

Code Migration from Conventional DSPs to VLIW DSPs

Bogong Su

Dept. of Computer Science
The William Paterson
University of New Jersey
Wayne, NJ 07470, USA
sub@wpunj.edu

Jian Wang

Wireless Speech and Data Processing
Nortel Networks Montreal
Montreal, QC,
Canada, H3E 1H6
jiwang@nortelnetworks.com

Erh-Wen Hu

Dept. of Computer Science
The William Paterson
University of New Jersey
Wayne, NJ 07470, USA
hue@wpunj.edu

Abstract

It is a challenge to convert the existing conventional DSPs' assembly code to that of VLIW DSPs'. We propose a new code migration method, which performs semantically equivalent code conversion. Key issues such as semantics of the conventional DSP code, specialized addressing modes and different data types are discussed.

1. Introduction

Recently, several families of high-performance VLIW DSP processors have become commercially available. Following the introduction of TI's C6x, other major DSP vendors have announced their VLIW-based DSPs, including ADI's TigerSharc, Infineon's Carmel, and Starcore's SCxxx [WOL98]. According to a recent study [STR00], the VLIW architectures can potentially offer a better cost/performance ratio than comparable conventional DSPs without suffering the code density and memory bandwidth problems. Moreover, due to the hardware resources and the instruction set flexibility, the VLIW architectures allow parallelism to be more easily exploited at the source code level and are easier targets for optimizing compilers [HAL00]. The lacking of effective optimizing compilers has been one of the major problems with conventional DSPs, often requiring manual coding of the applications at the assembler language level. The VLIW-based architectures are not just for high-end applications. For example, the StarCore 400 architecture is scalable [WOL98, WAN00] and can be tailored for a wider range of applications. VLIW DSPs have become mainstream and are expected to be the growing trend for future DSP applications.

Over the years, there have been a great number of applications hand-coded in assembly languages for conventional DSP processors. Although VLIW DSP vendors offer efficient development tools [HAL00], recompiling the existing applications originally

created for the conventional DSPs is often not an option. The corresponding C code is either unavailable or was solely used for the purpose of simulation or algorithm level verification; only the assembly code was fully tested and validated for the applications. Automating the migration of these existing assembler codes from conventional to VLIW DSPs in a timely manner becomes an interesting and important problem.

Several binary code translation systems have been built for general-purpose processors recently [CIF00, GSC00, ZHE00], however to our knowledge, there has no research work been reported on the translation from conventional to VLIW DSP processors. In this paper, we present a new code migration method of directly converting the conventional DSP assembly code to the VLIW DSP assembly code. Section 2 introduces our basic method, section 3 discusses some key issues, section 4 presents our algorithm, and section 5 is a working example.

2. Our Code Migration Method

Due to the large architectural differences between the conventional and the VLIW DSPs, and the diversity of instruction sets of each type, many challenging issues need be resolved. For example, in order to improve performance and to reduce the size of the instructions, the conventional DSPs often incorporate dedicated addressing modes and specialized execution controls such as hardware looping [SAG98, LAP95]. As a result, the tightly encoded instruction sets of conventional DSPs are highly constrained and are not an easily target for optimizing compilers. On the other hand, the control units of VLIW DSPs are much simpler, as there are no dedicated addressing mode and hardware looping, which lead to a highly orthogonal instruction sets and simplification of the programming task.

For the above-mentioned reason, our method is based on a two-step framework. We first transform the conventional DSP code to an intermediate code

with a RISC-like format that will be represented by a control flow graph and a data dependence graph (DDG). We then use a VLIW back-end instruction scheduler to generate the VLIW code. It is noted that in order to use the current VLIW instruction scheduler, the format of the intermediate code should be close to that of an RISC-like instruction set. However, the conventional DSPs are quite different from RISC processors. Thus, the challenge is how to convert the non-RISC code to the corresponding RISC-like intermediate code.

