# Instruction Level Loop De-optimization

## Loop Rerolling and Software De-pipelining

**Erh-Wen Hu, Bogong Su and Jian Wang**

**Abstract** Instruction level loop optimization has been widely used in modern compilers. Decompilation—the reverse of compilation—has also generated much interest for its applications in porting legacy software written in assembly language to new architectures, re-optimizing assembly code, and more recently, in detecting and analyzing malware. However, little work has been reported on loop decompilation at instruction level. In this paper, we report our work on loop de-optimization at instruction level. We demonstrate our approach with a practical working example and carried out experiments on TIC6x, a digital signal processor with a compiler supporting instruction level parallelism. The algorithms developed in this paper should help interested readers gain insight especially in the difficult tasks of loop rerolling and software de-pipelining, the necessary steps to decompile loops at instruction level.

**Keywords** Decompilation · Instruction level loop de-optimization · Loop rerolling · Software de-pipelining

## Abbreviations

| | |
|---|---|
| DSP | Digital signal processor |
| DDG | Data dependency graph |
| VLIW | Very long instruction word |
| ILP | Instruction level parallelism |
| TI | Texas Instruments |

E.-W. Hu · B. Su (✉)
Department of Computer Science, William Paterson University, Wayne, NJ, USA
e-mail: sub@wpunj.edu

E.-W. Hu
e-mail: hue@wpunj.edu

J. Wang
Mobile Broadband Software Design, Ericsson, Ottawa, ON, Canada
e-mail: jian.z.wang@ericsson.com



R. Lee (ed.), *Computer and Information Science 2015*,
Studies in Computational Intelligence 614, DOI 10.1007/978-3-319-23467-0_15

## 1 Introduction

Decompilation techniques [8, 9] have been applied to many areas such as porting legacy software written in assembly language to new architectures, re-optimizing assembly code [1], detecting bugs [6] and malware [7]. Decompilation is a complex process typically involves operations such as unpredication and unspeculation [16], reconstructing control structures [21], resolution of branch delays [3], loop rerolling [17] and software de-pipelining [4, 5, 18].

Software pipelining [13] is a loop parallelization technique used to speed up loop execution. It is widely implemented in optimizing compilers for very long instruction word (VLIW) architecture such as IA-64, Texas Instruments (TI) C6X digital signal processors (DSP) that support instruction level parallelism (ILP). To further enhance the performance of DSP applications, software pipelining is often combined with loop unrolling [14]. Therefore, it is often necessary to perform both loop rerolling and software de-pipelining in order to de-optimize loops at instruction level.

Recently Kroustek investigated the decompilation of VLIW executable files and presented the decompression of VLIW assembly code bundles [11]. However the paper did not address the de-optimization of the code at instruction level. In general, loop de-optimization is much more difficult at instruction level than at higher levels because processors that support ILP tend to have more complicated architectures and instruction sets. Furthermore, compilers for these processors often apply various optimization techniques during different phases of compilation in order to better utilize the ILP features of the processors. For example, TIC6x DSP processor contains two data paths and each of which consists of four functional units and one memory port. Thus, TIC6x DSP processor may issue up to eight instructions including two memory fetches at the same time [20].

In the following sections, we first introduce our observation on selected three loops from the functions of EEMBC Telecommunication benchmark [10] and five loops from SMV benchmark [19] and their optimized assembly code generated by the TI C64 compiler. The algorithms used for de-pipelining and rerolling are presented in Sect. 3. A working example along with the experimental results is presented in Sects. 4 and 5. Sections 6 and 7 are related work and our summary.

## 2 Observation

We use data dependency graph (DDG) to represent a loop and follow graph theory to check whether or not a loop is re-rollable and if so, loop rerolling is performed. It is noted that if a loop is unrolled by a compiler, original DDG of the loop is always duplicated, resulting in an identical set of subgraphs referred to as subDDGs in this paper. To facilitate the discussion, we introduce some concepts below:

Two subDDGs G and H are said to be isomorphic if and only if the two subDDGs have the same node sets and any two nodes have a data dependence edge in G, their corresponding nodes in H have the same dependence edge. Isomorphism is an important concept in graph theory. If a DDG can be split into n isomorphic subDDGs, then the loop is re-rollable.

However, compilers often perform addition optimizations after loop unrolling which almost always cause changes to some subDDGs such that not all subDDGs are isomorphic. For example, TI compiler replaces single-word instructions with more efficient double-word instructions. It also uses peephole optimization to remove some instructions in some subDDGs. In fact, after analyzing the TI compiler optimized assembly code of the eight selected unrolled loops from SMV and EEMBC telecommunication benchmarks, we observed that not all subDDGs of the eight loops are isomorphic. In order to reroll the loop, all altered subDDGs must be converted back to isomorphic form.

