

文章编号:1001-9081(2005)10-2439-02

代码翻译中 Case 语句的识别和恢复

苏 铭,赵荣彩,齐 宁

(信息工程大学 计算机科学与技术系,河南 郑州 450002)

(mingsu@sohu.com)

摘 要:提出了在开发 IA-64 二进制翻译系统中采用的 n-条件分支跳转表和目标地址恢复技术。着重论述了该技术的核心——过程内切片和表达式替换,以及针对 IA-64 特性的改进算法。

关键词:n-条件分支;程序切片;二进制翻译

中图分类号:TP314 **文献标识码:**A

Recovery of Case statement from binary code

SU Ming, ZHAO Rong-cai, Qi Ning

(Department of Computer Science and Technology, Information Engineering University, Zhengzhou Henan 450002, China)

Abstract: This paper presented a technique for recovering jump tables and their target address used in IA-64 binary translation system, and focused on explaining the key part of this technique-slicing and expression substitution.

Key words: n-conditional branches; binary translation; indexed jump

0 引言

随着对软件依赖性的增加和新的快速体系结构的不断开发,对机器代码的移植,仿真,调试,跟踪和剖析工具的需求越来越多。这些工具大致可以分为:二进制翻译^[1],模拟/仿真,代码插桩,反汇编和反编译。

在机器指令的解码中,需要解决的根本问题是如何将代码与数据分离,在冯诺曼体系结构中二者用相同的形式表示。对可执行程序来说,入口点在程序头给出。代码段和数据段的信息在程序头也可以得到。但是,当数据存放在代码段时,相应的信息在程序头中无法得到。这样,就需要分析程序代码段解码后的代码,将数据与代码分离。

机器代码解码的标准方法是沿着从入口点开始的所有可达路径^[2]。该方法当遇到间接转移语句时无法给出对代码段的完全覆盖。解决这个问题经常采用模式匹配,该技术广泛应用在针对特定编译器的处理工具中。例如,处理 Windows 平台上 Microsoft C++ 编译器产生的代码的跟踪工具。但是当代码存在优化,模式匹配就无法有效的工作。

本文针对 IA-64 跳转表的特性,提出了一种与机器和编译器无关的恢复跳转表和它们的目标地址的解决方案,该方案的基础是过程内切片技术^[3]和表达式替换技术。通过这些技术,可以将包含索引跳转的过程代码转换成一般的形式,同时可以确定跳转表装入位置和包含的信息。最后通过实验可以说明,该方案可在很大程度上提高二进制翻译的覆盖率。

1 n-条件分支编译后生成的代码

n-条件分支最先由 Wirth 和 Hoare 在 1966 年提出,当时是作为 Algol 语言的一种有用的扩展。n-条件分支允许在代码中确定 n 个分支中的一个。换句话说,case 语句的结果可以赋给一个变量。高级语言,如 C 中广为人知的 switch 语句以及 Pascal 中的 case 语句,对不同的条件分支使用标号,而

且允许缺省分支。例如下面例子中的 C 代码,对索引变量 num 进行检测,如果位于范围 2 至 7,则执行对应的动作,如果不成功,则执行缺省动作。switch C 语言程序例子如下:

```
#include <stdio.h>
int main()
{
    int num;
    printf("Input a number, please: ");
    scanf("%d", &num);
    switch( num ) {
        case 2:
            printf("Two! \n"); break;
        case 3:
            printf("Three! \n"); break;
        ...
        case 7:
            printf("Seven! \n"); break;
        default:
            printf("Other! \n"); break;
    }
    return 0;
}
```

编译器设计者在为 n-条件分支产生代码时重点考虑空间和速度的优化。其中最简单的方式是使用语句中每个分支比较结果的线性序列。这种形式对于分支较少的情况比较高效,例如 4 个或更少。另外一种技术是 if-tree,通过在组织成树形的比较集合中进行选择。最通用的实现方式是跳转表,表中可以包含标号或者特定标号的偏移。这种实现要求对范围进行检查,以确定该值是否属于跳转表。当 n-条件分支的分支比较稀疏,但是包含在一定范围时,一种通用的技巧是将查找树和跳转表进行合并。在本文中,主要探讨如何从生成的跳转表中来恢复代码,从而判定一个索引跳转的目标地址。

对于使用跳转表实现的 n-条件分支,要根据分支的每

收稿日期:2005-04-20;修订日期:2005-06-28 基金项目:河南省杰出人才创新基金资助项目(0521000200)

