ASSEMBLY CODE CONVERSION THROUGH PATTERN MAPPING BETWEEN TWO VLIW DSP PROCESSORS: A CASE STUDY

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

In this paper, we investigate a new pattern mapping method to convert the assembly code between two VLIW digital signal processors. The method is so named because the pattern of the code is kept unchanged during the conversion process in order to manage complexity. As a case study, we first manually convert the assembly code of a typical DSP application from Texas Instruments' TIC62, the source processor, to a high level intermediate code. We then feed the intermediate code to the SC140 compiler backend to generate the assembly code of the target SC140 processor, which is executed on the SC140 simulator. The result thus obtained compares favorably with that obtained directly by compiling and executing the C code on the SC140 system in terms of both validity and total execution cycles.

1. Introduction

Digital signal processing industry has been growing rapidly over the past few years [1]. Due to the constant need to improve the performance and to address a wide range of applications, the manufacturers of digital signal processors have introduced a variety of processors of different designs over the years. Recently, a new breed of processors based on VLIW architecture has become mainstream. How to take advantage of these newer and more powerful processors by migrating existing code to these processors in a timely manner has become a problem of practical importance.

Although binary code conversion between general-purpose processors has been investigated [2, 3, 4], few work has been done for code conversion between digital signal processors [5]. We have proposed an approach to convert the assembly code of traditional DSP processor to VLIW DSP processor [6]. In this paper, we are motivated to investigate the code conversion at assembly level between two different VLIW processors for the following considerations.

Despite recent progress in compiler technologies, much of the application code is still handcrafted and optimized at assembler level. Besides, as more and more VLIW DSP processors are available, code conversion between VLIW DSP processors becomes a new challenge. In the following section, we present our code conversion method.

2. Code Conversion Method

In this section, we will present our pattern matching code conversion method that converts the optimized assembly code between two VLIW DSP processors.

2.1 Pattern

Before we define the concept of pattern, we describe some important definitions.

Basic Instructions and *VLIW Instructions* - A basic instruction is the instruction that performs only one operation such as addition, multiplication, memory load or store, register data transfer, and in some case, we also consider MAC as a basic instruction. A VLIW instruction is a group of basic instructions that can be issued for execution at the same time. In this paper, when we say instructions without putting VLIW in the front, we imply they are basic instructions.

Basic Block and *Control Flow Graph* - A basic block is an instruction sequence that has only one entry and one exit. We use a directed graph to represent the control flow of a program, which is called a control flow graph. The control flow graph consists of all basic blocks of a program as its nodes and directed edges that specify the execution order of the basic blocks.

Pattern is defined as a control flow graph and a mapping function from each instruction to the node set of the graph. Formally, given a program P, its pattern is defined as (G,B), where G is the control flow graph of P and each node of G denotes a basic block, B is a mapping function - for each instruction I of P, B(I) is the basic block where I belongs to.

Pattern is not a schedule of a program. It doesn't give the information about which instructions will be executed in parallel, or about when an instruction will be issued for execution. But it gives the information about the control flow of a program and about which instructions are grouped into the same basic block.

[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

2.2 Pattern Mapping

The basic idea behind our technique is that we keep the pattern unchanged during the code conversion which greatly reduces the complexity. The pattern mapping technique performs code conversion in five steps: (1) for each basic block, convert all instructions from the source DSP assembly code to the high level intermediate code; (2) copy the source's pattern to the high level intermediate code. (3) conduct pattern simplification to remove machine dependent optimization and obtain a machine independent high level intermediate code (4) convert to target machine intermediate code. (5) feed to the backend of the target machine compiler to obtain the optimized assembly code

2.3 Conversion Algorithm

We now present our code conversion algorithm based on the ideas behind Pattern Mapping.

(1) Given a program, find out all basic blocks and build the control flow graph. The control flow graph denotes the partial order of all basic blocks. This order is used to pick out a basic block for the following conversion.

(2) For each basic block, find out all live-in registers and all live-out registers. A live-in register is the one which is written outside but read inside the basic block. A live-out register is the one which is written inside the basic block but read outside.

(3) Convert the source DSP assembly instructions in this basic block into a high level intermediate code format.

(4) Construct the data dependency graph for this basic block, for those basic instructions in a VLIW instruction, find out all anti-dependencies. A source node will be added to represent the entry of the basic block. If an instruction reads a variable which is written by another instruction outside the basic block, a dependency should be added between the source and this instruction.

(5) Repeat steps (2) to (4) until all basic blocks are done.

(6) Conduct pattern simplification to remove machine dependent optimization and latencies, and obtain a machine independent high level intermediate code.

(7) Convert to target DSP intermediate code.

(8) Feed into target DSP compiler to obtain the optimized assembly code.

3. Experiment

To verify our approach we have conducted an experiment using the popular TIC62 as the source DSP and SC140 as the target DSP. The steps of the experiment is shown in Figure 1.

