计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
43(12):2166~2173, 2006

二进制翻译应用级异常处理

唐 锋[1,2] 武成岗[1] 张兆庆[1] 杨 浩[1,2]
[1](中国科学院计算技术研究所先进编译实验室 北京 100080)
[2](中国科学院研究生院 北京 100049)
(tf@ict.ac.cn)

Exception Handling in Application Level Binary Translation

Tang Feng[1,2], Wu Chenggang[1], Zhang Zhaoqing[1], and Yang Hao[1,2]
[1](*Advanced Compiler Technology Laboratory*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100080)
[2](*Graduate University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract Binary translation is applied for not only the legacy code porting but also software being used in different hardware platform. Exception handling is a most important aspect of binary translation research. How to balance the exception handling and the efficiency of binary translation is the key problem. Two methods to solve the active and passive exception handling in the library function jacket level are presented. One algorithm comes up to deal with efficienly the passive exception such as signal exception efficiently: if the exception handler doesn't use the machine state after exception happens, the translator doesn't preserve the precise machine state in every block. Thus, the cost for exception handling is low. For the Spec 2000 CPU ref input set, the cost, when the algorithm is applied to the translator, is an average 15.42 percent lower than the cost when the algorithm is not applied to the system. Another algorithm using emulated stack unwinding is submitted to find the calling address to help dealing with the active exception such as try catch exception in C++. The experiments prove the validity of the two ways of handling the exception. Adding the exception mechanics to the system doesn't decrease the performance of the common programs. These algorithms solve the exception handler in the application level program in binary translation quickly and correctly.

Key words binary translation; exception; library function; system call

摘 要 二进制翻译可以用于解决遗产代码的迁移问题,也可以实现不同硬件平台之间软件的通用.异常处理是二进制翻译的一个重要方面,如何解决异常处理和二进制翻译效率的矛盾是问题的关键.提出了在库函数包装层面处理主动异常和被动异常的方法,一个算法可以高效处理信号异常,另一个算法使用栈展开技术,得到调用地址用于处理 try catch 异常.实验结果表明,上述方法能够正确处理异常函数,同时对于普通应用程序加入异常处理机制之后性能并不受影响.

关键词 二进制翻译;异常;库函数;系统调用

中图法分类号 TP314

研究和开发新体系结构必须要有相应的软件支持,该体系结构才能得到流行使用.如果一个体系结构没能够赢得足够的市场份额,那么软件厂商就不会去冒险为这个体系结构开发新的应用软件.现

收稿日期:2005-05-10;修回日期:2006-03-28
基金项目:国家自然科学基金项目(60103006,60403017)

在PC市场基本上是被X86体系结构所占据,而服务器市场X86也已经占据了大部分的市场份额,在X86体系结构下有着丰富的应用软件.科技发展使得新体系结构必然会被开发推广,如何能使这些软件在新的体系结构上继续应用将是一件很有意义的事情,二进制翻译在这方面可以发挥很大的作用.

二进制翻译作为代码迁移的重要方法,得到了广泛重视.早在20世纪80年代开始出现第1个二进制翻译系统,至今已经取得了许多研究成果,并相继研制出实验性和商用的系统.1987年,HP公司就开发了最早的一个商用二进制翻译系统,用来将HP3000的客户转移到新的PA体系结构上.从1992年开始,DEC公司开发了一系列二进制翻译器,用来将VAX/VMS,MIPS/Unix,Sparc/Unix以及X86/WinNT上的代码翻译到他们新开发的Alpha机器上,这其中以FX!32[1]最有代表性.1996年IBM公司开发的Daisy[2],是利用二进制翻译调度PowerPC代码到超长指令字(VLIW)处理器,增加并行性.1999年IBM公司开发BOA[3]系统,动态翻译了PowerPC的整个系统,用简单的指令实现原来语义,简化硬件.2000年Transmeta公司发布了Crusoe处理器芯片和动态翻译代码软件Code morphing[4-5],用来在不同的硬件平台上运行X86代码,甚至包括操作系统Windows.2003年,Intel公司开发了IA32(Intel architecture 32)执行层软件IA32 EL[6],通过软件方法在IA64机器上执行IA32的应用程序.2004年Transitive推出了多平台应用程序级的翻译QuickTransit[7].