作者简介:苏铭(1974-),女,博士研究生,主要研究方向:计算机软件与理论; 赵荣彩(1957-),男,教授,博士生导师,主要研究方向:计算机软件与理论; 齐宁(1978-),男,博士研究生,主要研究方向:计算机软件与理论。

一个 case 的地址或偏移建立索引表。该表存放在只读数据段或者在代码段。从效率的角度考虑,跳转表范围的检测必须简单。最常用的方法如下:

- 1) Case 的选择值减去下界,结果赋值给 K;
- 2) 比较 K 与上界和下界的差值之间的关系;
- 3) 如果大于,则跳转到越界处理;
- 4) 否则,则表明 case 的选择值位于上界与下界之间。

如果 case 的选择值位于上界和下界之间,那么可以基于表项的大小计算跳转表中的偏移,不同的架构表项的大小不同,对于 IA-64,表项长度为 8 字节。然后在根据体系结构的寻址方式的不同,得到跳转目标为表的地址加上偏移,或者直接为得到的偏移。

2 二进制代码中索引跳转的例子

对 IA-64 体系结构来说,如果编译器选择产生跳转表,该表被放置在代码段,每一个表项为表的基址与跳转目标之间的 64 位偏移。该偏移在链接时静态的确定,而在程序启动时无需重定位。根据与跳转表基址的偏移,代码能够简单的通过将偏移与表基址相加计算得到跳转目标地址。

下面是一个简单间接转移的例子,跳转表中存放每一个跳转目标与表基址相对的 64 位偏移。switch 程序 IA-64 汇编代码片段如下:

```
ld4 r14 = [r36]           |读下标变量
;;
...
adds r14 = -2, r14         |减去下界
cmp4.ltu p6, p7 = 5, r14   |检测范围
(p6) br. cond. dptk . L10   |越界,执行
;;                          |L10
adds r14 = 4, r36          |乘以机器字长
addl r14 = @ltoff(. L11), gp |计算链接表的入口
;;
ld8 r14 = [r14]           |装入跳转表
;;
add r14 = r15, r14         |计算跳转表项的入口;;
...
add r14 = r15, r14         |计算目标地址
;;
mov b6 = r14              |将目标地址放入 b6.
br b6                     |索引跳转
4000000000000780 30 00 00 00 00 00
4000000000000786: 00 00 60 00 00 00
400000000000078c: 00 00 00 00
4000000000000790: 90 00 00 00 00 00
4000000000000796: 00 00 c0 00 00 00
400000000000079c: 00 00 00 00
40000000000007a0: f0 00 00 00 00 00
40000000000007a6: 00 00 20 01 00 00
40000000000007ac: 00 00 00 00
```

该例子使用寄存器间接跳转,下标变量初始化为 r14,从栈的局部变量开始,计算下界并将下标变量放在 r14。检测表的范围,如果越界,则在 L10 处终止。反之,计算表的地址到 r14。索引寄存器乘以 8 获得在表中的偏移放在 r15。根据偏移查表中对应项,结果放入 r15,该值为跳转目标与跳转表首地址之间的偏移。最后计算跳转的目标地址结果放到 r14,然后执行到 r14 的跳转。跳转表的位置为地址 4000000000000780 到 40000000000007a0 之间的部分。

3 解决办法

IA-64 二进制翻译系统首先将二进制代码转换成由寄存器转移列表(RTL)和它的过程控制流图(CFG)组成的中间表示。其中,RTL 通过寄存器转移描述了机器指令的效果,而且能够支持大部分架构的指令语义描述。本文采用了一种从中间表示中恢复跳转表分支的算法,它与体系结构,编译器和语言无关,具体流程如下。

算法流程:

- 1) 解码遇到索引寄存器跳转,则在该处对二进制代码的 RTL 中间表示创建了一个过程内的反向切片^[5]。
- 2) 切片沿着索引跳转表达式中使用寄存器的传递闭包进行。切片结束的标志:该寄存器从内存装入(该条件对 IA-64 不适用);通过另外一个函数返回;或者是未经定义就到达一个过程的开始。

3) 执行拷贝传播以恢复伪高级语句。

- 4) 通过对跳转地址表达式的形式正规化进行模式匹配,判断是否为跳转表。成功,判定表的类型,表首地址,计算跳转目标;否则,返回;在这个算法中主要用到了过程内切片技术和表达式替换技术。