We use TIC62 compiler to get TI C62 assembly code from C source code. We build DDG and the pattern, and conduct pattern simplification manually on TI C62 assembly code to get a machine independent high level intermediate code which is in a 3-address form. After that we convert it to SC140 intermediate code manually, based on the definition of SC140 architecture and intermediate code format. We then feed this SC140 intermediate code to the backend of SC140 compiler for code generation and optimization. At last we use SC140 simulator to compare the computation result and running time of converted assembly code and the assembly code generated by SC140 compiler directly.

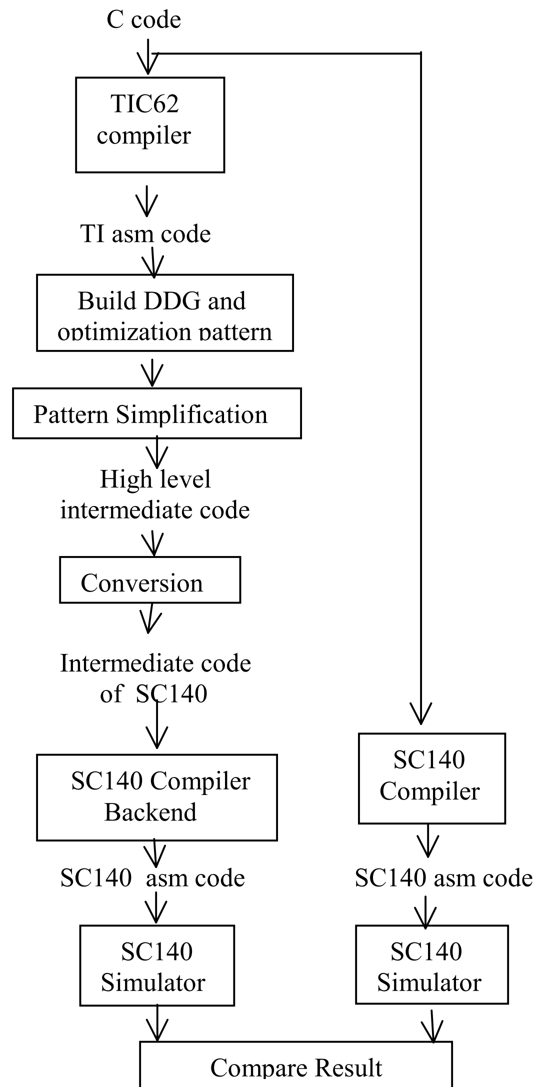

Figure 1 Flow Chart of Code Conversion Experiment

4. Working Example

We select a dot product function as a working example. Figures 2 and 3 show its C source code and the assembly code generated by TIC6 compiler respectively. Figure 4 shows the pattern of this working example; there are three basic blocks in its control graph with the loop body in the middle. The instructions in the basic block frames denote the mapping functions.

```
int dotp(short a[], short b[])
{
  int sum;
  int i;    sum = 0;
  for (i=0; i<20; i++)
  {
    sum+=a[i]*b[i];
  }
  return sum;
}
```

Figure 2 C Source Code

```
1        MV     .L1    A4,A3
2   ||   MV     .S1X   B4,A5
3        ZERO   .L1    A4
4        ZERO   .L1    A0
5        LDH    .D1T1  *+A5[A0],A6
6        LDH    .D1T1  *+A3[A0],A7
         NOP         3
L1:
         NOP         1
7        MPY    .M1    A7,A6,A6
8        ADD    .L1    1,A0,A0
9        ADD    .L1    A6,A4,A4
10       CMPLT  .L1    A0,5,A1
11  [ A1]  B      .S1    L1
12  [ A1]  LDH    .D1T1  *+A5[A0],A6
13  [ A1]  LDH    .D1T1  *+A3[A0],A7
           NOP         3
      ; BRANCH OCCURS
14         B      .S2    B3
           NOP         5
      ; BRANCH OCCURS
```

Figure 3 TIC62 Assembler Code

We convert manually from source assembly instructions in all basic blocks to a high level intermediate code and copy the source pattern to the high level intermediate code as shown in Figure 5. Corresponding to the source assembly code, the intermediate code has three basic blocks. Here we use a regular 3-address code as the machine independent intermediate code. Figure 6 is the SC140 intermediate code converted manually from Figure 5.