To systematically tackle the complexity of the conversion process, we subdivide the loops into five different types.

0. A loop whose subDDGs are all isomorphic and all use the same index register and have the same operations on their corresponding nodes.
1. A loop that contains some memory fetch instructions using an additional index register for accessing the same array due to the limitation of instruction format when unrolling too many times.
2. A loop that uses two index registers to access the same array and an additional instruction in some subDDGs to move data across datapath, because memory fetch instruction must use the index register from its own datapath in the TI processor.
3. A loop that uses complex instructions of the TI processor to replace some simple instructions. For example, for performance enhancement a complex double-word LDDW instruction is used by the TI compiler to replace two single-word LDW instructions resulting in two subDDGs to share a single source node.
4. A loop with some of its instructions missing in its subDDGs due to peephole optimization.

Note that except for type_0, loops of all other types contain non-isomorphic subDDGs. As will be discussed in the following section, it is always possible to convert these non-isomorphic subDDGs back to isomorphic form. We name these non-isomorphic subDDGs isomorphicable in the following sections.

Figure 1 shows the categorization of subDDGs. Table 1 summarizes the characteristics of the eight unrolled loops selected as the target of the study in this paper. Table 2 lists subDDG features of the selected unrolled loops.

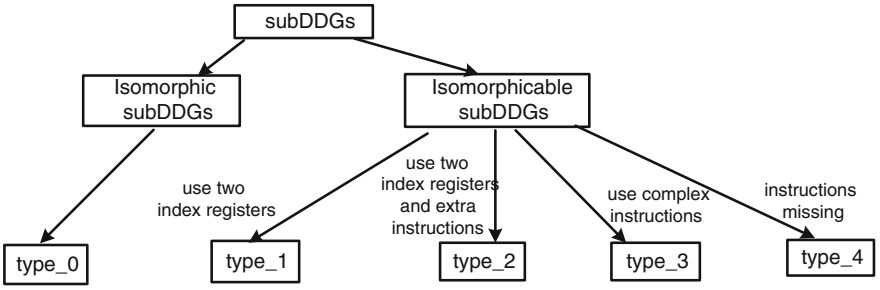



**Fig. 1** Category of subDDGs

**Table 1** Characteristics of unrolled loops

| # | Function Name | Source code | | | | Asm code | | | | Loop optimization applied | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Nest levels | Loop count | | | Nest levels | Loop count | | | | |
| | | | outer | mid | inner | | outer | mid | inner | | |
| 1 | Dot product | 1 | - | - | 100 | 1 | - | - | 50 | unroll x2, then s/w pipelining | |
| 2 | Viterbi Decoder | 1 | - | - | 31 | 0 | - | - | 0 | unroll x31 | |
| 3 | Viterbi StorePaths | 1 | - | - | 32 | 1 | - | - | 7 | unroll x4, then s/w pipelining | |
| 4 | SMV LSF_1 | 1 | - | - | 7 | 0 | - | - | 0 | unrolling x7 | |
| 5 | SMV LSF_2 | 3 | 9 | 128 | 10 | 2 | 9 | 128 | 0 | innermost unrolling x10 | |
| 6 | SMV LSF_3 | 3 | 9 | 128 | 7 | 2 | 9 | 128 | 0 | innermost unrolling x7 | |
| 7 | SMV LSF_4 | 2 | 7 | - | 10 | 1 | 7 | - | 0 | inner unrolling x10 then outer s/w pipelining | |
| 8 | SMV FLT | 2 | 170 | - | 9 | 1 | 85 | - | 0 | 1.inner unrolling x10 2.outer unrolling x2 3.s/w pipelining | outer_0 outer_1 |

## 3 Methodologies and Algorithms

Our methodologies for solving loop rerolling and software de-pipelining are described below:

1. Perform software de-pipelining first, then perform rerolling if the loop has been software pipelined after unrolling.
2. Build data dependency graphs of subDDGs based on the analysis of innermost loops in assembly code. The process begins from the *last_instructions* [18] to help reduce the search space.
3. Find clusters of potential unrolled copies including all isomorphic subDDGs and isomorphicable subDDGs.
4. Convert all isomorphicable subDDGs to isomorphic subDDGs using symbolic calculation, instruction replacing, de-peephole optimization and other techniques.
5. Use single loop to represent all isomorphic subDDGs, which is the rerolled loop.

