基于模式匹配的机器码翻译①

李彭勇, 郑启龙, 郭连伟, 刘 京

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(安徽省高性能计算重点实验室, 合肥 230027)

摘 要: 机器码翻译是把机器码转换成汇编代码的过程, 常用于反汇编、程序调试、病毒分析等领域. 机器码翻译依赖于描述指令集的一系列表格, 包括指令格式表、操作码表、寻址方式表等. 传统的机器码翻译方法, 通过逐步查询这些表格, 找到对应机器码的汇编码, 从而实现翻译. 由于传统方法查表过程复杂, 导致其翻译速度较慢. 本文提出了基于模式匹配的方法, 通过简化机器码翻译的查表过程, 从而在一定程度上提高机器码翻译速度. 同时, 采用该方法实现的程序开发时间更短、后期维护更容易.

关键词: 机器码翻译; 模式匹配; 反汇编; 指令集

Machine Code Translation Based on Pattern Matching

LI Peng-Yong, ZHENG Qi-Long, GUO Lian-Wei, LIU Jing

(School of Computer Science and Technology, USTC, Hefei 230027, China)
(Anhui High Performance Computing key laboratory at Hefei, USTC, Hefei 230027, China)

Abstract: Machine code translation converts machine code into assembler code, commonly used in disassembly, debugging, virus analysis and other fields. Machine code translation is based on a series of tables which are used as the description of the instruction set, including the instruction set format table, the op-code table, the addressing table, and so on. Traditional translation methods query these tables step by step to find the corresponding assembly code. Traditional methods are usually slow because of the complexity of table look-up process. This paper proposes a method based on pattern matching. By simplifying the table look-up process, the speed of the machine code translation to some extent. And at the same time, it's much easier to develop and maintain the translation program by this method.

Key words: machine code translation; pattern matching; disassembly; instruction set

机器语言[1]是计算机能直接识别和执行的唯一语言, 是用二进制码表示的. 由于直接用机器语言编写程序异常困难, 因此现今绝大部分程序都是用某种高级语言编写的, 然后通过编译、汇编、链接等过程转化成机器码. 然而在逆向工程、程序调试、病毒分析等领域, 常常需要将机器码反汇编成汇编码, 以便对程序进行分析. 机器码翻译是反汇编的重要组成部分, 它把二进制机器码转换成汇编代码, 是从机器语言到汇编语言的翻译过程.

机器码翻译依赖于描述指令集的一系列表格, 包括指令集格式表、操作码表、寻址方式表等. 机器码翻译的过程就是对这些表的查表过程, 实质就是找到机器码与指令集的映射. 传统的机器码翻译方法[2] [3], 按照特定指令集的特点, 通过嵌套式地逐层查询这些表格, 找到对应机器码的汇编指令. 其程序结构大致类似于嵌套的 switch/case 结构.

由于查表过程复杂, 采用传统方法实现的代码翻译速度较慢. 同时, 由于不同指令的查表过程不同, 因此基本上需要为每条指令的翻译单独编写一部分代码, 这导致代码繁杂冗长、开发时间长. 当指令集发生改变时, 需要直接修改大量的源代码, 这不利于代码后期维护. 并且, 这种方法基本不具备可移植性[4], 当

① 基金项目: "核高基"重大专项(2012ZX01034-001-001)
收稿时间:2015-02-06;收到修改稿时间:2015-04-26

要实现另一指令集的机器码翻译时, 基本需要完全从头开始编写代码.

本文涉及的 DSP 芯片(BWDSP100)处于研发早期, 指令集处于不断变化中, 因此希望机器码翻译方法易于维护和移植. 同时, 由于机器码翻译在反汇编、调试等程序中皆有应用, 因此希望机器码翻译速度较快. 本文采用基于模式匹配的机器码翻译方法, 在一定程度上提高了机器码翻译的速度. 另外, 这种方法在开发效率、可维护性以及可移植性方面有非常大的改善.

