Assembly Code Conversion of Software-Pipelined Loop between two VLIW DSP Processors

Bogong Su 1  Jian Wang 2  Erh-Wen Hu 1  Joseph Manzano 1
(973)720-2979  (514) 818-2541  (973)720-2196  (973)720-2649
sub@wpunj.edu  jiwang@nortelnetworks.com  hue@wpunj.edu  manzanoj@student.wpunj.edu

ABSTRACT
In order to fully utilize the instruction level parallelism of VLIW DSP processors, DSP programs have to be optimized by software pipelining. Software pipelining has been studied for many years and widely implemented in optimizing compilers. However, due to the rearrangement of the original instructions, it is often very difficult to re-use or port the code of a software-pipelined loop to other processors.

In this paper we present a practical approach to solve this problem. Our approach involves the following steps: (1) Using a newly developed software de-pipelining algorithm to convert the assembly code of a software-pipelined loop to a semantically equivalent sequential loop; (2) Using our pattern mapping technique to convert the sequential loop to a machine-independent high-level intermediate code; (3) Converting the machine-independent intermediate code to that 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.

We have conducted a series of experiments using the popular TIC62 as the source DSP and SC140 as the target DSP. Six DSP kernel programs have been selected for the experiments. To verify the validity of the converted code, we have used a simulator to compare the results of various steps in the conversion process. All the results of our experiments show that the converted assembly code of target DSP processor are correct with comparable performance to the assembly code directly generated by compiling the source code with the optimizing compiler of the target machine.

1. INTRODUCTION
Digital signal processing industry has been growing rapidly over the past few years. 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 [3]. How to take advantage of these newer and more powerful processors by migrating existing optimized assembly code to these new processors in a timely manner has become a problem of practical importance.

Although binary code conversion between general-purpose processors has been investigated [1], little work has been done for code conversion between digital signal processors. Recently, we have proposed an approach to convert the assembly code from a traditional DSP processor to a VLIW DSP processor [4]. Due to the increasing popularity of the VLIW DSPs, this paper addresses some of the issues of code conversion between VLIW DSPs, particularly on the challenging problem of de-pipelining a software-pipelined loop.

Software pipelining has been studied for many years [2, 7]. It is a loop optimization technique widely implemented in optimizing compilers to speed up the execution of loops on processors that support instruction level parallelism. Due to the transformation of the original sequential code, a software-pipelined loop code is very difficult to understand, test, and debug. It is even more difficult to re-use or port the code of a software-pipelined loop to other processors. So far there seems to have no publication that addresses how to convert a software-pipelined loop or the more general convertibility problem.

In this paper we present a practical approach to solve this convertibility problem by tackling the conversion of a software-pipelined loop. Our approach is based on a pattern mapping technique that we have presented in ICSP’02 [5]. The approach involves the following steps: (1) Using software de-pipelining technique [6], convert the assembly code of a software-pipelined loop to a semantically equivalent sequential code; (2) Using our

Categories and Subject Descriptors
[Software Development Tools]: assembly code conversion tool, conversion method, software pipelining, software de-pipelining.

General Terms
Design

Keywords
Software pipelining, software de-pipelining, VLIW DSP, assembly code conversion, pattern matching.

1 Dept. of Computer Science, The William Paterson University of New Jersey, Wayne, NJ 07470, US
2 Wireless Speech and Data Processing, Nortel Networks, Montreal, QC, Canada, H3E 1H6
pattern mapping technique [5], convert the sequential code to a machine-independent intermediate code; (3) Convert the machine-independent intermediate code to that of a target machine; (4) Feed the intermediate code of the target machine to the backend of the target machine compiler to obtain the optimized assembly code of the target DSP. In the following sections we present in more detail our code conversion method and the experiments.

1. ASSEMBLY CODE CONVERSION METHOD

The framework of our DSP assembly code conversion method is shown in Figure 1. This paper focuses on using de-pipelining technique to convert software-pipelined loop, which includes the following steps. More detailed algorithm and examples of software de-pipelining technique are presented in [6].

