过程恢复技术在IA64二进制翻译中的应用与实现

付 文 魏 博 张天雷 赵荣彩
(解放军信息工程大学信息工程学院,郑州 450002)
E-mail:rachelfu@hotmail.com

摘 要 论文介绍了静态二进制翻译中的过程恢复技术。根据IA64体系结构特点及其对过程调用的有关约定,实现了对IA64过程的恢复;并在对大量实例进行研究的基础上,提出了一种浮点参数恢复的改进方法。

关键词 静态二进制翻译 IA64 过程调用 过程恢复

文章编号 1002-8331-(2006)21-0081-03 **文献标识码** A **中图分类号** TP314

Application and Realization of the Technology of Procedure Recovery in IA64 Binary Translation

Fu Wen Wei Bo Zhang Tianlei Zhao Rongcai
(Department of Computer Science and Technology, Information Engineering College of PLA, Zhengzhou 450002)

Abstract: This paper represents the technology of procedure recovery in static binary translation, realizes the recovery of IA64 procedures, based on the characteristic of IA64 architecture and the calling convention. Besides that, puts forward an improvement on the method for recovering floating-point parameters after the analysis of examples.

Keywords: static binary translation, IA64, calling convention, procedure recovery

1 引言

二进制代码翻译,是一种将可执行代码从一种体系结构指令集翻译到另一种体系结构指令集的技术。目前,在二进制翻译领域,已经形成了两种主要的翻译器体系结构,即静态翻译器和动态翻译器。国内外关于动态二进制翻译技术的研究文献比较多,但是对于静态二进制翻译中的有些技术实现问题讨论比较少,如静态翻译器中对过程的浮点参数的恢复技术[1]。另外,现有的静态二进制翻译器,大部分是围绕32位体系结构展开的,因此对过程恢复技术在64位体系结构上的二进制翻译中的应用进行研究也是很有意义的。

本文将全面介绍过程恢复技术在实现IA64/Linux到Alpha/Linux的静态二进制翻译器中的使用情况,包括参数的恢复和返回值的恢复,并针对现有资料中没有给出的浮点参数恢复问题,结合IA64体系结构特点提出了一种简单的解决思路。操作系统平台为Red Hat Linux AS 2.1 for IA-64+libelf-0.7.0,编译环境使用GCC 2.96。研究借鉴了Queensland大学开发的UQBT[2](University of Queensland Binary Translator)框架。

2 过程恢复技术简介

一个二进制翻译器在概念上可以分为三个阶段[1]。前端解码器:通过指令模式匹配对二进制代码进行处理,得到中间表示IR1;分析器:去除IR1的机器相关性,得到与机器无关的高级中间表示IR2;后端编码器:从高级中间表示IR2生成优化的目标机代码。对过程的恢复是分析器主要完成的工作,它实际上是将源机器的一组指令,通过抽象恢复出调用的高级结构。与直接将源机器指令模拟成目标机器上的一组指令相比,在高级结构上进行到目标机器上指令的转化,思路更加清晰和灵活,效率和可重利用率显然更高,因而是进一步转化成目标机器上代码的关键。

过程(函数)恢复主要完成以下三方面的工作:

(1)恢复出函数定义,例如:int add(int a,int b);

(2)恢复出函数引用,例如:i=add(j,9);

(3)通过分析得到函数的参数和返回值。

源函数在二进制代码中的表示与特定的操作系统和体系结构相关,因此对过程的恢复必须基于相应的ABI(Application Binary Interface)中对调用规则的约定。根据ABI中规定的被调用函数的前缀(prologue)和后缀(epilogue),可以很容易地找到函数体;根据调用函数的前缀,识别函数引用也比较容易。因此,过程恢复的重点和难点集中在对函数参数和返回值的恢复上。

3 IA64体系结构上过程恢复的实现

IA-64是Intel公司开发的,与以往任何体系结构完全不同的新一代64位微处理器体系结构。它是一种显式并行(EPIC)的体系结构,其设计思想不同于传统意义上的RISC指令集和CISC指令集。因此,对IA64可执行程序进行二进制翻译的研究具有创新意义。