异常处理是二进制翻译中不可回避的问题,也是影响翻译性能的一个重要因素.异常处理,通常需要回退(roll back)机制[2-3,5],即在发生异常之后,能够把在产生异常之前的机器状态还原出来,机器状态主要指CPU的状态,包括通用寄存器以及一些控制、状态寄存器.二进制翻译分为系统级翻译和应用程序级翻译.系统级翻译器,模拟整个硬件平台,翻译运行在模拟硬件上的所有程序包括操作系统.与系统级翻译相对应,应用程序级的翻译是指翻译器不翻译操作系统,只翻译基于操作系统的应用程序,而且翻译器也使用操作系统提供的库函数和系统调用.

为了实现异常处理,必须加入额外的保存、恢复机器状态代码,牺牲运行效率.而二进制翻译若要具有较好的实用性,必须要求效率.所以如何使异常处理的花销尽可能减到最少,是值得研究的问题.

在绝大多数应用级异常处理程序不使用异常之前的机器状态的前提下,本文提出了一种实用、简化、高效的异常处理方法,用于处理应用程序中信号Signal异常,相比于其他系统所采用的检测点还原机制[6],效率有很大的提升.同时针对try catch异常,本文提出了根据运行时堆栈的内容,运用栈展开技术,得到每一层函数的调用帧信息及返回地址用于try catch中的异常处理的方法.

1　背景介绍

1.1　应用程序级二进制翻译中异常处理介绍

应用程序级翻译,按照对库函数是否翻译,有两种翻译层面:对库函数的包装和对系统调用的包装.如果翻译器是对库函数进行包装,那么在翻译中碰到库函数,就不对库函数本身进行翻译了,而是用相同功能的本地库函数进行替代.翻译器对系统调用的包装,则比库函数包装更加底层,即遇到库函数时,将进行库函数的翻译,只有碰到了系统调用,才用功能相同的系统调用进行模拟.

包装系统调用的翻译器,在这个层面可见的异常就是在执行指令时发生的硬件异常.产生异常之后,首先是由本地的操作系统捕获,然后由操作系统交给翻译器,再由翻译器决定如何进行异常处理.

而对于包装库函数的翻译器,因为库函数一般是基于系统调用实现的,库函数实现的功能也比系统调用复杂得多,所以处理库函数也会比处理系统调用更加复杂.对于异常处理的库函数,没有办法直接用目标机器上的库函数替代,需要特殊处理.在库函数包装这一层面上,异常处理总共有两类,一类是信号的被动调用,这一类处理方法同包装系统调用的翻译器方法类似;还有一类是C++中THROW的主动调用,需要对实现这种异常的库函数(Linux下通过gcc编译的就是*__throw*等函数)进行特殊处理.

1.2　相关工作

应用程序级翻译的整体效率高于系统级翻译,所以大多数商业实用的系统也是应用程序级的翻译器.

Intel公司的IA32 EL[6],Transitive公司的Quick Transit[7]都是对于系统调用的包装,所以它们对于异常的处理,只有当硬件发生异常,给操作系统发出Signal时才启动异常处理.

IA32 EL为了减小异常处理的开销,采用了还

原检测点机制,即在每隔一段间隔,插入一个机器的副本状态,一旦发生异常,就从最近的还原点把机器状态还原出来,从还原点开始重新执行. 在这种机制下,检测还原点的插入关系到翻译性能. 如果检测点间隔太短,那么加入的额外代码太多;间隔太长,发生异常之后的代价就会增大.