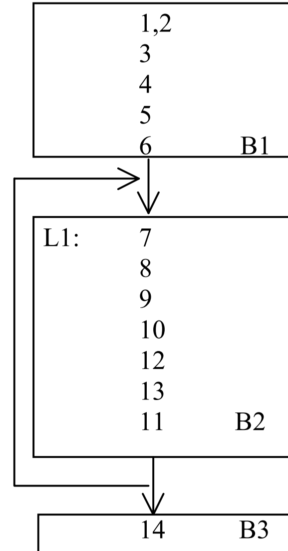

Figure 4 Pattern and DDG of dotp Function

```
;********B1*****************
          sum = 0;
          i = 0;
;********B2*****************
L1:       t3 = t1 * t2;
          sum = sum + t3;
          i = i + 1;
          t1 = a[i];
          t2 = b[i];
          if( i <= 5) goto L1;
;********B3******************
L2:       return sum;
```

Figure 5 High-level intermediate code of dotp Function

Finally we feed this code to the backend of SC140 compiler, and obtain the SC140 assembly code which is shown in Figure 7. Again, it has three basic blocks same as that of the source assembly code. We run this code on the SC140 simulator, its result and running cycles are identical to that of SC140 assembly code produced directly by SC140 compiler from C source code.

5. Discussion

(1) In Figure 3 TI assembly code has two instructions with NOP after the branch instruction, as the optimization of TI compiler moves these two instructions to fill part of branch delay slots. Unfortunately it results in an additional block. Our experience shows that the converted assembly code will be more complicated and less effective due to such kind of machine dependent

optimization. For this reason, the pattern simplification phase is necessary. In fact, Figure 5 is the result after pattern simplification, it switches the order of instruction 11 with 12 and 13, and deletes two NOPs.

(2) Higher level machine dependent optimization of source assembly code such as loop unrolling and software pipelining will make source assembly code more complicated which in turn will make the pattern simplification phase more difficult. We will discuss the impact of machine dependent optimization in another paper.

(3) The variable declaration part of SC140 intermediate code can be produced from the code part. However because of the data type and other architecture differences between the source and target DSP processors, user intervention might be needed.

```
.entry _d41
.arg
  _a, 4 [4], .td -1, .p .si;
  _b, 4 [4], .td -1, .p .si;
.local
  _sum, 4 [4], .td -1, .i;
  _I, 4 [4], .td -1, .i;
.code
 .i _I = .i 0;
 .i _sum = .i 0;
L1:
 .i $t1 =.i _I * .i 2;
 .p $t2 =.p _a + .p:.i $t1;
 .i:.si $t3 = .si *$t2;
 .i $t4 = .i _I * .i 2;
 .p $t5 = .p _b +.p:.i $t4;
 .i:.si $t6 = .si *$t5;
 .i $t7 = .i $t3 * .i $t6;
 .i _sum = .i $t7 + .i _sum;
 .i _I = .i _I + .i 1;
 if (.i _I <= .i 20 ) goto L1;
L2:
 return .i _sum;
.end_entry _d41
```

Figure 6 Converted SC140 Object File

```
;| Block 0, Succ F3
;| Freq 1, Nesting 0
;| Lx: d9,d10,d11,r0,r1
[
   clr     d9                  ;[0]
   doensh3  #19                ;[0] @II1
 ]
 [
   adda    #<24,sp             ;[0]
   move.w  (r0)+,d10           ;[0] 0%=0
 ]
DW_47
   move.w  (r1)+,d11           ;[0] 0%=0
;|
;| Block 3, Pred LB3,F0, Succ LB3,F2
;| Freq 20, Nesting 1, PIPELINED, VISITED
;| Lx: d9,d10,d11,r0,r1
;|
   loopstart3
L13
 [
   imac    d11,d10,d9          ;[0] 1%=1
   move.w  (r1)+,d11           ;[0] 0%=0
   move.w  (r0)+,d10           ;[0] 0%=0
 ]
   loopend3
;|
;| Block 2, Pred F3, Succ R:@NONE@
;| Freq 1, Nesting 0, VISITED
;| Lx: d0
;|
[
   imac    d11,d10,d9          ;[0] 1%=1
   suba    #<24,sp             ;[0]
 ]
DW_50
   rtsd                        ;[0]
   tfr     d9,d0               ;[0]
```

Figure 7 Converted SC140 Assembly Code

Acknowledgement
The authors would like to thank Motorola for the partial support of this project. Hu and Manzano would also like to thank the Center for Research, College of Science and Health, William Paterson University of New Jersey, for research support in the summer of 2001.

References

[1] Strauss, Will; "Digital Signal Processing: The New Semiconductor Industry Technology Driver"; IEEE Signal Processing Magazine, March 2000
[2] Cifuentes C. and Emmerik M.; "UQBT Adaptable Binary Transaction at Low Cost"; Computer, March, 2000
[3] Gschwind, M. et al; "Dynamic and Transparent Binary Translation"; Computer, March, 2000
[4] Zheng, C. and Thompson, C.; "PA-RISC to IA-64: Transparent Execution, No Recompilation"; Computer, March, 2000
[5] DSPOne web site: http://www.dsp1.com
[6] Su B., Wang J., and Hu E., "Code Migration from Conventional DSPs to VLIW DSPs", Proc. of ICPSAT2000, Oct. 2000