

面向具有 VLIW 结构 DSP 的汇编级翻译的方法

应 欢^{1,2}, 王雷欧^{1,2}, 薛志远^{1,2}, 王东辉², 侯朝焕²

(1 中国科学院大学, 北京 100049; 2 中国科学院 声学研究所, 北京 100190)

摘 要: 为了简化不同体系结构间代码迁移工作, 提出一种面向具有超长指令字架构的数字信号处理器的汇编级翻译的方法. 前端分析将汇编代码中的指令信息同语义映射为机器无关的中间表示. 采用路径探测法移除分支指令延迟槽构建指令流图, 并重构源程序控制流图. 基于各条指令的时间戳分配和指令间的数据依赖关系分析, 移动代码和修改时间戳来线性化并行代码. 实验证明, 该方法能够正确翻译汇编程序.

关键词: 汇编级; 翻译; 超长指令字结构; 控制流图

中图分类号: TN7

文献标识码: A

文章编号: 1000-7180(2014)09-0020-04

An Assembly-level Translation Scheme Towards a VLIW DSP

YING Huan^{1,2}, WANG Lei-ou^{1,2}, XUE Zhi-yuan^{1,2},
WANG Dong-hui², HOU Chao-huan²

(1 University of Chinese Academy of Sciences, Beijing 100049, China;

2 Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: To simplify code migration between different architectures, an assembly translation solution towards a very long instruction word Digital Signal Processor is presented in this paper. Based on the front-end analysis, valid instructions in assembly files are mapped to machine-independent representation. Through path detection method, the delay slots are eliminated and the instruction flow graph is built, based on which the control flow graph is reconstructed. Additionally, by allocating the timestamp for each instruction, and analyzing data dependency among execution-packet, the linearizing pipelined operation is accomplished by movement and timestamps modification of instructions. The experiment indicates that the assembly programs were correctly translated to the target architecture.

Key words: assembly-level; translation; VLIW architecture; CFG

1 引言

为了使已有的应用软件能够在不同的体系结构上运行, 二进制翻译技术作为解决软件兼容问题的一种重要手段, 成为近年来的研究热点. 二进制翻译系统划分为三个部分: 前端解码, 中端分析优化, 后端代码生成. 中端分析优化是将源体系结构中间表示, 等价映射为目标体系结构中间表示. 当这两种中间表示都是汇编级形式时, 二进制翻译的中端部分可以看作是一种汇编级翻译.

翻译是不同语言形式间的等价转换, 必须保证

源代码的信息能够准确再现. 文献[1]针对带延迟槽的分支指令的汇编程序, 提出建立 CFG 的算法, 该算法根据片段信息将源汇编代码分割成基本块, 并添加可能的跳转分支, 得到的 CFG 中跳转边错综复杂, 从中很难恢复源程序的执行序. 文献[2]针对调度后的汇编代码提出建立控制流图(Control and Data Flow Graph, CDFG)的算法, 但在解决带有延迟槽的指令时, 引入较多的额外指令. 文献[3]从高层的控制流语句角度来分类重构汇编级控制结构. 更多关于控制流图重构的研究^[4-6]基于二进制翻译领域, 针对间接跳转、间接调用指令的目标地址不

确定的情况下,采用程序切片、表达式替换等建立近似 CFG,提高翻译覆盖率。

数字信号处理领域涉及大量的数据运算,为了实现实时高效的计算功能,众多 DSP 具有超长指令字(Very Long Instruction Word, VLIW)架构,完成单周期多发射。经编译器指令调度或手工优化后的汇编代码已经成为高度并行化的代码,这会增加汇编级翻译的难度。本文提出了一种面向具有 VLIW 结构的 DSP 的汇编级翻译的方法,用来实现不同微处理器间的汇编代码的移植。

2 前端分析

汇编代码中涵盖了机器相关的信息,包括指令助记符,汇编伪指令、注释等。汇编文件一般以字符串的形式读入,需要基于已定义的语法、语义约束条件下提取出有效的指令信息,并将这些指令信息以中间语言的形式存储。中间语言是为了隔离源体系结构的机器相关性,从语义角度表达源机器指令的执行行为。为了便于后期翻译,以 TI C6200 DSP 的汇编代码为例,建立中间语言系统如表 1 所示。中间语言系统具有无穷多个虚拟寄存器,翻译后期针对目标体系结构特征,应用传统的寄存器分配算法。