**Table 2** SubDDG features and de-optimization solution

| # | Function Name | Loop Optimization Applied | | Sub DDGs numbers | Sub DDGs type | Features of Isomorphicable subDDGs | Solution of Loop De-optimization |
|---|---|---|---|---|---|---|---|
| 1 | Dot product | unroll x2, then s/w pipelining | | 2 | 0 | | software de-pipelining |
| 2 | Viterbi Decoder | unroll x31 | | 31 | 1 | two index registers | symbolic calculation |
| 3 | Viterbi StorePaths | unroll x4, then s/w pipelining | | 4 | 2 | one extra MV instruction and two index registers | 1. software de-pipelining<br>2. subDDG adjustment<br>3. symbolic calculation |
| 4 | SMV LSF_1 | unrolling x7 | | 7 | 3 | use complex instructions | use simple instructions to replace complex instruction |
| 5 | SMV LSF_2 | innermost unrolling x10 | | 10 | 3 | use complex instructions | use simple instructions to replace complex instruction |
| 6 | SMV LSF_3 | innermost unrolling x7 | | 7 | 4 | one less instruction due to peephole optimization | de-peephole optimization |
| 7 | SMV LSF_4 | inner unrolling x10<br>outer s/w pipelining | | 10 | 2 | one extra MV instruction and two index registers | 1. software de-pipelining<br>2. subDDG adjustment<br>3. symbolic calculation |
| 8 | SMV FLT | 1. inner unrolling x10<br>2. outer unrolling x2<br>3. s/w pipelining | outer_0 | 10 | 3 | use complex instructions | 1. software de-pipelining<br>2. use simple instructions to replace complex instruction |
| | | | outer_1 | 10 | 4 | 1. use complex instructions<br>2. two subDDGs have no load instruction due to peephole optimization<br>3. some subDDGs have extra MV instructions | 1. software de-pipelining<br>2. use simple instructions to replace complex instruction<br>3. de-peephole optimization<br>4. subDDG adjustment |

Figure 2 shows the flowchart of our loop de-optimization technique. Besides the normal control flow analysis and data flow analysis, we introduce the following 11 functions:

The **natural_loop_analysis** function:
From a given segment of assembly code and its control flow graph, the function finds the dominators, loop nest tree, loop headers, bodies, branches, nested loops, and the lengths of inner bodies. The algorithms of the function are very similar to that of [2].

The **software_pipelined_loop_checking** function:
The function checks all loops to find out whether the inner loop bodies are software-pipelined. If not, the execution of the software de-pipelining function is skipped.

Algorithm:

The algorithm checks for any pair of instructions $op_i$ and $op_j$ in the body of the inner loop and determines if the following conditions are true: (1) if $op_i$ writes to a register which is to be read by $op_j$ and $op_j$ is located not earlier than $op_i$ in the loop body and (2) if the latency of $op_i$ is greater than the distance from $op_i$ to $op_j$. If both of the conditions are true, then this loop has been software-pipelined because $op_i$ and $op_j$ cannot be in the same iteration.

The **software de-pipelining** function:
The function converts software pipelined loops to de-pipelined loop, the detailed description of the algorithm can be found in [5, 18].