(1) Loop detection.
Using the result of control flow analysis and the information of length of branch delay slot, one can find the loop entry and the loop length.
(2) Live variable analysis.
Using the information provided by data flow analysis, one can find all instructions registers written by which are live variables and all memory store instructions. We define those instructions as last_instructions.
(3) Build DDG.
With the help of loop unrolling technique, one can build the Data Dependence Graph (DDG) of the detected loop from all last_instructions bottom up.
(4) Software-pipelined loop checking.
By comparing the distances between instructions in the loop body and in the DDG, one can determine if the detected loop is a software-pipelined loop. If it is not, the loop can be treated by the pattern mapping directly.
(5) Find prelude and postlude parts.
From the body of a confirmed software-pipelined loop, one can find its prelude and postlude parts.
(6) Scheduling.
From last_instructions, one can schedule all instructions bottom up to convert the DDG to a sequential code, which is semantically equivalent to the code of the software-pipelined loop.
(7) Loop count calculation.
It is the most complicated step; one must consider the initial value of loop count in the body of the software-pipelined loop and several other factors in prelude, postlude, and loop body, in order to figure out the loop count of the sequential code.

The sequential loop code and other non-software-pipelined loops of the sample application are fed into pattern mapping phase [5]. The pattern mapping phase converts the assembly code of the source DSP processor to a machine-independent high-level intermediate code; it also removes the machine-dependent optimization for the source DSP processor.

We then convert the machine-independent high-level intermediate code to the intermediate code of the target processor. Finally, we feed it into the backend of the target machine compiler to obtain the optimized assembly code. Because of the data type and other architectural differences between the source and target processors, user intervention might be needed. However, total automatic code conversion is possible with advanced type analysis that provides the necessary data type information.

2. WORKING EXAMPLE

We select a dot product function as a working example. Two most popular VLIW DSP processors TIC62 and SC140 are chosen as the source and target DSP processors, respectively.

Figure 2 presents the code in certain phases of the conversion process: (a) is the C source code; (b) is a hand-optimized TIC62 assembly code that contains a software-pipelined loop, which is based on the assembly code generated by the TIC62 compiler. The vertical bar symbol means the instruction in the current line is executed in parallel with the instruction in the previous line; (c) is the semantically equivalent sequential code in TIC62 assembly language, which is the result of software de-pipelining; (d) is the machine-independent high-level intermediate code generated by the pattern mapping phase; (e) is the intermediate code of SC140 DSP generated by the conversion phase; and (f) is the final result in SC140 assembly code, which has been optimized by the backend of SC140 compiler.

We run TIC62 assembly code and the semantically equivalent sequential (TIC62) assembly code on TIC62 simulator. We also run the converted SC140 assembly code on SC140 simulator. We found that all computational results are identical, which shows our conversion is correct.

3. EXPERIMENT

We have conducted a series of experiments using the popular TIC62 as the source DSP and SC140 as the target DSP. Six DSP kernel programs have been used for the experiments. 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. Figure 3 presents the flowchart of the steps of our experiment. First, we use the simulator of the source DSP to verify the correctness of compiler-generated assembly code, hand-improved assembly code, and the semantically equivalent sequential loop code of the source DSP. Second, we use the simulator of the target DSP to verify the correctness of the converted assembly code of the target DSP. Finally, we use the simulators of both the source DSP and the target DSP to compare the performance of the following three assembly code: (1) the assembly code generated by the source DSP compiler or by hand optimization, (2) the converted assembly code of the target DSP, and (3) the assembly code produced by target DSP compiler.

4. DISCUSSION

The results obtained by running the de-pipelined code of all six kernel DSP programs are in agreement with those obtained by directly compiling and running the corresponding compiled C code on the source simulator. The results are also in agreement with those obtained by compiling and then running the C source code on the target simulator. Table 1 lists the execution time in
clock cycles of all six kernel DSP programs for the three assembly code mentioned above.

It is noted that the execution time of the dot product function is almost identical between the converted SC140 assembly code and the SC140 compiler-generated assembly code. Due to TIC62's longer latencies of memory load and branch instructions, both are shorter than that of TIC62 hand-written code.

Because TIC62 has higher instruction level parallelism, the converted SC140 assembly code of FIR function has a longer execution time than that of the TI compiler-generated code. However, due to the poor code generation performance of the older version of SC140 compiler that we used for the experiments, the converted SC140 code is more efficient than the compiler-generated SC140 code.

We obtain opposite results in the IIR experiment. Because the instruction level parallelism is restricted by loop carried dependency in IIR program, TIC62's longer latencies of memory load and branch instructions cause longer execution time.