表 1 中间语言系统的定义

TI 汇编指令	IR 表示
SUB B1, 2, A1	Srri% A1-B1-2
ADD A1, A2, A3	Arrr% A3-A1-A2
LDW *-B4[1], A3	Lrm@4% A3*-B4[1]
MV B6, A7	M% A7-B6
[B0] B LOOP	J% LOOP#P[2]
[! A2] MVK 1, A2	Esx% A2-1#! P[1]

3 重构控制流图

编译器设计中的控制流图是对程序中分支跳转关系的抽象,描述程序中所有可能的执行路径,常用于编译调度优化。对于具有 VLIW 结构的微处理器,分支指令是提高其性能的一个重要障碍。等待分支结果的延迟会造成流水线停顿,从而降低指令并行度。硬件设计中的延迟槽结构有助于解决此类控制相关而带来的性能问题,并与编译器配合完成分支指令的延迟槽调度。编译优化中完美的延迟槽调度减少流水线的停顿,但增加汇编级控制流图重构的压力。

3.1 构建指令流图

编译器优化技术中的分支指令延迟槽调度是为了减少分支指令带来的流水线停顿周期,以提升处

理器的性能。其基本思想是将一些与分支指令不相关的有用指令填充到该分支指令后的延迟槽中,这些指令不受分支指令结果的影响,无论分支成功与否,流水线都会执行这些指令。传统复原源程序的控制流的方法是,根据分支指令的执行周期数,将其延迟槽中的所有指令前移。然而,一些具有良好硬件支持的高性能微处理器,其编译器允许延迟槽调度的对象是分支指令。对于这种延迟槽中有其他分支指令的情况,传统的控制流分析方法不再适用。

程序执行过程中,程序计数器(Program Counter, PC)大多数情况下按照固定的步长累加,只有当执行到分支指令时控制走向才被打破。对于特定的目标体系结构,分支指令的延迟槽个数一定。如分支指令 i 的延迟槽个数为 n ,表明指令 i 执行 n 个周期后,PC 才会切换控制走向,跳转至指令 i 的目标地址取指执行。静态分析源程序,根据分支指令的延迟槽信息,从指令级获取程序执行时流经的所有可能控制流,获得指令流图(Instruction Flow Graph, IFG)。IFG 关心程序中的各条指令间可能的逻辑控制流向,是有向图。其中的每个结点代表一条指令,每一条边代表从一条指令到另一条指令可能的执行序关系。汇编代码片段及其相应的 IFG 分别如图 1(a)、(b)所示(假设分支指令带有 3 个延迟槽)。建立 IFG 的伪代码如下。

```

Queue =  $\emptyset$ 
DS_list =  $\emptyset$ 
Node_list =  $\emptyset$ 
start_node = CreateInsnNode(start_insn, DS_list)
Queue.push(start_node)
while(Queue.size() > 0)
    head = Queue.pop()
    ProcessNode(head)
    children = GetChildren(Head)
    for each child in children
        if child is not NULL
            Queue.push(child)
    ProcessNode(Node)
    if Node.insn is branch instruction
        (idx, cnt, tgt, type) = GetBranchInfo(Node, insn)
        DS_list.append({"idx": idx, "count": cnt, "target":
            tgt, "type": type})
    if Node.insn is the last instruction:
        return
    else
        true_node = succ
        false_node = null
        for each branch_DS in DS_list
            branch_DS["count"] -= 1
            if branch_DS["count"] == 0
                true_node = branch_DS["target"]
            if branch_DS["type"] is conditional
                false_node = succ
            break for

```

```

Node_list.append(Node)
for each enode in Node_list
    if enode has same insn and DS_list with false_node, true
_node
        // connect Node's true/false node branch to enode
    else
        //create new nodes and connect it to Node's two
        branches separately

```

3.2 构建控制流图

传统的 CFG 构建包括创建基本块, 添加基本块之间的连接关系. 基本块是程序中的只有一个执行入口和执行出口的指令序列^[7]. 基于文中描述指令间所有可能的执行路径的 IFG, 合并指令序列, 创建基本块, 形成 CFG. 构建 CFG 的步骤如下 (开始时当前基本块 cur_block 为空):