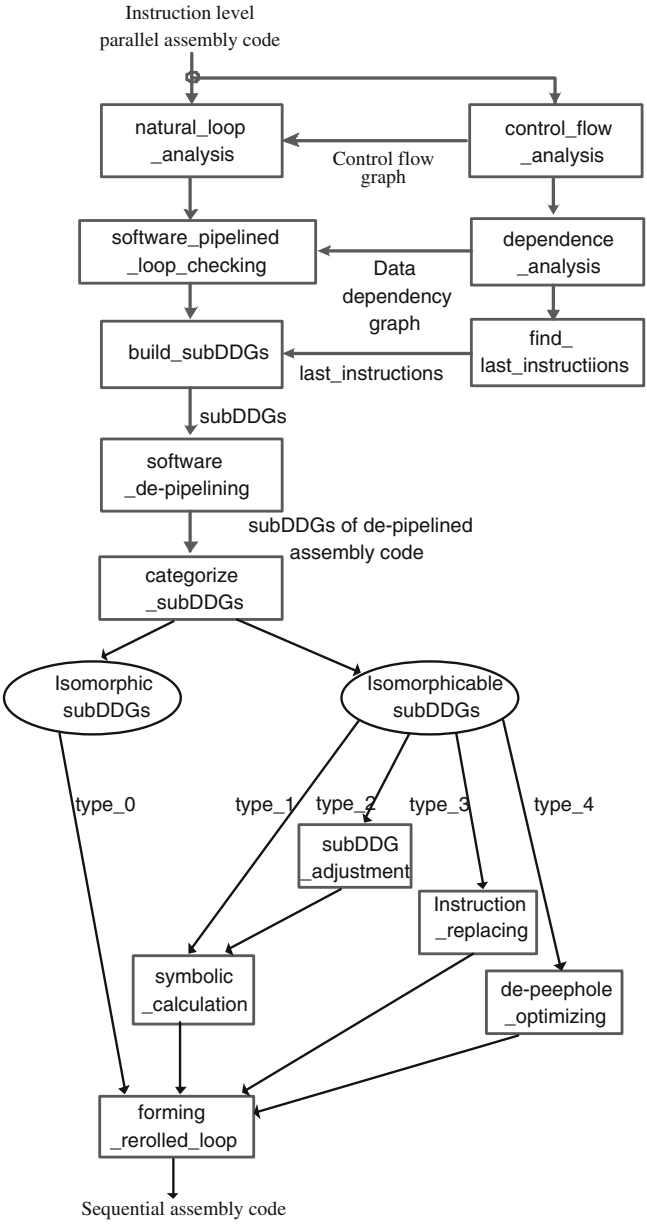



**Fig. 2** Flow chart of loop de-optimization technique

The **find_last_instructions** function:
The function performs a bottom-up search of all de-pipelined loops for all last_instructions. A last_instruction belongs to either of the following two categories: (1) instructions that write to registers involving live variables with transferred values to be used after loop exits and (2) All memory store instructions.

The **build_subDDGs** function:
The function builds subDDGs for the bodies of all inner loops.
Algorithm:

1. Set $subDDG_j$ = {$last\_instruction_j$} for each $last\_instruction_j$ in $subDDG_j$,
2. Define instruction $pool_j$ as the set of all instructions in de-pipelined loop body.
3. Add $instruction_k$ to $subDDG_j$ by performing a bottom up search for $instruction_k$ in instruction $pool_j$ from $last\_instruction_j$ where data precedes any instruction in $subDDG_j$ with true dependence, output dependence, or antidependence.
4. Repeat 3 until the first instruction in de-pipelined loop body has been reached.

The **categorize_subDDGs** function:
The function analyzes subDDGs and determines their types. It then selectively calls other functions depending on the type of the subDDG as described below.

```
Algorithm:
 If subDDGs are isomorphic (i.e., type_0)
    {
    Use the same index registers and call forming_rerolled_loop function to
       reroll all isomorphic subDDGs;
    }
 Else subDDGs are isomorphicable
    {
    If type_1, call symbolic_calculation function;
    If type_2, call call subDDGs_adjustment and symbolic_calculation
              functions;
    If type_3, call instruction_replacing;
    If type_4, call de-peephole_optimization function and other functions;
    }
```

The **symbolic_calculation** function:
This function merges two different index registers. It does so by tracing back to the original source index register, replace it by a virtual register and recalculate all indexes.

The **instruction_replacing** function:
The function replaces a complex 32-bit instruction by two16-bit instructions with the same source and destination registers.

The **subDDG_adjustment** function:
The function applies to type_2 subDDG that uses a MV instruction to move data across datapath. This subDDG is semantically equivalent to the rest of subDDGs, therefore removing that MV instruction does not change the semantics.

The **de-peephole_optimizing** function:
The function recovers removed nodes and converts type_4 isomorphicable subDDG to isomorphic subDDG as some isomorphicable subDDGs have some of their nodes removed due to peephole optimization. For example, in one isomorphicable subDDG of SMV FLT a multiplication instruction node misses a load instruction node to provide its operand because peephole optimization removed this load instruction and the operand of that multiplication instruction is provided by another load instruction shared with another multiplication instruction. Another example is with SMV LSF_3 one isomorphicable subDDG in which one node of MV instruction is removed because the destination register of that MV instruction is dead.