FX!32[1]是在 Windows NT 平台下从 X86 到 Alpha 的应用程序级翻译器,它是对 Windows 的 API 以及 com 组件进行包装,对异常的处理是支持 SHE(structured handling exception).

而学术界的产物,Queensland 大学的可变源目标的二进制翻译器 UQBT[8]没有对异常进行处理,任何产生异常的程序都会导致程序非正常结束.

2 应用程序级异常处理方法

对于大量的应用程序,应用程序级翻译的效率高于系统级翻译. 接下来介绍我们研究的应用级二进制翻译中的异常处理方法.

2.1 应用程序级二进制翻译中的异常

应用程序级异常可以分为被动异常和主动异常.被动异常,即在程序执行时,执行的某一条指令触发硬件的异常,通过信号传递给操作系统. 主动异常指用户自己抛出异常,即自己主动调用异常触发函数,运行时支持就会把用户抛出的异常进行一系列分析比较,例如异常类型、异常发生区段等,最终确定程序需要转向哪个异常处理程序入口,进行异常处理. try catch 异常处理机制就属于主动异常范畴,Java, C++[9]等面向对象的编程语言中都使用这类异常处理机制,使用静态的异常处理表,运行时查找该表,以搜寻异常处理句柄[10].

下面我们将给出这两种异常的处理方法.

2.2 信号 Signal 异常的处理

2.2.1 Signal 异常处理介绍

这是类 Unix 系统最普遍的异常,操作系统提供了一个库函数调用实现信号机制,该库函数 Signal 的函数原型为 void (* *signal*(int *signum*, void (* *sighandler*)(int)))(int),*signum* 为信号的值,*sighandler* 为异常处理函数指针,当硬件产生异常信号,如果这个信号已经被 *signal* 函数注册过,那么就将会执行 *sighandler* 函数.

本节简要介绍 Signal 异常的处理机制:当异常产生之后,通过中断门,CPU 的执行进入内核,首先保存机器状态,然后通过回调的方式调用 *sighandler* 函数,最后还原机器状态,回到异常发生的地址. 从中我们可以看出异常发生之后的机器状态,是操作系统所关心的. 因为机器状态的保存和还原都是操作系统所完成的工作,并不是应用程序所关心的,所以绝大多数情况下 *sighandler* 是不会去访问机器状态的. *sighandler* 异常处理函数一般使用高级语言书写,如果一旦异常处理函数需要使用机器状态,那么必然将会使用嵌入式汇编或者使用专门的库函数读取机器状态. 为了确保正确,处理极少数使用机器状态的异常处理函数,需要对 *sighandler* 进行分析,根据程序语义,只要 *sighandler* 没有调用读取机器状态的库函数,并且也未在对机器状态在定值之前就引用,那么我们就可以认为 *sighandler* 没有使用异常之后的机器状态,从而不必对没有使用机器状态的异常处理程序进行机器状态的保存.

由此可见,只要通过分析,得出 *sighandler* 并不使用机器状态,那么我们在翻译时,也就不需要在每个翻译单元之前保留机器状态. 通过对内部上千个程序测试表明,基本上应用程序不会使用异常发生之后的机器状态的. 为了处理少数使用异常之后机器状态的特例,在我们的算法中,需要加上分析 *sighandler*,如果一旦发现 *sighander* 中调用了读取机器状态的库函数,或者在对机器状态没有定值之前就已经引用,那么就必须在翻译系统中加入机器状态的保存,确保异常处理程序中得到准确的机器状态.

2.2.2 Signal 处理算法

在二进制翻译中,源平台上注册过的 *signal* 函数,到了目标平台上,对应的 *sighandler* 函数是不能直接执行的,而是要通过翻译执行. 因此需要维护一张 *sig_table* 表,每一个 *signum* 项对应源平台处理函数的地址. 当发生异常时,翻译器通过比较 *signum*,得到源平台处理函数地址.