1. 广度优先遍历 IFG 中的各个结点 x_i ;
2. 判断当前结点 x_i 的入度是否大于 1, 如果是, 则说明 cur_block 已经到最远出口位置, 初始化新基本块 $succ_block$, 将当前结点添加到 $succ_block$ 中, 添加边信息 $\langle cur_block, succ_block \rangle$ 至 CFG 中, $succ_block$ 作为当前基本块, 转到 5;
3. 判断当前结点 x_i 的出度是否大于 1, 如果是, 则说明 cur_block 已经到最远出口位置, 将当前结点添加到 cur_block 中, 初始化新基本块 $succ_block$, 添加边信息 $\langle cur_block, succ_block \rangle$ 至 CFG 中, $succ_block$ 作为当前基本块, 转到 5.
4. 添加当前结点至当前基本块 cur_block 中;
5. 重复上述步骤直到遍历完所有的结点.

CFG 中各个结点间的连接关系代表着源程序中各条指令间所有可能的执行序关系, 因此在合并指令序列, 创建基本块时, 跳过 IFG 中原有的分支跳转指令. 图 1(c) 为图 1(a) 对应的 CFG.

4 线性化

VLIW 硬件设计多个功能单元, 编译器遵循数

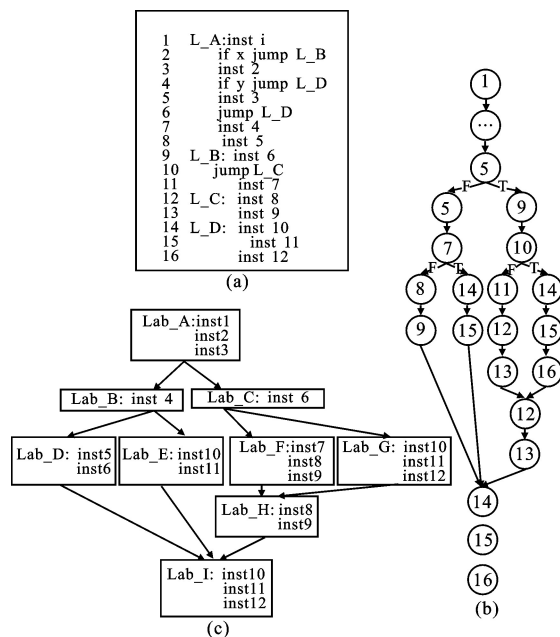


图 1 汇编代码片段及其相应的 IFG、CFG

据依赖约束、控制依赖约束以及目标体系结构的硬件资源约束进行静态调度, 形成包含多个并行字段的超长指令, 在多个硬件功能单元上并行处理. 因此, 汇编代码的并行操作与具体目标体系结构相关.

多周期指令的发射和结果写回不在同一周期内, 如图 2(a) 中 TI 的汇编代码片段 (LDW 指令有 4 个延迟槽) 所示, 设当前执行包在第 i 拍发射, 寄存器 A3, B2 的新值在第 $i+5$ 拍时才可用. 参考文献 [2] 中的时间戳概念, 为每一个执行包分配基准时间 T_b , 当前包中每条指令的发射时间为 $T_i = T_b + 0.01 * n$, 每条指令的写回时间为 $T_w = T_i + \text{delay}$, T_w 为指令的等价执行时间. 图 3(b) 表示经过时间戳定义的汇编代码, 设定当前执行包的基准时间为 1.00. 根据每条指令分配的时间戳信息, 在 CFG 上移动指令, 复原源程序中各条指令的等价执行.

		T_i T_w		T_w	
<pre> [! B1]SUB B4,B10,B4 LDW *-B4[1],A3 LDW *+A4[1],B2 ADD A4,A0,A4 [! B1]SUB A4,4,B11 </pre>	(1)	1.00	1.00	[! B1]SUB B4,B10,B4	(1)
	(2)	1.01	5.01	LDW *-B4[1],A3	(2)
	(3)	1.02	5.02	LDW *+A4[1],B2	(3)
	(4)	1.03	1.03	ADD A4,A0,A4	(4)
	(5)	1.04	1.04	[! B1]SUB A4,4,B11	(5)
				...	
				5.01 LDW *_VR_0[1],A3	(6)
				5.02 LDW *+_VR_1[1],B2	(7)