We therefore focus our discussion on issues related to the first step. Namely, we investigate how to convert the conventional DSP assembly code to RISC-like intermediate code. To begin with, it is necessary to understand the semantics of the conventional DSP code. As a case study, we take TI's C6x and Motorola's DSP 563xx as examples. In this paper, we first discuss some key issues related to the semantics of the conventional DSP code, then we present two new concepts - semantic dependence and enhanced data dependence graph (enhanced DDG), finally present our code migration algorithm.

3. Some Key Issues

In this section, we discuss some key issues that are directly related to the code conversion process.

(1) Data types: Some conventional DSPs provide both integer and fraction data types. Different data types may have different semantics in the assembly code. For example, in DSP563xx, the instruction MPY treats its two operands as fractional by default, so it automatically left-shifts the result of the multiplication one bit to get the correct fractional result. However, if the operands are integers, a right-shift ASR instruction must be explicitly appended immediately after the MPY. Specifically, examine the following two instructions:

```
MPY X0, Y0, A
ASR A
```

If X0 and Y0 are integers, the two instructions should be interpreted as $A=X0*Y0$. But if X0 and Y0 are fractional, they mean $A = (X0 * Y0) / 2$.

(2) Addressing modes: Conventional DSPs provide several special addressing modes (e.g. circle buffering), which the VLIW DSPs usually do not support. Different addressing modes may render the same assembly code different semantics. For example, the instruction MOVE X:(R2)+, A can not simply be interpreted as $A = X[R2]$ followed by $R2++$. Its semantics depends on the value of register

M2. If M2 is \$FFFF, the above interpretation is correct. But if M2 is \$F, the instruction should be interpreted as $A=X[R2]$ followed by $R2 = (R2+1) \text{ mod } 16$.

(3) Non-RISC like instructions: The conventional DSP instructions are usually complex. For example, the MAC instruction performs one multiplication and one addition. An instruction may also include parallel move operations in Motorola DSP 563XX.

3.1 Data Dependence Graph and Register Renaming

In order to make this paper self-contained, we first present some definitions about data dependence and data dependence graph. We also discuss how to use register renaming to optimize a data dependence graph. Data dependence can be clarified into the *write-read dependence*, *read-write dependence* and *write-write dependence*. Assume instruction s1 writes its result into register r1, then instruction s2 reads s1's result from r1, finally, instruction s3 writes its result into r1. See the DSP563XX code below.

```
MOVE #\$1,r1; s1
MOVE r1,r2; s2
MOVE #\$2,r1; s3
```

The data dependency between s1 and s2 is called *write-read dependence*. It is the true data dependence, as s2 has to wait until s1's result is available. This dependency cannot be removed by any instruction-level code transformation (e.g. register renaming). The data dependency between s2 and s3 is called *read-write dependence*. It is also called anti-dependence as s2 and s3 can be issued for execution at the same cycle.

The data dependency between s1 and s3 is called *write-write dependence*. Actually, it is a redundant dependency when the dependencies between s1 and s2 and between s2 and s3 have been considered. A *data dependence graph(DDG)* is used to represent all the data dependencies in the code. Each node in the graph denotes an instruction in the code. Each directed edge denotes a dependency -- for instance, if there is a dependency between s1 and s2, there is an edge from node s1 to node s2, as shown in Figure 2.

Read-write dependence and write-write dependence are created when the same register is re-used. These two types of dependence can be removed by an instruction level transformation, called register renaming. For example, in the above example, we modify s3 such that it writes its result into another register, say r3. Then, the dependency between s2

and s3 and the dependency between s1 and s3 disappear.

Register renaming is an important optimization technique for code migration from conventional DSPs to VLIW DSPs. First, conventional DSP processors have fewer registers than VLIW DSP processors. Secondly, when inspecting conventional DSP assembly code, we find that registers are re-used frequently and intensively. It will cause the code to have many data dependencies that prevents VLIW instruction-level scheduling in the second step of our method. When we build a DDG for conventional DSP assembly code, we will find that there are many edges in the graph and it is very difficult to be scheduled.

To start register renaming, we first create a logical register set with unlimited number of registers. Then we use the logical registers to replace the physical registers and never write the same logical register twice. It is shown in Figure 3 and 5.