3.1 过程内切片

程序切片是根据对控制和数据流分析而引入的一种程序分析技术。它实际上是一种分解技术,从程序中抽取与特定计算相关的语句片段,以获得源代码的行为,程序片段一般可以通过可达性分析来获得。近来,切片广泛应用于源码的调试,维护和并行化等研究中。在二进制代码工具的研究领域,切片技术已经应用于二进制剖析工具 qpt,并通过重写 RISC 可执行文件来测量程序的行为。另外,在 EEL 中也嵌入了切片技术。

对于二进制翻译系统来说,不仅要讲二进制文件解码,而且还需将指令的信息和过程的控制流图转换成一种低级的中间表示,作为分析和代码提升的基础。基于这种中间表示的切片技术是 Horwitz 提出的通用静态切片算法^[3],后来经过 Agrawal 扩展后称为通用 goto 算法^[4]。当进行二进制代码的切片时,使用基本块的控制流图(CFG)作为分析的中间表示,通过寄存器和条件码的引用一定义链来表示程序数据相关性,每个过程的控制相关性通过构建控制相关图(CDG)和后支配树(PDT)确定。基于 PDT 和 CFG,如果索引跳转和返回指令包含在指令到达路径中,则加入到切片中。其中,构造 PDT 的时间复杂度为 $O(N\alpha(N))$,这里 N 为 CFG 中节点的个数。CDG 构建的时间复杂度为 n^2 。

3.2 表达式替换

一旦计算出一个切片后,通过向前替换来执行表达式替换(对寄存器和条件代码执行拷贝传播)。根据文献[4],指令 i 中的寄存器 r 按照寄存器集 a_k 的定义, $r = f_1(\{a_k\}, i)$, 可以使用 r 在指令 j 上的应用进行向前替换, $s = f_2(\{r, \dots\}, j)$, 如果在 i 处的定义是 r 在程序中沿所有路径到达 j 的唯一的定义,而且在该路径中不存在寄存器 a_k 被重新定义。那么在 j 处的结果指令如下:

$$s = f_2(\{f_1(\{a_k\}, i), \dots\}, j)$$

同时在 i 处的指令就可以消失。前面的关系可以通过指令的定义—引用(du)以及引用一定义链(ud)获得:如果只有一条指令到达某寄存器,则对寄存器应用的定义唯一,也就

(下转第 2443 页)

8) 当源节点 S 收到 RREP,就用新路由代替路由标识为弱状态的路由(见图 3(b))。

当路由表中的路由标识为弱状态时,传送数据包到下一跳节点还是可能的,但是此时节点即使收到 RREQ 也不会传送 RREP,尽管此时路由是有效的。而且在与下一跳节点之间的链路中断时也不会传送 RERR。路由表中的条目也要经过一段固定的时间后才会失效。

通过以上步骤的实施,路由中断就可以避免了,通信和传输就不会收到中断的影响。

3 结语

本文提出了一种基于 AODV 的新的路由维护机制 AODV-BA,用来避免路由的中断。活性路由上的中间结点探测链路中断的可能主要基于四个方面的要素:接收到的无线

信号,路由的重叠,能量的供应和节点的密度。而避免路由的中断主要是在链路中断前就重新建立一条新的路由来代替原有的路由。AODV-BA 比 AODV 具有更高的执行效率。

参考文献:

- [1] PERKINS C. Nokia Research Center, E. Belding-Royer. Ad hoc On-Demand Distance Vector (AODV) Routing[S]. RFC3561, 2003 - 7.
- [2] CASTANCEDA R, DAS SR. Query Location Techniques for On-demand Routing Protocols in Ad Hoc Networks[A]. proc of ACM International Conference on Mobile Computing and Networking(MOBI-COM)[C], 1998.
- [3] MARINA MK, DAS SR. Ad Hoc On Demand Multipath Vector Routing[A]. Proc of ACM SIGMOBILE Mobile Computing and Communications Review Special Feature on the First AODV Next Generation Workshop[C], 2002.