1 BWDSP100处理器及其指令集

1.1 BWDSP100 处理器

BWDSP[5]系列处理器由中国电子科技集团公司第三十八研究所研制, 可广泛运用于各种高性能计算领域, 如雷达、电子对抗、精确制导等. BWDSP100[5]作为该系列的第一款产品, 是一款 32 位浮点 DSP, 它采用 VLIW 架构, 具有强大并行处理能力.

BWDSP100 处理器内核结构为 eC104, eC104 核内部含 4 个基本执行宏(分别用 x、y、z、t 表示), 每个执行宏由 8 个算术逻辑单元、4 个乘法器、2 个移位器、1 个超算器和 1 个通用寄存器组组成.

1.2 BWDSP100 指令集

BWDSP100 指令集[5]总共有 633 条指令, 其中包含很多功能强大的复杂指令, 因此属于复杂架构指令集. 但它又具备许多精简指令集架构的特点: 指令字长固定为 32 位或 64 位, 指令格式简单, 操作码、运算宏位置、长度固定, 并且只有很少的几种寻址方式.

BWDSP100 指令集按功能可分为运算类指令和非运算类指令, 按字长可分为单字指令(32 位)和双字指令(64 位), 其中非运算类指令都是单字指令. 各类指令格式如表 1 所示.

表 1 BWDSP100 指令格式

单字指令格式					
31	30 27	26	25 18		17 0
行标志	运算宏	0	操作码		其它
双字指令格式					
31	30 27	26	25	24 18	17 0
行标志	运算宏	1	0	操作码	其它
行标志		1	1	其它	
非运算类指令格式					
31	30 27	26	25 23	22 18	17 0
行标志	0000	0	000	操作码	其它

在 BWDSP100 指令集中, 对应某些操作码只存在唯一的一条指令, 而对应另一些操作码则存在多条指令. 对于共享同一操作码的几条指令, 通过机器码中的标志位取不同值来区别. 因此, 要唯一确定一条指令, 单是用操作码是不够的, 需要采用三元组<opcode, mode, modeValue>. mode 用于指出标志位在机器码中的位置, modeValue 指出标志位应该具有的值(mode、modeValue 的关系类似于网络中子网掩码与网络号的关系).

2 基于模式匹配的机器码翻译方法

2.1 概述

机器码翻译的实质是找到机器码到汇编指令的映射. 传统的机器码翻译方法, 通过查询描述指令集的一系列表格(指令集格式表、操作码表、寻址方式表等), 从而找到这种映射关系. 由于每条机器指令的翻译都涉及多个表的查询, 导致翻译代码冗长复杂、翻译速度较慢. 基于模式匹配的机器码翻译方法并不直接使用这些表, 而是从指令集本身出发, 通过建立模式表、匹配表, 从而直接建立机器码与汇编指令间的映射关系. 由于这种映射是在翻译之前预先建立起来的, 因此基于模式匹配的翻译方法的翻译过程得以简化、翻译速度得以提升.

模式表存储了指令集中所有指令所具有的模式, 而匹配表记录了指令集中所有指令到模式表的映射以及用于翻译的额外信息. 结合模式表和匹配表, 就建立起了机器码到指令模式的映射, 即三元组<opcode, mode, modeValue>到指令模式的映射. 结合机器码与指令模式, 便能完成对机器码的翻译. 模式表和匹配表可以独立于代码写在配置文件中, 从而实现了指令集相关的部分与代码的分离. 当指令集发生改变时, 基本只需要修改包含此二表的配置文件, 而不需要修改代码, 这大大改善了代码的可维护性[6].

2.2 模式表的结构及建立方法

2.2.1 模式表结构

模式表的表项是一个字符串, 称为模式串, 模式串是对指令的泛化表示, 即将指令中需要翻译的部分用一些特殊符号替代而形成的字符串. 模式串与指令间是一对多的关系, 比如寄存器加法系列指令(Rs=Rm+Rn, Rs=Rm+Rn(U), hRs=hRm+hRn 等)对应同一个模式串: “%R = %R + %R %U”.