算法分为以下 3 个模块:

1) 初始化

1.1 判断所有的异常处理程序 *sighandler* 是否会使用异常之后的机器状态,如果都不使用,那么在翻译执行时就不会加入机器状态的保存,否则就在每个程序块的开始加入保存代码.

1.2 在目标平台上用 *signal* 函数对所有信号量注册一个统一的异常处理函数 *sighhandler_target*. 为全局数据 *sig_table* 初始化,*sig_table* 每一项包含 3 个内容:信号值 *signum*,异常处理函数在源平台的入口地址 *sighandler*,异常处理函数在目标平台

相应的经翻译完成的本地码地址 *nativecodeaddr*.

该模块是为异常处理做了两项准备工作,程序开始时调用一次. 对于使用异常之后机器状态的应用程序,翻译效率会有所下降,但是对于其他的应用程序,加入这个判断之后,通知翻译程序在翻译时就不必加入机器状态的保存了,使效率和没有加入异常处理机制相比基本没有下降.

2) *signal* 函数包装

查找 *sig_table* 中是否存在对应的信号 *signum*,如果已经存在,那么就什么也不需要做;如果没有,则在 *sig_table* 中加入新的一项 *sigunm* 和 *sighandler*,如果异常处理函数 *sighandler* 还没有被翻译,不存在本地码地址,那么 *nativecodeaddr* 就填 NULL,否则就填写翻译的本地码地址.

只要源程序中调用 *signal* 函数,那么翻译执行也需要调用该模块. 按照源平台的语义,*signal* 函数是把 *signum* 和 *sighandler* 联系起来,但是在翻译系统中所有的 *signum* 都与异常处理模块相关联,由 *sighhandler_target* 判断是哪一个信号,对应哪一个源平台上异常处理程序.

3) 异常处理 *sighhandler_target*

判断产生的信号 *signum* 否是在 *sig_table* 中. 如果不在 *sig_table* 中,那么就说明该信号没有被注册过,调用操作系统默认的异常处理程序. 如果异常产生的信号在 *sig_table* 中,那么判断信号对应的处理函数是否已经存在本地码,如果存在,直接转到该地址执行. 否则,如果信号对应的处理函数不存在本地码,那么先翻译异常处理函数 *sighandler*,再执行翻译好的本地码.

本模块是在异常发生之后被调用的,因为 *signal* 函数把所有的信号都与这个模块关联起来了,异常发生之后都是首先进入这个模块,在这个模块中再决定是哪一个信号,需要翻译执行哪一个 *sighandler* 函数.

2.2.3 算法分析

处理系统级异常或者应用程序级的精确异常,需要恢复非常精确的机器状态,这在翻译执行中所占的开销将是非常巨大的. 为了能够恢复精确的机器状态,必须要有一个机器状态的副本随时保存着,那样发生异常之后才有可能从这个副本开始重新执行到发生异常的机器状态,维护这样的一个机器状态的副本,将会导致执行效率成倍的下降. 我们系统处理的对象是应用级程序,只要分析得到 *sighandler* 函数没有对异常时的机器状态进行访问,就不必恢复机器的精确状态. 恢复异常发生之前的机器状态,需要一系列的源目标平台寄存器对应分析以及上下文切换,这一切的开销都将使系统的性能大幅下滑. 上述方法避免了这部分开销,因此就应用级程序中的异常处理问题而言,既保证了正确性,又保证了程序的效率.

本算法在绝大多数情况下不需要恢复和保存精确的机器状态,这样使得翻译系统即使加入了异常处理机制,普通应用程序性能几乎没有下降. 而一旦在我们的每个翻译单元加入为了恢复机器状态而额外增加的翻译代码,那么执行效率将会下降,具体可参见我们在下文中给出的实验数据. 因此,我们的翻译系统比较其他翻译系统的异常处理在处理高级语言编译得到的应用程序时相较其他加入异常处理的翻译系统高效得多.