图 2 TI 汇编代码片段及其时间戳的分配、并行代码线性化

指令间的数据依赖关系主要有三种: 写后读依赖 (Read after Write, RAW), 读后写依赖 (Write

after Read, WAR), 写后写依赖 (Write after Write, WAW). 如图 3(a) 中指令 (3)(4) 在寄存器 A4 上存在 WAR 相关, 指令 (4)、(5) 在寄存器 A4 上存在 RAW 相关. 建立汇编级数据依赖图来描述已调度汇编代码中的执行包内指令间的数据依赖关系. 当存在 WAR、RAW 的数据依赖关系时, 修改指令的时间戳, 避免并行代码线性化后各条指令间的数据冲突. 对于多周期指令, 不仅需要将该指令移动到正确的目标位置, 当与包内其他指令存在数据依赖关系, 还要引入虚拟寄存器来暂存原值. 对于图 3(a) 所示的汇编代码, 经线性化后的汇编代码如图 3(c) 所示. 并行代码线性化伪代码如下:

Linearize(CFG):

```
for each block in CFG:
    for each EP in block:
        CreatADDG(block)
        Tb = AllocTimebase(EP)
        for each insn in EP:
            insn.Tw = CalcWriteTime(insn, Tb)
            hasWAR_RAW = CheckDependRel(insn)
            if hasWAR_RAW is True:
                Tw = ModifyTw(insn)
                time_tmp = int(insn.Tw) //对 Tw 下取整
            if time_tmp > Tb:
                if time_tmp > block_max_time:
                    MoveInsnCrossBlk(insn, insn.Tw)
            else:
                MoveInsnInnerBlk(insn, insn.Tw)
            if hasWAR_RAW is True:
                //在原指令处引入虚拟寄存器暂存原值
                insn.Tw = Tw
```

5 实验

本文以中科院声学研究所自主研发的 SuperV DSP 为目标体系结构, 该处理器具有 VLIW 结构, 支持四发射, 支持推断和预测执行, 应用于通信和多媒体领域, 现已成功流片. 以 TI C6200 DSP 为汇编级翻译的源体系结构, 完成了数字信号处理领域中常用算法的汇编代码的移植, 并在 SuperV DSP 配套开发的模拟器上调试通过. 表 2 中给出了测试程序完成翻译时的相关数据信息. 其中, blocks、edges 分别表示重构 CFG 后的基本块、跳转边的数目, 而额外指令是在完成并行代码线性化时, 由于指令间存在的数据冲突, 需要虚拟寄存器暂存原值时引入的额外指令.

表 2 汇编级翻译实验结果

测试程序	blocks	edges	额外指令
autocor	5	6	12
fir_sym	8	11	8
iir	10	7	21
iirlat	10	7	22
fir_gen	14	16	28
fir_cplx	14	16	26
blk_move	12	8	17

6 结束语

本文面向具有 VLIW 结构的 DSP, 提出一种针对已调度的汇编代码翻译的方法, 尤其对延迟槽中带有其他分支指令的分支指令都能正确处理. 该方法经实验, 证明翻译结果正确, 并可适用于其他具有 VLIW 结构的源、目标机器之间的汇编代码的移植.

参考文献:

- [1] Cooper K, Harvey T, Waterman T. Building a control-flow graph from scheduled assembly code [R]. USA: Rice University, 2002, TR02-399.
- [2] David Zaretsky, Gaurav Mittal, Robert P Dick, et al. Generation of control and data flow graphs from scheduled and pipelined assembly code [C] // LCPC. USA, NY, 2005: 76-90.
- [3] Cifuentes C, Simon D, Fraboulet A. Assembly to high-level language translation [C] // Software Maintenance, Proceedings. International Conference on. USA: Maryland, IEEE, 1998: 228-237.
- [4] De Sutter B, De Bus B, De Bosschere K, et al. On the static analysis of indirect control transfers in binaries [C] // PDPTA. USA: Nevada, 2000.
- [5] Theiling H. Extracting safe and precise control flow from binaries [C] // Real-Time Computing Systems and Applications, Proceedings. Seventh International Conference on. Korea: Cheju Isand, IEEE, 2000: 23-30.
- [6] Kinder J, Zuleger F, Veith H. An abstract interpretation-based framework for control flow reconstruction from binaries [C] // Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg, Springer, 2009: 214-228.
- [7] Appel A W. 现代编译原理 C 语言描述 [M]. 北京: 人民邮电出版社, 2006.

作者简介:

应欢 (1988-), 博士研究生. 研究方向为编译后优化.