3.2 Semantic Data Dependence and Enhanced DDG

As we discussed in the beginning of this section, the regular DDG defined in section 3.1 is not sufficient for semantically correctness during the code migration. We present the fourth data dependence - *semantic data dependence* - and enhance the regular DDG.

If instruction s1's result has an impact on the semantics of instruction s2, we say there is a *semantic data dependence* between s1 and s2. Let us see the example we discussed in section 2.1.

```
MOVE #F,M2    ;s1
MOVE X:(R2)+,A ;s2
```

There is not any write-read dependence between s1 and s2, nor read-write, nor write-write. However, any DSP563xx assembly programmer can see there is a data dependency between s1 and s2. It is the semantic data dependence which forces s2's addressing mode to be a circle buffering one.

The *enhanced DDG* is defined as the DDG which may includes semantic data dependence edges and *component instruction nodes*. A component instruction is the one which may contain more than one basic operations (e.g. addition, multiplication, load and store). Examples of component instructions of the conventional DSPs are MAC (one addition and one multiplication), instructions with parallel moves (the basic operation plus loads or stores), shown in Figure 1 and 2.

An enhanced DDG may not be scheduled by a VLIW instruction scheduler for several reasons. First,

semantic data dependence edges may exist so the semantics of some nodes are unknown. Secondly, some nodes may be component instruction nodes that cannot be executed in a VLIW DSP processor. Finally, component instruction nodes are non-RISC like format, thus even they may be executed in the VLIW DSP processor the performance of the VLIW instruction scheduler may be degraded.

Therefore, before the VLIW instruction scheduler is applied, the enhanced DDG needs to be converted into a regular DDG. The conversion actually performs the following two tasks:

(1) Solve all semantic data dependence edges: If there is a semantic data dependency between s1 and s2, the result of s1 should be known during compiling time. Otherwise, the enhanced DDG and its corresponding code cannot be automatically transformed to the VLIW code. After inspecting many DSP applications' assembly code, we find that s1's result is known in compiling time. If s1's result is available, the semantics of s2 can be solved. After solving s2's semantics, the semantics data dependence edge can be removed from the enhanced DDG. If there are no dependence edges attached to s1, s1 can be removed too.

(2) Convert all component instruction nodes into nodes that are of RISC-like format: For each component node, we split it into several simple nodes each of which contains only one basic operation and is of RISC-like format. Some data dependence edges may be added among these simple nodes and the edges to/from the component node may be modified to/from the simple nodes. To implement this, we build a *read-register set* and a *write-register set* for each component instruction node. The *read-register set* contains all registers that are read by the component instruction but were written by other instructions. The *write-register set* contains all registers that are written by the component instruction but will be read by other instructions. These two register sets will be used as a guide to add and modify the data dependence edges.

4. The Algorithm

The algorithm includes the following steps.

(1) Build the enhanced DDG for the given conventional DSP's assembly code. Each instruction is represented as a node in the graph. Map all data dependencies including semantic data dependencies and add the dependence edges to the graph.

(2) Solve all semantic data dependencies and modify the graph.

(3) Convert all non-RISC-like nodes into RISC-like nodes and modify the graph. Now, the enhanced DDG becomes a regular DDG where there is no semantic data dependence edge and all nodes are of RISC-like format.

(4) Apply register renaming to optimize the graph. Now the optimized DDG contains less data dependence edges.

(5) Apply a VLIW instruction scheduler to schedule the optimized DDG.

5. Working Example

We present a working example shown in Figures 1 to 7. Figure 1 is a piece of Motorola 56300 assembly code. Instructions from s2 to s10 are in a loop, which perform a sequence of three array operations: $a[i]=w1*b[i\%10]$, $c[i]=w2*d[i]$, and $e[i]=a[i]+w3*f[i]$, assuming $w1$, $w2$ and $w3$ are stored in X0, Y0, and X1 registers. Instruction s1 is in front of that loop. For simplicity, we omit some loop related instructions here. Figures 2, 4, and 6 are some kinds of DDG, in which we use single line to present true data dependency, dash line to present read-write data dependency, and double line to represent semantic data dependency.