2.3 try catch 异常的处理

2.3.1 异常处理流程

C++ 中通过 throw 函数抛出异常,然后通过 catch 子句捕捉异常. 在源平台的可执行文件中是通过调用 *_throw* 库函数进行异常处理的.

对于源目标码中的库函数,当我们选择包装本地码库函数而不是翻译系统调用时,就会遇到这类异常. 包装本地码库函数的方法执行效率肯定会比通过翻译系统库函数高. 同时也会使翻译相对简单,不必考虑特权指令的翻译. 但是此做法的不利之处就是需要包装的库函数数量巨大,而且有些库函数需要特殊处理,例如异常抛出函数 *__throw*,异常重新抛出 *__rethrow* 等都是需要特殊处理的.

在翻译器中,通过下面的步骤进行 try catch 异常处理.

1) 得到发生 throw 操作的函数的被调用地址.

2) 判断 throw 是否发生在 try 区域中.

3) 如果 throw 确实发生在 try 区域中,那么翻译器就会把发生的异常类型和每一个 catch 子句所包括的异常类型进行比较.

4) 如果比较结果吻合,流程控制转到 catch 子句中,即翻译执行所对应的 catch 子句的源平台代码.

5) 如果 throw 没有发生在 try 区段中,或者没有一个 catch 子句所包括的异常类型和 throw 出来的异常类型相吻合,那么从源平台的堆栈中将函数展开,进入这个当前函数的调用者函数. 然后重复上述步骤 2)~5).

6) 如果所有的源平台堆栈都弹掉了,仍旧没有

找到合适的 catch 子句,就说明出现了未捕捉到的异常,调用终止函数终止.

2.3.2 关键技术——栈展开

上述流程中,关键的一点是需要定位函数调用地址,也就是说从调用 *_throw* 的地址开始,把每一层函数的调用函数被调用的地址比较是否在 try 区段中.

在二进制翻译的情形下,没有静态编译可以根据高级语言语义得到的函数调用关系表,但是幸运的是我们在动态情况下可以得到栈的运行状态.我们把根据被调用者(callee)的调用帧得到调用者(caller)的调用帧的技术称为栈展开.

下面以 X86 的函数调用栈帧为例说明如何栈展开,得到函数调用地址.

我们假设当前的函数是 f_1,它的 caller 是 f_2,那么 f_1 的栈帧范围就是图 1 中 *ebp* 和 *esp* 所表示的范围之内.

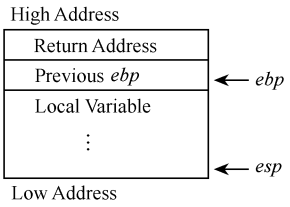

Fig.1 Function stack frame.

图 1 函数栈帧

当前 *ebp* 所指的内容就是 f_2 栈帧的帧顶,再往上的内容就是 f_2 调用 f_1 这条指令的下一条指令地址.我们得到这个返回地址之后就减去函数调用指令所占的字节得到调用地址.算法伪码表示如下:

1) 得到当前函数的栈帧信息,帧顶 *ebp*.

2) *esp* = *ebp*;*ebp* = *esp* 所指内容;/* 恢复 caller 帧顶,此时的 *ebp* 已经从 callee 退到 caller 了 */

3) *call_address* = (*esp* + 4 所指的内容) - 函数调用指令所占的字节数;/* x86 是 32 位字长,一个字有 4 个字节. *call_address* 记录 caller 调用 callee 的地址 */

4) 调整 *esp* = *esp* + 8;/* 把 *esp* 指向 caller 的帧低,我们已经完全从 callee 退到了 caller,现在 caller 是当前函数了 */

通过上述算法,我们把栈帧上移了一层,同时也提供了函数的调用地址.

C++ 中还支持异常的重新抛出,是通过 *_rethrow* 这个库函数进行处理的,对于 *_rethrow* 的处理,完全和 *_throw* 的处理类似,可采用上面的方法处理.