Algorithm:
Compare a type_4 $subDDG_k$ with the isomorphic subDDG

1. If $node_i$ is found in isomorphicable $subDDG_k$ and its preceding node is missing in $subDDG_k$, then:
   i. Find $node_j$' that precedes $node_i$' in isomorphic subDDG where the corresponding $node_j$ in $subDDG_k$ is missing.
   ii. Copy $node_j$' and attach it to isomorphicable $subDDG_k$ such that the attached node precedes $node_i$.
2. If $node_i$ is found in isomorphicable $subDDG_k$ with its succeeding node missing, then:
   i. Find $node_j$' that succeeds $node_i$' in isomorphic subDDG but $node_j$ is missing in isomorphicable $subDDG_k$ as a succeeding node to $node_i$.
   ii. Make a copy of $node_j$' and add it to isomorphicable $subDDG_k$ as a succeeding node to $node_i$. If the destination register of $node_j$ is dead in isomorphicable $subDDG_k$, then convert isomorphicable $subDDG_k$ to isomorphic subDDG.

The **forming_rerolled_loop** function:
The function performs the following operations:

1. Replace all isomorphic subDDGs by a single subDDG.
2. Use list scheduling from last_instructions to arrange the partial order list of this subDDG in a bottom-up manner.
3. Add a backward branch instruction to form the rerolled loop body if no branch instruction in found in this subDDG.
4. Adjust loop count

## 4 Working Example

We have selected the StorePaths function in Viterbi of EEMBC telecommunication benchmark as a working example to demonstrate our loop rerolling and de-software pipelining techniques.

Figure 3a is its assembly code generated by TI C64 complier where each line is an instruction group and all instructions in one instruction group are executed at the same time in parallel.

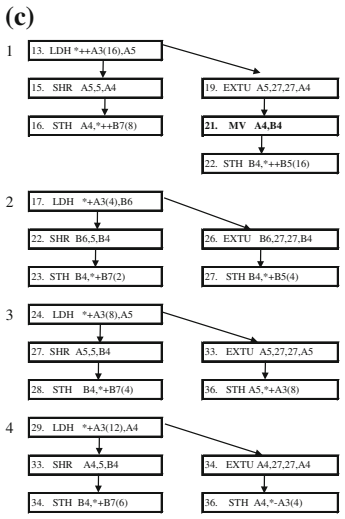
**(a)**

```
1  $C$21:   LDW *+DP(_BufSelector),B4
2           MVKL   _BufPtr,B5
3           MVKH   _BufPtr,B5
4           MVK    7,A0
5           MVK    0x1,A1
6           LDW    *+B5[B4],B4
7           SUB    A4,8,B7
8           NOP    3
9           SUB  B4,16,B5
10          NOP    1
11          MV   B5,A3
12 $C$L4:   ; PIPED LOOP PROLOG
13 $C$L5:   ; PIPED LOOP KERNEL
14          NOP          3
15           SHR   A5,5,A4
16          [!A1]  STH  A4,*++B7(8)
17          [!A1]  LDH   *+A3(4),B6
18           NOP          1
19          EXTU  A5,27,27,A4
20          NOP  1
21          MV   A4,B4
22          [!A1]  STH  B4,*++B5(16)        SHR  B6,5,B4
23          [!A1]  STH  B4,*+B7(2)
24          [!A1]  LDH   *+A3(8),A5
25          NOP    3
26          EXTU   B6,27,27,B4
27          [!A1]  STH B4,*+B5(4)           SHR  A5,5,B4
28          [!A1]  STH  B4,*+B7(4)
29          [!A1]  LDH  *+A3(12),A4
30           NOP          2
31          [ A0]  BDEC $C$L5,A0
32          NOP          1
33          EXTU  A5,27,27,A5               SHR    A4,5,B4
34          [!A1]  STH A5,*+A3(8)           EXTU A4,27,27,A4      [!A1] STH B4,*+B7(6)
35          LDH *++A3(16),A5
36          [ A1]  SUB   A1,1,A1            [!A1]STH A4,*-A3(4)
37 $C$L6:   ; PIPED LOOP EPILOG
38          NOP          3
39          SHR    A5,5,A4
40          STH   A4,*++B7(8)
41          LDH    *+A3(4),B6
42          EXTU  A5,27,27,A4
43          NOP          2
44          MV   A4,B4
45          STH  B4,*++B5(16)               SHR B6,5,B4
46          STH B4,*+B7(2)                  EXTUB6,27,27,B4
47          STH B4,*+B5(4)                  LDH  *+A3(8),A4
48          NOP          4
49          SHR A4,5,B4
50          STH   B4,*+B7(4)
51          LDH   *+A3(12),A5
52          EXTU A4,27,27,A4
53          RETNOP  B3,2
54          SHR    A5,5,B4
55          STH   A4,*+A3(8)                STH  B4,*+B7(6)       EXTU A5,27,27,A5
56          STH     A5,*+A3(12)
57          ; BRANCH OCCURS
```