(上接第 2440 页)

是说,它的 ud 链集合只有一个元素。这种关系称作寄存器 r 的 $r - clear_{i \rightarrow j}$ ^[5]。更为形式化的定义如下:

$$s = f_2(\{f_1(\{a_k\}, i), \dots, j\} \text{ iff } |ud(r, j)| = 1 \wedge \\ ud(r, j) = i \wedge j \in du(r, i) \wedge \forall a_k \cdot a_k - clear_{i \rightarrow j}$$

这个定义不对 i 中 r 的定义使用数目做限制。因此,如果 $ud(r, i)$ 中元素的数目为 n ,指令 i 可以被 n 个不同的指令 j_k 取代,条件是要满足 $r - clear_{i \rightarrow j_k}$ 特性。

经过上面两步, n -条件分支语句恢复到正规化表示如下:

jcond (var > numu) out

其中 var 为 switch 变量, numl 和 numu 分别为下界和上界, T 为跳转表的地址。Out 表示 switch 语句后的标号。根据不同的体系结构 (IA-64, Pentium, SPARC), 存在不同的地址表达式,大致可以将正规化形式分为以下三种:

- 1) $jmp\ m[< \exp\ r > * 4 + T]$
- 2) $jmp\ m[< \exp\ r > * 4 + T] + T$
- 3) $jmp\ m[(((< \exp\ r > \&mask) * 8) + T) + 4]$

形式 A (地址) 用于表示索引表,将标号作为表的值。形式 O 包含从表 T 开始位置到每一个分支的偏移。形式 H (哈希) 包含索引表的地址或偏移。它在跳转表的每个入口项中包含一个 ($< value >$, $< address >$) 对。

3.3 算法改进

针对 IA-64 编译器产生的机器代码的特性,我们对上面提到的算法进行了进一步的改进。

根据 IA-64 编译器的约定^[6],并分析产生的汇编代码,可以确定跳转表位于代码段,并且表中存放的为表首地址与分支地址之间的偏移,正规形式为 O 形;表的基地址固定为寄存器跳转指令的下一个指令束的地址。因此,在判定是否为跳转表时,通过在指令解码时读取寄存器的跳转指令之后的内容来判定,同时可以获得表的基址。这样就不必依赖固定的模式匹配,增强了程序的灵活性和实用性。

除了前面提到的切片停止条件,我们设定一个额外的停止判定:如果找到了索引跳转的下界,而且找到了其他的有关信息,则不执行更多的切片。同时还增加了对下界为 0 的特殊处理,这样使得判定算法的适用面更加广泛。

改进算法:

- 1) 解码遇到索引寄存器跳转,则在该处预读取指令,判

断是否为跳转表,如果是,则获得跳转表的地址,转 2) 执行;不是,返回。

2) 切片沿着索引跳转表达式中使用寄存器的传递闭包进行。切片结束的标志:通过另外一个函数返回;或者是未经定义就到达一个过程的开始;找到给下界赋值表达式。

3) 执行拷贝传播以恢复伪高级语句。

4) 确定跳转表的类型,同时在切片时获得索引变量以及跳转表的上界与下界。根据表首地址,计算跳转目标;

与以前的算法相比,进一步提高了效率,充分利用 IA-64 体系结构的特性,消除代码冗余以及模式匹配的不灵活性。

4 结语

在本文中,以 IA-64 的跳转表恢复为例,描述了基于切片和表达式传播的方法来实现索引跳转代码恢复的方法。并且在 UNIX 环境下,基于 IA-64 体系结构使用 SPEC95 整数基准程序的代码测试,结果超过 500 个 n -条件分支能正确的被检测到,大约 90% 的代码可以被恢复。

实验还发现,对于含有间接调用的程序代码恢复能力比较差,目前的分析技术还无法很好的处理,这类问题在当前的二进制翻译中,主要依靠解释程序在运行时动态的处理,也是下一步研究的重点。

参考文献:

- [1] SITES R, CHERNOFF A, KIRK M, et al. Binary translation[J]. Commun, ACM, 1993, 36(2): 69 - 81.
- [2] CIFUENTES C, SIMON D, FRABOULET A. Assembly to high-level language translation[A]. IEEE CS Press[C]. Washington DC, USA, 1998. 18 - 20.
- [3] HORWITA S, REPS T, BINKLEY D. Interprocedural slicing using dependence graphs[J]. ACM, 1990, 12(1): 26 - 60.
- [4] SALE A. The implementation of case statements in Pascal[J]. Software-Practice and Experience, 1981, 11: 929 - 942.
- [5] CIFUENTES C, FRABOULET A. Intraprocedural slicing of binary executables[A]. Proceedings of the International Conference on Software Maintenance[C]. Bari, Italy, October, 1997. 188 - 195.
- [6] Intel: IA-64 Software Convention and Runtime Architecture Guide; Intel Corp, July, 2000.