在IA64/Linux到Alpha/Linux的静态二进制翻译器中,对过程的恢复主要分两部分实现:

(1)使用PAL[3](Procedural Abstraction Language)对IA64/

基金项目:河南省杰出人才创新基金(编号:0521000200)

作者简介:付文(1980-),女,硕士生,研究方向:软件逆向工程。魏博,硕士生。赵荣彩,教授,博士生导师。

Linux 的 ABI 中与机器相关的部分进行抽象。

(2)根据 PAL 描述,在中间表示 IR1 的基础上进行过程分析,恢复出参数和返回值,得到机器无关的中间表示 IR2。

以下将首先介绍什么是 PAL 语言,以及如何对 IA64 体系结构进行 PAL 描述,然后以对参数的分析为例给出基于 PAL 描述的过程分析,并针对参数不固定的过程间浮点参数的恢复给出一种新的解决思路。

3.1 IA64 的 PAL 描述

PAL 语言[3],是昆士兰大学于 1999 年在研发 UQBT 项目时开发的一种用于过程抽象的语言。PAL 的文法由一套 Backus-Naur 范式所定义,其语义可以用来描述处理器的体系结构中与过程调用相关的内容,如操作系统的调用约定,栈帧的组织方式,以及用于存放参数和返回值的有效地址等内容。

对 IA64 体系结构,有四个方面的内容需要使用 PAL 语言进行抽象:内存栈、局部变量的大小、参数以及返回值。IA64 处理器[4]提供了 128 个通用寄存器和 128 个浮点寄存器(分别命名为 GR0-127、FR0-127),并规定栈指针用寄存器 GR12 存放。因此只要通过简单的赋值语句将 r12(r12 即 GR12,后文中的 f8 即 FR8)指定为栈寄存器,就可以实现对内存栈的抽象。在进入一个过程的时候,IA64 会使用 alloc 指令为该过程分配栈帧,并指出其大小。因此在解码阶段从 alloc 指令中取出该值并用变量 locals 保存,就可以实现对局部变量的抽象。

PAL 描述中对参数的抽象,需要给出可能用于传参的有效地址,主要有寄存器地址和内存栈地址两类。对寄存器的描述需要给出可能用于传参的整型寄存器及浮点寄存型的列表;对内存栈的描述则需要给出用于存放参数的内存栈区域的信息,包括该区域的开始位置、增长方向以及每次增长大小的基数。在发生过程调用的时候,IA64 处理器[4]将为每个过程分配从 r32 开始的共 96 个寄存器作为一个寄存器栈,以供该过程使用,这是由 IA64 特有的寄存器栈机制及其寄存器重命名机制保证的。由于同一个参数对于调用过程和被调用过程而言,值相同但引用的方式不同,因此在 PAL 描述中对参数的抽象要从调用者和被调用者两个方面分别描述。这里给出的是从调用者角度来看的 PAL 描述:

```
INTEGER REGISTERS->%r32 %r33 %r34 %r35 %r36 %r37 %r38 %r39…%r127
DOUBLE  REGISTERS->%f8 %f9 %f10 %f11 %f12 %f13 %f14 %f15
```

另外,IA64 ABI 中还规定,参数在过程间传递时,前 8 个参数用通用寄存器或浮点寄存器传递,多于 8 个的部分用内存栈传递。因此在 PAL 描述中也需要给出对内存栈的描述。

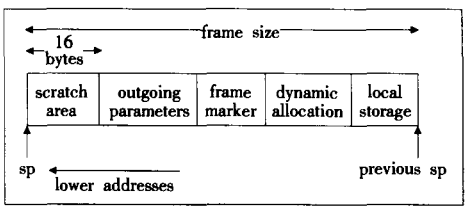

图 1 IA64 过程栈帧的结构

从图 1 中可以看到,每个过程的内存栈帧的前 16 个字节是空出来为内存的溢出做准备的,因此用内存传递的参数只可能从该过程栈帧的底部加 16 个字节的位置放起。同时,由于 IA64 的一个参数槽占 64 位,所以一个参数的位数必定是 8 个字节的正整数倍。因此在 PAL 描述中加上以下语句:

```
STACK          ->BASE=[%afp+16]
               OFFSET=8
```