First of all we build an enhanced DDG as shown in Figure 2 from the assembly code in Figure 1. As I set value 9 to M3 such that R3 is a pointer to a circle buffer, and there is a semantic data dependency between s1 and s2. There are three anti-dependencies from s3 to s5, from s6 to s8, and from s3 to s9, the rest are true dependencies.

To solve semantic data dependency we must insert some operations to perform modulo operation, which are p2, p3 and p4 in Figure 3, the RISC-like format. Besides, the component instruction s9 is replaced by two basic operations: p11 and p12 in Figure 3. Figure 4 is the regular DDG built from Figure 3.

Figure 5 is the RISC-like format applying register renaming. Figure 6 is the DDG from it, we can find that all unnecessary read-write data dependencies are eliminated. Figure 7 is the final TIC6 assembly code after some sort of optimization, here $w1$, $w2$, and $w3$ are stored in registers A1, A2, and B2 respectively.

6. Conclusion

This paper presents a new two-step code migration method. The first step is a semantically

equivalent transformation, which converts the conventional DSP's assembly code into a machine-independent intermediate code with a RISC-like format. In the second step, the code generator of a VLIW DSP compiler is applied to the intermediate code to generate the VLIW code.

Compared to the code translation between two general-purpose processors, it is more difficult to convert a conventional DSP code into a VLIW DSP code due to the conventional DSP's non-orthogonal architecture with specialized hardware units. The method presented in this paper is our first effort to tackle this difficulty.

In our future work, we will study in depth our two-step method and its implementation in a practical code translation system. We will study the impact of other specialized hardware units of conventional DSPs (e.g., looping, I/O interfacing) on the code migration. We will also study other potential new issues when we consider the memory mapping and inter-routine code translation.

Reference

- [CIF00] Cifuentes C. and Emmerik M.; "UQBT Adaptable Binary Transaction at Low Cost"; Computer, March, 2000
- [GSC00] Gschwind, M. et al; "Dynamic and Transparent Binary Translation"; Computer, March, 2000
- [ZHE00] Zheng, C. and Thompson, C.; "PA-RISC to IA-64: Transparent Execution, No Recompilation"; Computer, March, 2000
- [HAL00] Halfhill, T.; "Embedded Market Breaks New Ground: Network Processors, Configurable Cores, New Architectures Are Key Trends"; Microdesign Resources, January, 2000
- [LAP95] Lapsley, P., Bier, J., Shoham A., and Lee E.; "DSP Processor Fundamentals: Architectures and Features"; IEEE Press, 1995
- [SAG98] Saghir, Mazen et al; "A comparison of VLIW and Traditional DSP Architectures for Compiled Code"; Proc. Of CASE '98
- [STR00] Strauss, Will; "Digital Signal Processing: The New Semiconductor Industry Technology Driver"; IEEE Signal Processing Magazine, March 2000
- [WANG00] Wang J., Su B., and Hu E.; "A Scalable Loop Optimization Approach for Scalable DSP Processors"; IEEE International Conference on Acoustics, Speech, and Signal Processing, June, 2000
- [WOL98] Wolf O., and Bier J., "StarCore Launches First Architecture"; Microprocessor Report, vol. 12, no. 14, Oct. 26, 1998

```

s1  MOVE #S9, M3
s2  MOVE X:(R3),Y1
s3  MPY X0,Y1,A
s4  MOVE A,X:(R1)+
s5  MOVE X:(R4)+,Y1
s6  MPY Y0,Y1,B
s7  MOVE B,X:(R2)+
s8  MOVE X:(R6)+,Y1
s9  MAC X1,Y1,A
s10 MOVE A,X:(R5)+