**(b)**

```
1  $CSDWS21   LDW *+DP(_BufSelector),B4
2              MVKL  _BufPtr,B5
3              MVKH  _BufPtr,B5
4              MVK   8,A0
5              MVK    0x1,A1
6              LDW   *+B5[B4],B4
7              SUB   A4,8,B7
8              NOP    3
9              SUB  B4,16,B5
10             NOP    1
11             MV   B5,A3
12 $C$L5:      ; de-PIPED LOOP KERNEL
13             LDH *++A3(16),A5
14             NOP          4
15              SHR   A5,5,A4
16             STH   A4,*++B7(8)
17             LDH   *+A3(4),B6
18              NOP          1
19             EXTU  A5,27,27,A4
20             NOP  1
21             MV   A4,B4
22             STH  B4,*++B5(16) SHR  B6,5,B4
23             STH  B4,*+B7(2)
24             LDH   *+A3(8),A5
25             NOP    3
26             EXTU   B6,27,27,B4
27              STH B4,*+B5(4)    SHR   A5,5,B4
28             STH    B4,*+B7(4)
29              LDH   *+A3(12),A4
30              NOP          2
31             [A0] BDEC $C$L5,A0
32             NOP          1
33             EXTU  A5,27,27,A5 SHR    A4,5,B4
34             STH A5,*+A3(8)    EXTU A4,27,27,A4   STH  B4,*+B7(6)
35             NOP    1
36              STH  A4,*A3(4)
```

**Fig. 3** Working example. **a** Assembly code of Viterbi StorePaths. **b** After software de-pipelining. **c** SubDDGs. **d** subDDGs adjustment. **e** Indexes of memory load and store instructions. **f** Rerolled loop of Viterbi StorePaths




**(e)**

| Load | Store_1 | Store_2 | |
|---|---|---|---|
| | | old | new |
| *+A3(4) | *+B7(2) | *+B5(4) | *+A3(4) |
| *+A3(8) | *+B7(4) | *+A3(8) | *+A3(8) |
| *+A3(12) | *+B7(6) | *-A3(4) | *+A3(12) |
| *++A3(16) | *++B7(8) | *++B5(16) | *++A3(16) |

**(f)**

```
1  $C$DW$21    LDW   *+DP(_BufSelector),B4
2                MVKL  _BufPtr,B5
3                MVKH  _BufPtr,B5
4                MVK   32 ,A0
5                MVK   0x1,A1
6                LDW   *+B5[B4],B4
7                SUB   A4,8,B7
8                NOP   3
9                SUB   B4,16,B5
10               NOP   1
11               MV    B5,A3
12 $C$L5:        ; Rerolled LOOP KERNEL
13               LDH *+A3(4),B6
14               [ A0] BDEC $C$L5,A0
15               NOP   3
16               SHR  B6,5,B4        EXTU B6,27,27,B4
17               STH  B4,*+B7(2)     STH B4,*+A3(4)
18 $C$L6:        RETNOP  B3,2
```

**Fig. 3** (continued)

The iteration number of this loop body is seven. By using software_pipelined_loop_checking function, it is determined that this loop is software pipelined because register A5 is written by instruction LDH *++A3(16), A5 at line 35 and register A5 is read by instructions SHR A5,5,A4 and EXTU A5,27,27,A4 at lines 15 and 19, respectively; both instructions occur earlier than instruction LDH *++A3(16),A5.

Figure 3b shows the result of the software_de-pipelining function where the iteration number of de-pipelined loop body changes to eight. There are eight STH store instructions as last_instructions found by the find_last_instructions function.

Figure 3c is the result generated by build_subDDGs function. From the categorize_subDDGs function, we find that Viterbi StorePaths has four unrolled loop copies of type_2. The instruction numbers in Fig. 3c tie to the line numbers of instructions in Fig. 3b.

Among the four loop copies, one isomorphicable subDDG has one additional MV instruction generated by TI compiler for the purpose of moving data to another

datapath. Figure 3d shows the semantically equivalent subDDGs before and after the removal of the MV instruction by the subDDGs_adjustment function.

After the above operations, we now have four type_1 subDDGs that are not yet isomorphic. This is because there are one load instruction and two store instructions in each unrolled copy, and the second store instructions of the four unrolled copies use different index registers. After the execution of symbolic_calculation function, all unrolled copies use the same index register for the second store instruction. Figure 3e lists the indexes of all memory load and store instructions, indicating that all subDDGs are now isomorphic and thus rerollable.