以上语句按照先寄存器后内存栈的顺序排列,这也是与 IA64 的传参约定相对应的。

对返回值的抽象,需要指出返回值的类型与用于存放返回值的寄存器或者内存栈之间的一个映射关系。IA64 体系结构中对返回值的存放遵从以下约定:整型和地址类型的返回值存放在 r8 寄存器中;浮点类型的返回值放在浮点寄存器 f8 中。具体描述如下:

```
INTEGER.64   IN %r8
INTEGER.32   IN %r8
FLOAT.82     IN %f8
ADDRESS      IN %r8
```

3.2 基于 PAL 描述的过程分析

在对 IA64 体系结构进行了 PAL 描述后,接下来将在 PAL 描述的基础上,对调用结构进行抽象,从而达到去除机器相关性,得到高级中间表示的目的。

3.2.1 参数分析流程及产生的问题

现有的对 32 位体系结构(如 pentium)的参数分析主要由以下几步来完成:

(1)在调用点上对每个基本块的中间表示 IR1 中的位置(寄存器或内存单元)进行活跃变量分析(liveness analysis),得到每个过程的 LiveIn(在入口处活跃)和 LiveOut(在出口处活跃)变量,分别作为该过程可能的入参和出参。

(2)以 PAL 中给出的用于传参的寄存器列表和内存区域为依据,通过比较第一步中得到的 LiveIn 和 LiveOut 变量是否存在于该范围内,来判断其是否为参数,并将可能为参数的变量存入相应的结构中。

(3)过程间的分析。将调用者的出参与相应的被调用者的入参进行匹配,最终确定真正的参数。若判断出形参为可变参数列表类型,则将当前得到的所有出参都作为该过程的参数。

分析以上步骤可以发现,只有在同一个参数只用一个位置(一个寄存器或一个内存单元)进行传递的情况下,才能保证分析是正确的;一旦出现同一个参数用不同的位置(如同一个参数既用通用寄存器传递又用浮点寄存器传递)的情况,第三步分析中得到的参数将会出现冗余。

在 IA64 体系结构中,当函数参数个数不固定且参数中存在浮点参数的时候,就会出现以上情况,比较常见的情况如调用库函数 printf。因此以上方法不能适用于 64 位体系结构 IA64,需要对其进行改进。

下面将基于 IA64 传递浮点参数的特点,以 printf 为例,针对该问题给出一个简单的解决思路。

3.2.2 可变参数中浮点参数恢复问题的解决思路

IA64 体系结构中规定,如果一个浮点参数对应的形参是可变参数列表(variable-argument),则该参数必须由通用寄存器传递。因此,若在调用点上编译器无法决定该实参对应的形参是否为可变参数列表,则编译器将为其生成能满足这两种情况的代码,即对该浮点参数既通过通用寄存器传递又通过浮点寄存器传递[4]。

对带有 printf 浮点参数输出的 IA64 汇编例子进行总结,得到同一个浮点参数用寄存器传递的情况如下所示:

1	用浮点寄存器和通用寄存器传,且通用寄存器类型为整型	如f8<82f>和 r37<64i>
2	用浮点寄存器和通用寄存器传,且通用寄存器类型为浮点	如f8<82f>和 r37<64f>
3	只用通用寄存器传,且通用寄存器类型为整型	如 r37<64i>
4	只用通用寄存器传,且通用寄存器类型为浮点	如 r37<64f>

以上表格中的寄存器类型是由往寄存器中存值时所使用的指令决定的,如果使用的是浮点指令,则该寄存器的类型为浮点,如果使用的是整型指令,则该寄存器的类型为整型。

由于以上四种情况中都会用到通用寄存器传递参数,因此可以考虑在实参列表中去掉浮点寄存器而只保留通用寄存器,以解决参数冗余的问题。具体实现思路如下:

(1)在第一步中,将经过活跃变量分析得到的浮点寄存器和通用寄存器都作为该过程的活跃变量,存入相应的 LiveIn 和 LiveOut 变量列表中,无论是否出现同一浮点参数由多个寄存器传递的情况,也不用分析通用寄存器是何种类型;

(2)第二步保持不变;

(3)第三步中进行出参和入参的匹配时,若判断出入参为可变参数列表类型,则要先将出参列表中的浮点寄存器过滤掉,再作相应的匹配;若不为可变参数列表类型,则直接进行匹配。最终确定真正的参数。

需要说明的是,在参数个数确定的过程调用中,浮点参数仍然是用浮点寄存器进行传递的,因此为了保证以上方法的通用性,不能在第一步中就将浮点寄存器过滤掉。

经过以上处理,可以得到正确的实参列表。在接下来的后端处理中,再根据 printf 第一个参数(即输出字符串)中对需要输出数据的类型的定义,动态修正通用寄存器参数的类型,从而保证浮点参数在目标机上输出的正确性。

以上改进对传参情况的考虑是全面的。因为在对一个浮点参数进行的分析中需要考虑的因素有两个方面:是否用浮点寄存器传参,以及用于传递浮点参数的通用寄存器的类型被定义为浮点还是整型,所以总共需要考虑的情况只可能有 2×2=4 种,而无论哪种情况下该方法显然是可以正确地恢复出参数的。

另外,由于该算法只对形参为可变参数列表的函数做相应的过滤操作,所以对于参数个数确定的过程间浮点参数的恢复也不会产生影响。因此该算法能够适用于大多数的情况,并且经实践证明是切实可行的。

4 实例分析

以下给出一个简单的带有浮点输出的例子,来验证参数恢复的正确性。

源程序:

```
main() {
    float i;
    i=2.5;
    printf("i = %f\n",i);
}
```

可以看到,源程序中由 main 调用 printf 库函数来输出一个浮点数据 2.5。通过 IA64 上 objdump 命令得到的部分汇编码如下:

```
… 40000000000006c0:  ldfs f6=[r35]
   40000000000006c6:  nop.f 0x0
   40000000000006cc:  addl r36=88,r1;;
   40000000000006d0:  ld8 r36=[r36]
   40000000000006d6:  mov f8=f6
   40000000000006dc:  nop.b 0x0
   40000000000006e0:  getf.d r37=f6
   40000000000006e6:  mov r32=r1
   40000000000006ec:  br.call.sptk.many
                      b0=4000000000000460...
```

上述第一列给出的是代码装载地址,第二列给出的是相应汇编指令的操作码及操作数。第一行的汇编码中,浮点数据被存入临时浮点寄存器 f6 中,随后的第五行和第七行,该数据又被分别放入到了浮点寄存器 f8 和通用寄存器 r37 中。因此在 printf 的调用点上,这两个寄存器 f8 和 r37,以及存放字符串地址的寄存器 r36 都是活跃的。过程恢复之前的中间表示 IR1 如下:

```
40000000000006c0:0  *82* r[262] := fsize(32,82,m[r[35]])
40000000000006c0:1
40000000000006c0:2  *64* r[36]:=sgnex(22,64,88)+r[1]
40000000000006d0:0  *64* r[36]:=m[r[36]]
40000000000006d0:1  *82* r[264]:=r[262]
40000000000006d0:2
40000000000006e0:0  *64* r[tmpd]:=fsize(82,64,r[262])
                    *64* r[37]:=r[tmpd]
40000000000006e0:1  *64* r[32]:= sgnex(14,64,0)+r[1]
40000000000006e0    CALL 0x4000000000000460()
```