模式串中需要翻译的是形式为“% X”的部分(比如

“%R=%R+%R %U”中的“%R”和“%U”). X 代表任意大写字母, 称为格式符, 指出当前翻译部分的类型, 如这里的 R 表示寄存器. 对于每种格式符, 都存在与之对应的若干修饰符(修饰符将用于 2.3 节介绍的匹配表中), 这些修饰符用于说用格式符的一些属性. 修饰符用小写字母表示, 比如上例中格式符 R 的一个修饰符为 h, 表示该寄存器为 16 位的. 表 2 例举了本文用到的部分格式符及对应的修饰符.

表 2　BWDSP100 使用的部分格式符

格式符	含义	修饰符	输出示例
M	运算宏		x, y, xyt
R	寄存器	o, h, f, c	hR2, ohR3
E	寄存器对	o, h, f, c	E1:2, cfE1:2
C	立即数	u, f, c	1, –1, 0x34
U	无符号运算	u	(U)
P	子程序名		foo
…	…	…	…

2.2.2 模式表建立方法

建立模式表的首要步骤是确定需要用到哪些格式符. 格式符是对指令集中需要翻译的部分(比如指令“Rs+=C”中, 需要翻译的是寄存器编号 s 和立即数 C)的泛化表示, 因此必须先对指令集进行分析归纳, 将指令集中需要翻译的内容进行分类. 对每一种需要翻译的类型, 确定一个格式符与之对应. 通常, 寄存器、立即数、子程序名、跳转标签等是每种指令集必备的格式符.

在将指令集中需要翻译的内容归类时, 存在一个粒度问题. 粒度粗, 类别少, 相应地, 格式符数目少, 但对应每种格式符的修饰符数目多. 比如, 表示寄存器的格式符“R”与表示寄存器对(Rs+1:s)的格式符“E”可以合并成一个格式符(假设为 T), 但合并后, 为了区分单一寄存器与寄存器对, 势必为格式符 T 增加若干修饰符. 修饰符数目过多, 将导致模式表、匹配表的建立比较困难, 并且对模式的处理也将较复杂. 因此, 需要合理选择分类的粒度. 一个简单的原则是, 是否可以合并掉一个分类, 应根据此次合并增加的修饰符数量确定.

确定格式符后, 就可以开始建立模式表. 先为每一条指令写出其对应的模式, 然后合并掉重复的模式, 便形成最终的模式表. 需要注意的是, 在写出每条指令对应的模式时, 需要记住指令到模式的映射, 这个映射是匹配表的部分内容. BWDSP100 指令集对应的模式表共包含 107 个模式, 表 3 给出了其部分内容.

表 3　BWDSP100 使用的模式表部分

编号	模式串	对应指令举例
0	%R=%R+%R %U	hRs=hRm+hRn(U)
1	%R=%R–%R %U	hhRs=hhRm+hhRn
2	%R+=%C %U	Rs+=C(U)
3	%R=(%R+%R)/2	Rs=(Rm+Rn)/2
4	%R=%R&%R	Rs=Rm&Rn
5	%R=max(%R,%R)	Rs=max(Rm,Rn)
…	…	…

2.3 匹配表的结构及建立方法

2.3.1 匹配表结构

匹配表的表项(简称匹配项)为五元组<opcode, mode, modeValue, patternIndex, flagsStr>, 前三项含义如前文 1.2 所述. patternIndex 指出对应该指令的模式串在模式表中的索引(即表 3 中的“编号”列). flagsStr 是一个字符串(简称修饰串), 包括两部分内容: 模式串中各个格式符的值所在机器码中的位置, 以及各个格式符具有的修饰符. 格式符的值, 对于不同格式符有不同含义, 对于表示寄存器的格式符 R, 表示寄存器编号, 而对于表示立即数的格式符 C, 则表示该立即数的数值.

以指令 hRs=hRm+hRn 为例, 其对应的匹配项为<35,0x300,0x100,18,”17–12:h;11–6:h;5–0:h”>, 表示其 opcode 值为 35, mode 值为 0x300(表示标志位位于机器码的第 8–9 位), modeValue 值为 0x100(表示机器码的第 8–9 位值必须为 01), patternIndex 值为 2(表示该指令的模式在模式表中的索引为 2). flagsStr 值为“17–12,h;11–6,h;5–0,h”, 表明修饰串包含对 3 个格式符的说明(分号分隔), 第一个分号前的是对 hRs 的说明, 表明该寄存器的编号(即 s 的值)在机器码中的第 12 至 17 位, 并且是个 16 位寄存器.