Figure 3f is the rerolled loop after the execution of forming_rerolled_loop function, which is semantically equivalent to the original assembly code shown in Fig. 3a. The iteration number of rerolled loop body changes to 32. The comparison before and after loop de-optimization is shown in Table 3, which is discussed in more detail in Sect. 5.

## 5 Experiment

We have chosen eight loop examples to conduct experiments manually. The original sets of assembly code are generated by TIC64 compiler, which are then optimized by loop unrolling and/or software pipelining. Their characteristics are summarized in Table 1. Their subDDG features and the solutions of de-optimization are summarized in Table 2.

Besides Dot product, Viterbi Decoder and Viterbi StorePaths are from Viterbi function of EEMBC Telecommunication benchmark. The other five kernels are from LSF_Q_New_ML_search_fx and FLT_filterAP_fx functions of the SMV benchmark. Table 1 presents the number of nested levels and loop counts of the source code and assembly code; it also shows the optimization methods applied by TI C64 compiler. All examples have loop unrolling; some involve both loop unrolling and software pipelining. In addition, Table 2 presents the characteristics of subDDGs, the types of isomorphicable subDDGs, the causes for their occurrences, as well as the solution for loop de-optimization. Dot product is the simplest example; all its subDDGs are isomorphic subDDGs using the same index register. The function categorize_subDDGs determines it is type_0 and the forming_rerolled_loop function can thus be called immediately. The remaining examples need conversion from isomorphicable subDDGs to isomorphic subDDGs. SMV FLT is the most complicated case, in which the compiler unrolls the inner loop first, and then unrolls the code of outer loops, and finally software pipelines them. Moreover, peephole optimization is used to reduce some instructions, which further complicates the rerolling process. In general, loop de-optimization requires a range of activities and techniques including software_de-pipelining, instruction_replacing, subDDG_adjustment, de-peephole_optimizing, and finally forming_loop_rerolling.

Table 3 presents our experimental results, where #I denotes number of instructions; #IG number of instruction groups; #CC clock cycles which represents the execution

**Table 3** Experimental results

| # | Function name | Original | | | | After de-pipelining | | | | After rerolling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #I | #IG | #CC | #LC | #I | #IG | #CC | #LC | #I | #IG | #CC | #LC |
| 1 | Dot product | 44 | 19 | 113 | 50 | 14 | 12 | 552 | 50 | 11 | 8 | 802 | 100 |
| 2 | Viterbi Find-Metrics | 50 | 38 | 38 | 0 | – | – | – | – | 21 | 15 | 204 | 32 |
| 3 | Viterbi Store-Paths | 65 | 4 | 208 | 7 | 40 | 35 | 211 | 8 | 8 | 6 | 243 | 32 |
| 4 | SMV LSF_1 | 13 | 3 | 13 | 0 | – | – | – | – | 6 | 6 | 37 | 7 |
| 5 | SMV LSF_2 | 86 | 3 | 40 | 0 | – | – | – | – | 13 | 11 | 120 | 10 |
| 6 | SMV LSF_3 | 37 | 9 | 19 | 0 | – | – | – | – | 21 | 11 | 48 | 6 |
| 7 | SMV LSF_4 | 144 | 5 | 125 | 7 | 71 | 38 | 146 | 7 | 10 | 8 | 630 | 70 |
| 8 | SMV FLT | 237 | 92 | 2997 | 85 | 171 | 60 | 5100 | 85 | 29 | 38 | 13260 | outer 170<br>inner 9 |

time of specific code used in the experiment; and #LC loop count. There are three sections in Table 3, the leftmost one is original assembly code, and the rightmost section is the final result of the semantically equivalent sequential code after loop de-optimization. The second section lists certain kernels that have been optimized by software pipelining after loop unrolling by the compiler. Based on the final results of loop de-optimization, it is obvious that code sizes, including both instruction count and number of instruction groups, are reduced while the number of clock cycles is increased.

## 6 Related Work

Since Cifuences and her colleagues presented their work [9], many decompilation techniques have been published [1, 8, 12]. However, few papers tackle deoptimizaton technique and fewer still investigate loops with instruction-level parallel architectures.

Snavely et al. [16] present instruction level deoptimization approaches on Intel Itanium including unpredication, unscheduling and unspeculation. However they did not tackle loop de-optimization and software de-pipelining. Wang et al. [21] apply

un-speculation technique on modulo scheduled loops to make the code easier to understand, however they do not tackle software de-pipelining and loop rerolling.