其中,第二列给出的是与汇编指令对应的 SSL[2]描述。在经过过程抽象及基于 PAL 描述的相关分析后,得到机器无关的高级中间表示 IR2 如下:

```
40000000000006c0:0   *82*r[262]:=fsize(32,82,m[%afp+16])
40000000000006c0:1
40000000000006c0:2   *64* r[36]:=r[1]+sgnex(22,64,88)
40000000000006d0:0   *64* r[36] := m[r[36]]
40000000000006d0:1   *82* r[264] := r[262]
40000000000006d0:2
40000000000006e0:0   *64* r[tmpd]:=fsize(82,64,r[262])*64*r
                     [37]:=r[tmpd]
40000000000006e0:1   *64* r[32] := r[1] + sgnex(14,64,0)
40000000000006e0     CALL printf(r[36]<64i>,r[37]<64f>)
```

由于 r35 是该过程的栈帧寄存器,因此在生成 IR2 的过程中会用机器无关的抽象帧指针 afp 将其替换掉;在地址 40000000000006e0 处,IR1 中对地址的调用在 IR2 中被恢复成对函数名 printf 的调用,并过滤掉了浮点寄存器 r264(即 f8。264 是该寄存器在 IA64 所有寄存器的列表中的编号,相应的 r262 即 f6),恢复出 printf 的参数 r36 和 r37。

5 实验结果

上述对 IA64 进行的过程恢复的方法已经应用于笔者参与研制的 IA64/Linux 到 ALPHA/Linux 静态二进制翻译系统中。目前,该系统能够成功地对过程进行恢复。特别的,在对带有参数个数不确定的函数调用的例子的翻译过程中,该系统能够在生成的高级中间表示 IR2,以及最终生成的低级 C 中正确的恢复出浮点参数和返回值,并且翻译后的代码在 ALPHA 机上经

(下转 89 页)

当然这三个步骤并不是完全顺序和独立的，而是相互影响，需要反复的交叉调试，获取对整个系统而言，在性能、面积、功耗上最优的解决方案。

现在 OptimoDE 开发的相关工具，如 OptimoDE Audio Edition、OptimoDE Standard Edition 和 AMBA Design Kit For OptimoDE 等，都可以从 ARM 公司获得授权[4]。

4 OptimoDE 的应用

OptimoDE 的架构满足现在应用形式不断产生和变化需求，它提供各种数字产品的基本功能，同时还直面当前许多设计挑战，如：很难把单个 DSP 设计得更快、更有效[5]。OptimoDE 能把硅片面积和能耗控制在最有效的范围内，并且由于实现上的简单性(处理引擎的 RTL 代码由工具自动生成，设计可在线进行等)，能大大加快产品的上市时间。此外，OptimoDE 架构可与 ARM 的开发工具配合使用，非常容易集成在基于 AMBA 的SOC 中[6]。至今 OptimoDE 已在多个不同的领域中得到成功的应用。

以 OptimoDE 在 MPEG-2 图像的解码算法为例，可以把算法分成二部分：一是 Z 扫描+IQ+IDCT；二是运动补偿。可用两个数据处理引擎来分别完成，相当于用两个硬件处理器。这两个处理引擎像流水线一样同时工作，而单个 DSP 方案必须在算法的各个部分来回切换[7]，这明显地提高了效率，也表明只要需要可以在一个系统中同时采用多个 OptimoDE 处理引擎，先根据功能划分算法，再进行多个处理引擎的协同设计。以上应用的模块如图 3 所示。

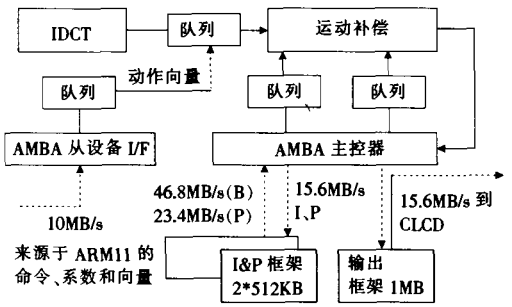

图 3 MPEG-2 图像解码算法的应用

我们也可以通过 OptimoDE 在复杂算法中的应用，看到它的有效性。表 1 例举了几种网络通信应用中，所需硬件电路门数和运算速率的数据。

表 1 OptimoDE 复杂算法应用数据

应用	速率	电路门数
IEEE802.11a	60MHz	1.75 万
H.264	90MHz	10 万
TurboDecoder(3G modem)	40MHz	4.5 万