3 实验数据

3.1 Digital Bridge 系统简介

对于上述方法,我们在动态二进制翻译系统 Digital Bridge 上已经成功验证其可行性. Digital Bridge 系统是一个应用程序级的动态二进制翻译系统.主要功能是把 Linux 下 X86 的应用程序级的由高级语言编译下来的 elf 格式的二进制文件翻译成可在 mips 芯片的 Linux 环境中运行的代码. DB 系统的翻译单元为程序块,为一串线性代码,直到遇到跳转结束.二进制翻译系统只要保证目标平台和源平台的输出结果保持一致,就说明翻译执行是正确的,DB 系统已经能够正确执行上千个内部测试例子,也能够翻译较大型程序例如 Spec 2000[11] 中的大部分例子.

3.2 try catch 异常测试用例

源程序 test_ha.cc 如下所示,本测试用例是在汉诺塔程序基础上修改而来,该测试用例说明翻译器能够处理程序递归调用的异常.

```
#include<stdlib.h>
#include<stdio.h>
int num[4];
void mov(int n, int from, int to)
{
  int other;
  try {
  if(n==1) {
    num[from] = num[from] - 1;
    num[to] = num[to] + 1;
    printf("%d->%d\n",from,to);
    throw other;
  }else {
    other=6-from-to;
    mov(n-1, from, other);
    mov(1, from, to);
    mov(n-1, other, to);
  }
  }
  catch (…) {
    printf("exception throw\n");
  }
}
void mov_pre(int n, int from, int to) {
```

```
    int a;
  try {
    scanf("%d",&a);
    if (a==0)
     {
      throw a;
      }
    a=100/a;
    printf ("a=%d\n",a);
    }
  catch (int e)
   {
    a=0x1234;
    printf("Hello World!%d\n",a);
  }
    mov(n,from,to);
  }
  main() {
    int disk;
    disk=3;
    num[0]=0;
    num[1]=disk;
    num[2]=0;
    num[3]=0;
    mov_pre(disk,1,3);
}
```

通过g++用优化选项O3编译之后,其在X86服务器上运行结果如图2所示;在mips服务器上,通过DB运行的结果如图3所示.我们可以看到翻译器输出的结果完全一样,这就说明我们这样处理结果是正确的.同时我们也测试了重新抛出异常的问题,翻译执行之后的结果和X86本机执行的结果是完全一样的.这说明算法处理try catch异常是完全正确可行的.

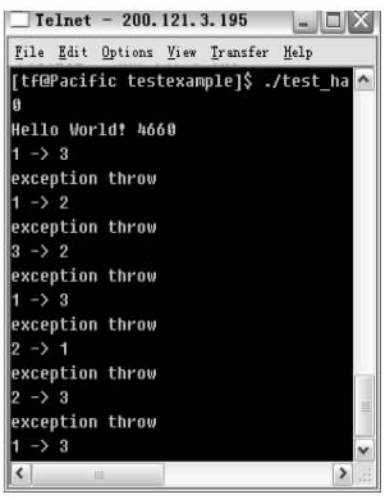

Fig. 2　Output on X86 server.

图2　X86服务器输出

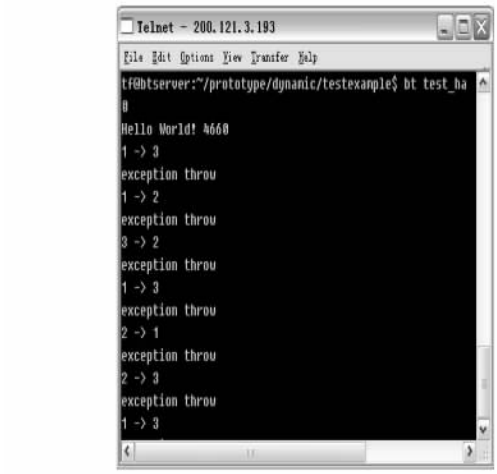

Fig. 3　Output on mips server.