Loop rerolling has been implemented at source code level in LLVM compiler [15]. Stitt and Vahid [17] use loop rerolling technique for binary-level coprocessor generation, which is the reverse of loop unrolling. They use character string to represent instruction sequence, and then use the suffix tree to represent the character string for efficient pattern matching unrolled loop copies. However, their techniques are applicable only to decompiling assembly code such as MIPS and ARM without instruction level parallelism.

Bermudo et al. present an algorithm for reconstruction of the control flow graph for assembly language program with delayed instructions which was used in a reverse compiler for TI DSP processors [4]. Su et al. present software de-pipelined technique [18] for single-level loops. Their method based on building linear data dependency graph in software pipelined loop can convert the complicated software pipelined loop code to a semantically equivalent sequential loop. Bermudo et al. extend software de-pipelined technique to nested loops [5].

## 7 Summary

We present our instruction level loop de-optimization algorithms which involve software de-pipelining and loop rerolling. Instruction level loop de-optimization can be very complicated, particularly when the assembly code after loop unrolling is combined with peephole optimization. It is noted that although different compilers may generate different optimized assembly code, our approach can be a useful technique to help interested readers gain insight especially in the difficult tasks of loop rerolling and software de-pipelining, the necessary steps to decompile loops at instruction level. In this paper, we consider only loop independent dependency and plan to extend it to handle loop carried dependency in the future.

**Acknowledgments** Su would like to thank the ART awards of William Paterson University.

## References

1. Anand, K., et al.: Decompilation to compiler high IR in a binary rewriter. Technical report, University of Maryland (2010)
2. Aho, A., et al.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley (2007)
3. Bermudo, N., et al.: Control flow graph reconstruction for reverse compilation of assembly language programs with delayed instructions. In: Proceedings of SCAM2005, pp. 107–118 (2005)
4. Bermudo, N.: Low-level reverse compilation techniques. Ph.D. thesis. Technische Universität Wien (2005)
5. Bermudo, N., et al.: Software de-pipelining for nested loops. In: Proceedings of IMCEEME'12, pp. 39–44 (2012)

6. Cesare, S.: Detecting bugs using decompilation and data flow analysis. In: Black Hat USA 2013, https://media.blackhat.com/.../US-13-Cesare-Bugalyze.com (Accessed 2013)
7. Cesare, S., et al.: Malwise—an effective and efficient classification system for packed and polymorphic malware. IEEE Trans. Comput. 1193–1206 (2013)
8. Chen, G., et al.: A refined decompiler to generate C code with high readability. In: Proceedings of the 17th Working Conference on Reverse Engineering (2010)
9. Cifuentes, C.: Reverse compilation techniques. Ph.D. Dissertation, Queensland University of Technology, Department of CS (1994)
10. Hu, E., et al.: New DSP benchmark based on selectable mode vocoder (SMV). In: Proceedings of the 2006 International Conference on Computer Design, pp. 175–181 (2006)
11. Křoustek, J.: Decompilation of VLIW executable files—caveats and pitfalls. In: Proceedings of Theoretical and Applied Aspects of Cybernetics (TAAC'13), pp. 287–296. TSNUK, Kyiv (2013)
12. Křoustek, J., Kolář, D.: Preprocessing of binary executable files towards retargetable decompilation. In: Proceedings of the 8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13), pp. 259–264. IARIA, Nice (2013)
13. Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the SIGPLAN 88 Conference on PLDI, pp. 318–328 (1988)
14. Lavery, L., Hwu, H.: Unrolling based optimizations for modulo scheduling. Proc. MICRO **28**, 328–337 (1995)
15. LLVM: LoopRerollPass.cpp Source File. http://llvm.org/docs/doxygen/html/LoopRerollPass_8cpp.html (Accessed 2014)
16. Snavely, N., Debray, S.: Unpredication, unscheduling, unspeculation: reverse engineering Itanium executables. IEEE Trans. Softw. Eng. **31**(2) (2005)
17. Stitt, G., Vahid, F.: New decompilation techniques for binary-level co-processor generation. In: Proceedings of the International Conference on Computer Aided Design—ICCAD, pp. 547–554 (2005)
18. Su, B., et al.: Software de-pipelining technique. In: Proceedings of the SCAM2004, pp. 7–16 (2004)
19. TELE BENCH, an EEMBC Bench, http://www.eembc.org/benchmark/telecom_sl.php (Accessed 2015)
20. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, SPRU732H ( 2008)
21. Wang, M., et al.: Un-speculation in modulo scheduled loops. In: Proceedings of the 2nd International Multisymposium on Computer and Computational Sciences, pp. 486–489 (2008)