ARM 的 AMBA Design Kit 能够把多个处理引擎集成在一个系统，所集成的 AMBA 总线接口主要有三类，控制接口：包含启动、停止、复位和中断；数据接口：输入处理引擎运行的代码及数据，输出运行结果，进行内外存储器的数据交换；调试接口：读出处理引擎的内部信息和内存储器的数据，用于程序的诊断。数据处理引擎的数据端口也可以不通过总线，直接访问外部存储器或 I/O，这使系统的设计具有极大的灵活性和高效率。

5 结束语

如前文所述，OptimoDE 有超强的可塑性，系统中的各种资源：算术、逻辑、移位、乘加、存储、I/O 和地址运算单元，都可以被配制，但数据处理引擎不是这些资源的简单拼凑，应用工程师根据产品的需求，量身定制最优性价比的处理引擎，可以是一款 DSP，也可以是 MCU。ARM 配套推出的 VLIW 指令压缩软件，有效地解决了 VLIW 长指令架构可能导致的对存储器空间的更多占有，是 OptimoDE 技术更具实用性。

OptimoDE 可配制的设计理念为处理器设计提供了充分的灵活性和有效性，利用 ARM 公司相应的自动化工具加快了设计进程，缩短了开发周期。采用 OptimoDE 技术是目前嵌入式产品设计的发展趋势。(收稿日期：2006 年 2 月)

参考文献

1.Nathan Clark.OptimoDE：Programmable Accelerator Engines Through Retargetable Customization.http://cccp.eecs.umich.edu,2004-08

2.Matthew Byatt.Delivering high-performance embedded signal processing with ARM OptimoDE™.http://www.arm.com/,2003-11

3.ARM Data Engines for Complex Multimedia Solutions.http://www.arm.com/,2005-08

4.OptimoDE Data Engine Family.http://www.arm.com/products/CPUs/families/OptimoDE.html,2005-08

5.Steve Furber.田泽等译.ARM SOC 体系结构[M].北京：北京航空航天大学出版社，2002：210~226

6.Andrew N Sloss.沈建华译.ARM 嵌入式系统开发-软件设计与优化[M].北京：北京航空航天大学出版社，2005：527~544

7.魏洪兴，周亦敏.嵌入式系统设计与实例开发实验教材 I[M].北京：清华大学出版社，2005：183~198

(上接 83 页)

过动态处理后编译、运行的结果与 IA64 上的运行结果完全相同，从而验证了上述改进的过程恢复方法的正确性。

6 结束语

本文介绍了过程恢复技术在 IA64 二进制翻译中的应用与实现，给出了 IA64 的 PAL 描述，并针对现有的对 32 位体系结构的二进制翻译中的参数分析方法无法正确恢复出 IA64 体系结构上的浮点参数的问题，给出了一种简单可行的解决方法。不足之处在于，该方法中用于存放浮点参数的通用寄存器的类型在过程恢复阶段无法正确得到，需要在后段中插入代码，并在运行时对寄存器类型进行动态修正。是否可以通过前端中的静态处理来保证生成的 IR2 中参数类型的正确性，从而增加生成的C 代码的可读性，提高代码的执行效率，将是下一步的研究内容。(收稿日期：2005 年 11 月)

参考文献

1.马湘宁等.二进制翻译中的过程恢复技术[J].计算机工程与应用，2002；38(19)：1~5

2.Cristina Cifuentes et al.The University of Queensland Binary Translator(UQBT) Framework.The University of Queensland and Sun Microsystems,Inc,2002

3.Cifuentes C,D Simon.Procedure Abstraction Recovery from Binary Code[C].In：Proceedings of the Conference on Software Maintenance and Reengineering,IEEE Computer Society Washington,DC,USA,2000

4.IA-64 Software Conventions and Runtime Architecture Guide.2000