2.3.2 匹配表建立方法

匹配表的建立过程就是针对每一条指令建立一个匹配项的过程. 以指令 hRs=hRm+hRn 为例, 其对应的匹配项建立过程为: 从指令集说明中获得匹配项的前 3 个子项, patternIndex 可以根据 2.2.2 中提到的映射获得. 修饰串的确定, 即是确定模式串中每个格式符的值所在机器码的位置以及对应的修饰符. 该指令中有 3 个格式符, 都为 R, 而对应 R 需要确定其值(即寄存器编号所在机器码位置)以及位数(这里为 16 位, 需要修饰符 h). 匹配表的大小与指令集的大小相同, 即长度为 633. 表 4 给出了匹配表的部分内容.

表 4 BWDSP100 使用的匹配表部分

opcode	mode	modeValue	patternIndex	flagsStr	指令
7	0x030000	0x000000	0	“11–8;7–4;3–0; ”	Rs=Rm+Rn
7	0x030000	0x010000	0	“11–8;7–4;3–0; u”	Rs=Rm+Rn(U)
7	0x030000	0x020000	0	“11–8, h;7–4,h; 3–0,h; ”	hRs=hRm+hRn
7	0x030000	0x030000	0	“11–8,f;7–4,f;3–0,f; ”	fRs=fRm+fRn
12	0x020000	0x000000	2	“11–8;7–0;”	Rs+=C
12	0x020000	0x010000	2	“11–8;7–0;u”	Rs+=C(U)
21	0	0	5	“11–8;7–0;”	Rs=C
…	…	…	…	…	…

2.3.3 匹配表查询算法

匹配表(mt)的查询算法是比较简单直观的. 首先以操作码 opcode 为键, 二分搜索(binarySearch)匹配表. 假设找到的记录为 item, 将其 mode 值与机器码(mc)做与运算, 若结果等于 modeValue(即 item.mode&mc==item.modeValue), 则找到对应匹配项, 否则分别往前、往后遍历匹配表. 算法如图 1 所示. 假设指令集大小为 N, 则匹配表大小为 N, 因此匹配表的搜索时间复杂度为 O(logN+M)(M 为对应同一 opcode 的指令条数, 均值为 633/256=2.47). 匹配表的查询算法如下所示:

```
/*以 opcode 为键, 二分搜索匹配表*/
item = binarySearch(mt, opcode);
if(item.mode&mc == item.modeValue)
  return item; //find it
/*从二分搜索找到的位置开始, 前向顺序搜索匹配表*/
iter=item;
while(iter != mc.end()){
  if(iter.opcode>opcode)
    return NULL;
  if(iter.mode&mc == iter.modeValue)
    return iter; //find it
  iter++;
}

/*从二分搜索找到的位置开始, 后向顺序搜索匹配表*/
iter=item;
while(iter != matchingTable.begin()){
  if(iter.opcode<opcode)
    return NULL;
  if(iter.mode&mc == iter.modeValue)
    return iter; //find it
  iter--;
}
```

2.4 编写格式符的处理方法

模式表、匹配表的建立, 将对指令集中上千条指令的处理统一为对为数不多的若干种格式符的处理. 通过查询匹配表、模式表找到对应机器码的模式串后, 遍历该模式串, 若遇到格式符, 则调用对应的格式符处理方法, 若遇到非格式符(比如“+”、“=”等)则直接输出. 如此, 便实现了机器码翻译.

这里以格式符 R 为例, 其处理方法如下:

(1)首先从匹配项中的修饰串 flagsStr 中取得对格式符 R 的说明部分, 比如为“11–6:h”;

(2)取出“11–6:h”冒号前短横线分隔的两个整数 11 和 6, 读取机器码中第 6–11 位, 转化为无符号整数(假设值为 2);

(3)取出“11–6:h”冒号后的修饰符 h, 表明为 16 位寄存器;

(4)该部分的输出为“hR2”.

2.5 机器码翻译过程

机器码翻译过程如图 1 所示.