图3　mips服务器输出

3.3　Spec2000 int测试用例

在本小节中,我们给出加入异常处理机制之后,Spec2000 int的10个测试用例ref测试集的运行速度比较.图4给出的数据是通过分析得出应用程序不使用异常之后的机器状态,因此没有保存机器状态的运行结果.图5给出的是没有进行分析,保守认为应用程序会使用到机器状态,因此在每个程序块的开始,都加入了保存X86机器状态的指令.

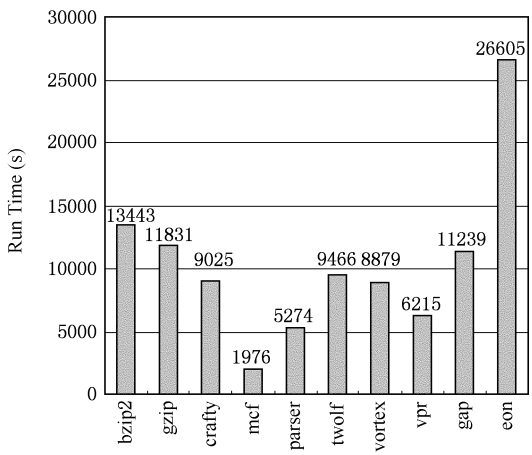

Fig. 4　No backup register.

图4　未保存寄存器

通过图4图5的比较,可以看出即使加入了对异常处理函数分析是否使用机器状态而增加了开

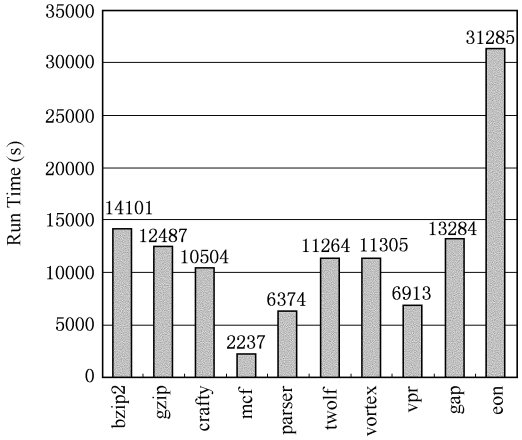

Fig. 5 Backup register.

图 5 保存寄存器

销,但是这部分开销相比于每个程序块都加入保存机器状态的开销就不值一提,最终还是未保存机器状态的运行速度明显要优于保存机器状态的.

从中我们可以得到 10 个测试例子平均下降了 15.42 个百分点,由此可见,为了保存寄存器现场是很费开销的. bzip2 和 gzip 下降的幅度最小是因为这两个程序的程序块最少,只有 2712 和 2746 个,而 vortex 下降幅度最大是因为其程序块个数最多,为 14133 个. 为了保存机器状态,需要在每个程序块的开始保存机器状态,所以增加的开销是和程序快的数目是成正比的,检测还原点的机制就是为了最少的插入还原点,但同时也要和发生异常之后的开销相平衡,所以一般间隔 10 个左右程序块会加入一个还原点,那么即使每个程序块只是保留 8 个通用寄存器,为每个程序快保留并不一定会使用到的机器状态,依然是一份不小的开销,需要的检测还原点越多,耗费的开销就越多;而我们的算法在检测到不会使用异常之后的机器状态之后,就不会再加入任何的检测还原点了,因此对于这类不使用异常之后机器状态的程序,本文算法效率要高于检测还原点机制.

4 结 论

异常处理是二进制翻译中一个十分重要的问题,本文提出了针对应用级程序二进制翻译的异常处理方法. 绝大部分的应用程序不会使用异常之后的机器状态或者也有可能不发生异常,所以我们在保证正确性的前提下,丢弃了对机器状态副本的维护,使效率得到了保障;同时正确翻译执行了异常的处理函数,以最小的代价,把信号异常处理机制加入到了翻译系统中. 为了在库函数包装这个层面上快速处理C++中的异常抛出以及重新抛出的问题,我们提出了动态栈展开的思想,从而得到本该静态编译阶段才可以得到的函数调用关系,分析得到 caller 调用 callee 的准确地址,用于异常区间的检查.