Because the compiler for SC140 does not perform software pipelining, both the converted and compiler generated SC140 assembly code of vec_mpy function have longer execution time. Besides, TIC62 compiler conducts loop unrolling, the converted SC140 assembly code is better than that of compiler generated code because it inherits loop unrolling result from TIC62 assembly code.

TIC62's WVS(Weighted Vector Sum) performance is pretty poor, because its compiler does not really perform software pipelining. The long latency of load instruction of TIC62 leads to long execution time. Both converted and compiler-generated SC140 assembly code contain software-pipelined loop which results in a much shorter execution time. Since the code generation part of SC140 compiler uses instructions that move two integer words to/from a register pair, the performance of SC140 compiler-generated code is better than that of converted assembly code.

TIC62 compiler performs very efficient software pipelining for the Latsynth program, some instruction group in loop kernel contains eight instructions, the maximum number of instructions that can be executed in parallel. SC140 compiler could not perform software pipelining for both converted and compiler-generated assembly code, it cause worse performance.

5. CONCLUSION
(1) All the results of our experiments 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 2 shows the normalized execution times from Table 1, normalized to TIC62. The geometric means of the converted SC140 assembly code is a little better than that of TIC62 only, and it is very close to that of the SC140 compiler-generated code.

(2) The converted assembly code inherits the high level optimization from the compiler of the source DSP processor, and it has the same low level optimization as the assembly code generated by target DSP compiler.

---

**Table 1 Execution Times Comparison**

<table>
<thead>
<tr>
<th>Program</th>
<th>TIC62 assembly code</th>
<th>Converted SC140 assembly code</th>
<th>Compiler generated SC140 assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot product</td>
<td>74</td>
<td>56</td>
<td>53</td>
</tr>
<tr>
<td>FIR</td>
<td>12522</td>
<td>23303</td>
<td>38903</td>
</tr>
<tr>
<td>IIR</td>
<td>1217</td>
<td>1004</td>
<td>602</td>
</tr>
<tr>
<td>Vec_mpy</td>
<td>257</td>
<td>378</td>
<td>602</td>
</tr>
<tr>
<td>WVS</td>
<td>1070</td>
<td>204</td>
<td>154</td>
</tr>
<tr>
<td>Latsynth</td>
<td>570</td>
<td>1294</td>
<td>1194</td>
</tr>
</tbody>
</table>

**Table 2 Execution Times Normalized to TIC62**

<table>
<thead>
<tr>
<th>Program</th>
<th>TIC62 assembly code</th>
<th>Converted SC140 assembly code</th>
<th>Compiler generated SC140 assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot product</td>
<td>1.00</td>
<td>0.76</td>
<td>0.95</td>
</tr>
<tr>
<td>FIR</td>
<td>1.00</td>
<td>1.86</td>
<td>1.67</td>
</tr>
<tr>
<td>IIR</td>
<td>1.00</td>
<td>0.82</td>
<td>0.60</td>
</tr>
<tr>
<td>Vec_mpy</td>
<td>1.00</td>
<td>1.42</td>
<td>1.59</td>
</tr>
<tr>
<td>WVS</td>
<td>1.00</td>
<td>0.19</td>
<td>0.14</td>
</tr>
<tr>
<td>Latsynth</td>
<td>1.00</td>
<td>2.27</td>
<td>2.09</td>
</tr>
</tbody>
</table>

| Geo_mean | 1.00 | 0.95 | 0.96 |

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
Figure 1  Framework of DSP Assembly Code Conversion
short start, stop;

int dotp(short a[], short b[])
{
    int sum0, sum1, sum, i;
    sum0 = sum1 = 0;
    start = clock();
    for(i = 0; i < 100 ; i+=2)
    {
        sum0 = a[i] * b[i];
        sum1 = a[i + 1] * b[i + 1];
    }
    stop = clock();
    sum = sum0 + sum1;
    return sum;
}

(a) C source code

(b) TIC62 assembly code

(c) TIC62 sequential assembly code

(d) High-level intermediate code

(e) SC140 intermediate code

(f) SC140 asm code

Figure 2 Conversion of Dot Product Assembly Code
Figure 3  Flow Chart of Code Conversion Experiment