purpose VLIW and superscalar architecture easily. We will study the application of de-pipelining on code verification.

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. Su and Hu would like to thank the ART awards of William Paterson University

References

[1] C. Cifuentes. and M. Emmerik., UQBT Adaptable Binary Transaction at Low Cost, *Computer*, March, 2000

[2] C. Cifuentes and N. Ramsey, A Transformational Approach to Binary Translation of Delayed Branches with Applications to SPARC and PA-RISC Instructions sets, *SMLI TR-2002-104*, 2002

[3] C. Cifuentes, M. Emmerik, N. Ramsey, and B. Lewis, Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework, *SMLI TR-2002-105*, 2002

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

[5] Decompilation process, www.program-transformation.org/transform/decompilationprocess, 2002

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

[7] E. Granston etc., Controlling Code Size of Software-Pipelined Loops On the TMS320C6000VLIW DSP Architecture, *Proc. of MICRO-34*, 2001

[8] J. Hennessy & D. Patterson, *Computer Architecture, A quantitative Approach*, Morgan Kaufmann, 2003

[9] A. Johnstone, E. Scott, and T. Womack, Reverse Compilation of Digital Signal Processor Assembler Source to ANSI-C, *Proc. of ICMS99*, 1999

[10] Kumbhare R. Optimizing DSP Applications on TMS320C6x, *Proc. of ISPC'03*, 2003

[11] M. Lam, Software Pipelining: An effective Scheduling Technique for VLIW Machines, *Proc. of SIGPLAN 88 Conferenece on Programming Language Design and Implementation*, 1988

[12] Llosa J. and Freudenberger S., Reduced Code Size Modulo Scheduling in the Absence of Hardware Support, *Proc. of MICRO-35*, 2002.

[13] S. Muchnick, *Advanced Compiler Design & Implementation*, Academic Press, 1997

[14] REC – Reverse Engineering Compiler, www.backerstreet.com/rec/rec.htm, 2000

[15] B. Su, S. Ding, and J. Xia, "URPR - An Extension of URCR for Software Pipelining", *Proc. of MICRO-19*, Oct. 1986,

[16] B. Su, J. Wang, E. Hu, and J. Manzano, De-Pipeline a Software-Pipelined Loop, *Proc. of ICASSP03*, 2003

[17] B. Su, J. Wang, E. Hu, and J. Manzano, Assembly Code Conversion of Software-pipelined Loop between two VLIW DSP Processors, *Proc. of the International Signal Processing Conference (ISPC03)*, 2003

[18] B. Su, J. Wang, E. Hu, and J. Manzano, Code Size-Constraint Loop Optimization for DSP Applications, submitted to *EUSIPCO2004*, 2004

[19] B. Su, J. Wang, E. Hu, and J. Manzano,, Software De-pipelining Technique, Technical Report, WPU, 2004

[20] E. Visser, A survey of Rewriting Strategies in Program Transformation Systems, *Electronic Notes in Theoretical Computer Science,* 57(2001), 2001

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