由于 BWDSP100 指令集分为单字指令、双字指令、非运算类指令三类, 因此机器码翻译的首要步骤是确定指令的类型. 然后, 从机器码中取出操作码并以操作码为键查询匹配表, 找到对应机器码的匹配项. 根据找到的匹配项中的 patternIndex, 查询模式表, 找到对应模式串. 遍历该模式串, 若遇到格式符, 则调用对应的格式符处理方法, 若遇到非格式符则直接输出. 如此, 便输出了对应指令的汇编码.

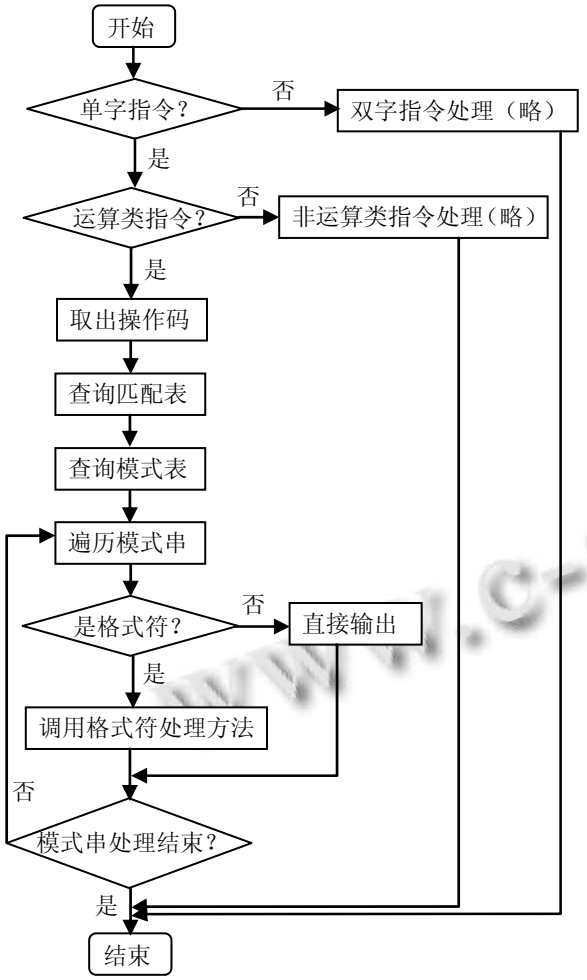

图 1　机器码翻译流程图

3　性能分析

BWDSP100 提供了一套完整的软硬件开发环境, 包括集成开发环境 ECS 和仿真器[5]. 本文使用 ECS 完成了程序的开发调试, 并基于仿真器进行了相关实验.

采用基于表的传统方法(简称传统方法)实现的机器码翻译程序, 因此程序的开发时间与指令集大小直接相关[2]. 而采用基于模式匹配的方法, 程序开发时间与指令条数没有直接关系. 表 5 是采用两种方法实现的翻译程序在开发和维护方面的时间对比. 可以看出, 基于模式匹配的方法, 程序开发时间更短、维护更容易.

表 5　两种方法开发维护对比

比较项	模式匹配方法	传统方法
开发时间/人月	1.8	2.2
代码长度/行	18000	24000
版本升级平均维护时间/人月	0.45	0.6

本文以 Linux 系统文件操作常用命令(可执行文件)作为实验数据, 在 BWDSP 仿真器中分别采用传统方法和基于模式匹配的方法对它们进行机器码翻译, 实验结果如表 6 所示.

表 6　传统方法与模式匹配方法性能对比实验结果

实验程序	指令数(条)	传统方法翻译时间(MS)	模式匹配方法翻译时间(MS)	传统方法所耗内存(KB)	模式匹配方法所耗内存(KB)
pwd	3564	34	31	2528	2765
head	4861	38	34	2534	2774
cat	7372	44	38	2543	2784
rm	8685	49	43	2548	2795
ls	11181	54	48	2556	2802
mv	20238	74	60	2590	2833
cp	21029	76	63	2593	2837
mv	22266	79	65	2597	2839

由于两种方法都是先整体读入机器码文件, 然后逐条进行翻译, 因此时间空间效率都正比于指令条数. 在空间效率方面, 由于基于模式匹配的方法要预先读入模式表、匹配表, 因此需要多耗费一定量(约 240KB)的内存. 在时间效率方面, 虽然基于模式匹配的翻译方法需要多花费一定的时间创建模式表、匹配表(约 5ms), 但由于其预先建立起了机器码与模式间的映射关系, 使得机器码翻译的查表过程简化为对模式表、匹配表的查询. 查表过程的简化使得每条指令翻译时间得以在一定程度上缩短. 当指令条数较多时, 加速比可达到 1.2.