```

Fig.1 Motorola 56300 assembly code

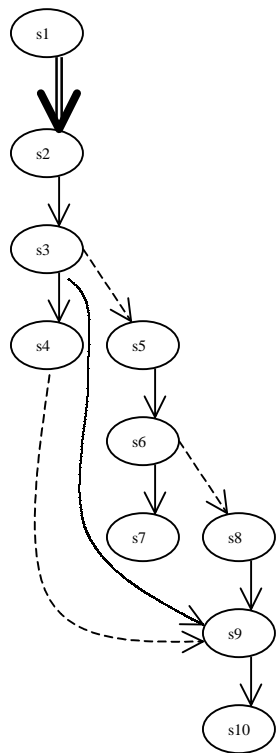


Fig. 2 Enhanced DDG

```

p1  load X(R3),Y1
p2  if R3=9
p3  then R3=0
p4  else R3=R3+1
p5  A=X0*Y1
p6  store A,+X(R1)
p7  load +X(R4),Y1
p8  B=Y0*Y1
p9  store B,+X(R2)
p10 load +X(R6),Y1
p11 B=X1*Y1
p12 A=B+A
p13 store A,+X(R5)

```

Fig. 3 RISC-like format

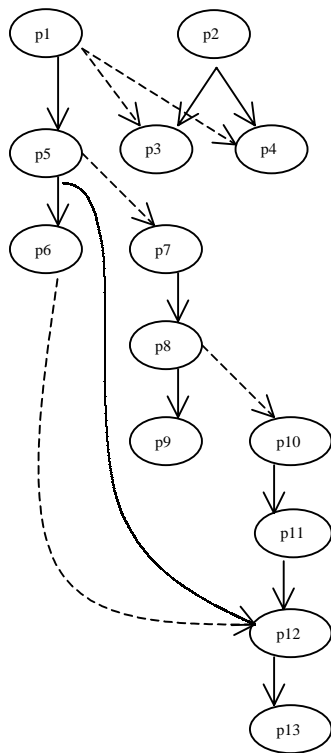


Fig. 4 Regular DDG

```

n1  load X(r4),r5
n2  if r4=9
n3  then r4=0
n4  else r4=r4+1
n5  r6=r1*r5
n6  store r6,+X(r7)
n7  load +X(r8),r9
n8  r10=r2*r9
n9  store r10,+X(r11)
n10 load +X(r12),r13
n11 r14=r3*r13
n12 r6=r14+r6
n13 store r6,+X(r17)

```

Fig. 5 RISC-like format after register renaming

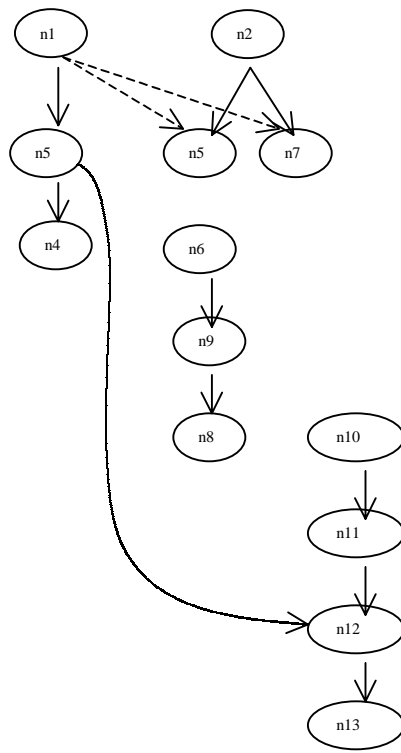
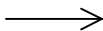

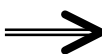


Fig. 6 DDG after register renaming

 Write-read data dependency
 Read-write data dependency

 Semantic data dependency

```

S1  LDH .D1T1 *A5[A0], A8
    || MV .L2X A4, B4
S2  LDH .D1T1 *A6[A4], A9
    || LDH .D2T2 *B7[B4], B10
S3  LDH .D1T1 *A7[A4], A10
    || CMPEQ .L1 9, A0, B0
S4  [B0] ZERO .D1 A0
    || [-B0] ADD .D1 1, A0, A0
S5  NOP 1
S6  MPY .M1 A1, A8, A8
S7  MPY .M1 A2, A9, A9
    || MPY .M2 B2, B10, B10
S8  STH .D1T1 A8, *+A11[A4]
S9  STH .D1T1 A9, *+A12[A4]
    || ADD .D1 A10, A8, A10
S10 STH .D2T2 B10, *+B13[B4]

```

Fig.7 TI C6 assembly code