对于普通的应用程序,我们的算法还是非常鲁棒健壮的. 我们还需要继续研究在性能损失不大的前提下,维护堆栈中活跃的对象,在展开函数的同时,释放这些对象,解决存储碎片问题.

参 考 文 献

[1] Anton Chernoff, Mark Herdeg, Ray Hookway, *et al*. FX!32: A profile-directed binary translator [J]. IEEE Micro, 1998, 18(2): 56-64

[2] Kemal Ebcioglu, Erik Altman. Dynamic binary translation and optimization [J]. IEEE Trans on Computers, 2001, 50(6): 529-548

[3] Michael Gschwind, *et al*. Dynamic and transparent binary translation [J]. Computer, 2000, 33(3): 54-59

[4] R Halfhill. Transmeta breaks x86 low-power Barrier [J]. Microprocessor Report, 2000, 14(2): 1,9-18

[5] A Klaiber. The Technology Behind Crusoe Processors [M]. Santa Clara: Transmeta Corporation, 2000

[6] Leonid Baraz, Tevi Devor, Orna Etzion. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium® -based systems [C]. In: Proc of the 36th Annual IEEE Int'l Symp on Microarchitecture (MICRO-36 2003). Los Alamitos, CA: IEEE Computer Society Press, 2003. 191

[7] Transitive Corporation. http://www.transitive.com/, 2002

[8] C Cifuentes, M van Emmerik. UQBT: Adaptable binary translation at low cost [J]. Computer, 2000, 33(3): 60-66

[9] A Koenig, B Stroustrup. Exception handling for C++ [J]. Journal of Object Oriented Programming, 1990, 3(2): 16-33

[10] Ding Yuxin, Cheng He. Exception handling in Java virtual machine [J]. Journal of Computer Research and Development, 2000, 37(5): 622-626 (in Chinese)
(丁宇新, 程虎. Java 虚拟机异常处理机制的设计与实现[J]. 计算机研究与发展, 2000, 37(5): 622-626)

[11] Standard Performance Evaluation Corporation. http://www.spec.org/, 2002

Tang Feng, born in 1979. Ph. D. His main research interests include binary translation and compiler optimization.
唐锋,1979 年生,博士,主要研究方向为二进制翻译、编译优化.

Wu Chenggang, born in 1969. Ph. D. and associate professor, senior member of China Computer Federation. His main research interests include binary translation and compiler optimization.

武成岗,1969 **年生,博士,副研究员,中国计算机学会高级会员,主要研究方向为二进制翻译、编译优化**(wucg@ict.ac.cn).

Zhang Zhaoqing, born in 1938. Professor and Ph. D. supervisor, member of China Computer Federation. Her main research interests include advanced compiler technology and tools.

张兆庆,1938 **年生,研究员,博士生导师,中国计算机学会会员,主要研究方向为先进编译技术及相关工具环境**(zqzhang@ict.ac.cn).

Yang Hao, born in 1979. Master. His main research interests include binary translation.

杨浩,1979 **年生,硕士,主要研究方向为二进制翻译**(yanghao1102@ict.ac.cn).

Research Background

Binary translation is used to help the new(target) ISA machine execute the old(source) ISA binary code without the hardware support. The research on the binary translation and the related optimization has signality not only for legacy code migration but also for the program performance improvement and other aspect.

One of the challenges in binary translation is how to achieve run time performance on target architecture as good as or even better than native code. Exception handling is a most important aspect in binary translation. The exception handling mechanism in binary translation not only assures to execute the program correctly but also enhances the speed of program executing.

This research is supported by the Chinese Natural Science Foundation (60103006) and another Chinese Natural Science Foundation (60403017).