4　相关工作

传统的机器码翻译方法可分为两类: 基于表(table–based)的方法, 例如 HDTrans[7]、JudoDBR[8]等; 基于中间表示(IR–based)的方法, 例如 Valgrind [9]、RIO

[10]等. 本文分别从实现难易程度、翻译速度以及适用性方面, 将这两种传统方法与基于模式匹配的方法作简要对比.

(1)实现难易程度: 基于表的方法, 实现方式较为简单, 但由于不同指令对应不同查表过程, 因此翻译代码的开发涉及许多简单而重复的工作; 基于中间表示的方法, 依赖于一个机器码与汇编码间的中间表示, 因此实现较为复杂; 基于模式匹配的方法, 实现的复杂点在于模式表的建立, 因此, 实现的难易程度与指令集格式的复杂程度直接相关.

(2)机器码翻译速度: 基于表的方法, 由于需要逐步查询多个表格, 因此翻译速度较慢; 基于中间表示的方法, 其机器码翻译速度更多的取决于基于中间表示实现的各种优化; 基于模式匹配的方法, 虽然需要额外的开销建立模式表、匹配表, 但由于机器码翻译过程的简化, 使得翻译速度在一定程度上得以提升.

(3)适用性: 基于表的方法基本可用于任何指令集的机器码翻译; 基于中间表示的方法, 依赖于机器码与汇编指令间的中间表示, 因此一般基于一些像 LLVM 等的开源架构; 基于模式匹配的方法, 要求指令集格式较为简单规整, 因此适用于 ARM 等精简指令集以及指令格式简单的 BWDSP 等复杂指令集.

5 结语

本文提出了一种基于模式识别的机器码翻译方法. 相较于传统方法依赖于描述指令集的一系列表格, 该方法从指令集本身出发, 通过建立模式表、匹配表, 从而在翻译之前预先建立起了机器码与汇编指令的映射关系. 实验结果表明, 该方法在翻译速度方面也有一定提升. 另外, 由于该方法具有易开发、易维护的特点, 因此特别适用于芯片研发早期、指令集不断变化的场景.

参考文献

1 McCabe M. Machine language (review). Computer Music Journal, 2009, 33(4): 90–91.

2 Pfeffer TF, Herber P, Schneider J. Reverse engineering of ARM binaries using formal transformations. Proc. of the 7th International Conference on Security of Information and Networks. 2014. 345.

3 Song WI, Zeng YJ, Xi Q. Code disassembly technology combining dynamic and static state. Computer Engineering, 2012, (1): 68–70.

4 Shen BY, Hsu WC, Yang W. A retargetable static binary translator for the ARM architecture. ACM Trans. on Architecture and Code Optimization, 2014, 11(2), article no: 18.

5 CETC 38. BWDSP 100 Software User Manual, 2013.

6 Nands S, Li W, Lam LC, et al. BIRD: Binary interpretation using runtime disassembly. Proc. of the International Symposium on Code Generation and Optimization. 2006. 358–370.

7 Sridhar S, Shapiro JS, Bungale PP. Hdtrans: a low-overhead dynamic translator. ACM SIGARCH Computer Architecture News, 2007, 35(1): 135–140.

8 Olszewski M, Cutler J, Steffan JG. Judostm: A dynamic binary-rewriting approach to software transactional memory . Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques. 2011. 365–375.

9 Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. 2007. 89–100.

10 Hazelwood K, Smith MD. Managing bounded code caches in dynamic binary optimization systems. ACM Trans. on Architecture and Code Optimization, 2012: 263–294.