第 35 卷 第 2 期 计 算 机 工 程 2009 年 1 月
Vol.35 No.2 Computer Engineering January 2009
·开发研究与设计技术· 文章编号：1000—3428(2009)02—0283—03 文献标识码：A 中图分类号：N945

动态二进制翻译系统的调试器框架

郑举育，管海兵，梁阿磊
(上海交通大学软件学院，上海 200240)

摘 要：传统的动态二进制翻译系统缺少调试器支持或者调试功能有限，随着开发规模的扩大，调试手段成为制约设计开发进度的瓶颈。该文提出一种针对动态二进制翻译系统的调试器框架，引入观察点、回退执行与调试脚本 3 个功能，通过在Crossbit平台上的验证，证明该技术能够高效地帮助程序员发现错误，提高系统的开发进度。

关键词：动态二进制翻译；Crossbit虚拟机；调试器

Debugger Framework on Dynamic Binary Translation System

ZHENG Ju-yu, GUAN Hai-bing, LIANG A-lei
(School of Software, Shanghai Jiaotong University, Shanghai 200240)

【Abstract】Common dynamic binary translation system is lack of debugging tools or only provides primitive debugging supports, so debugging technology becomes the bottleneck of developing system software or large scale software. This paper introduces a new debugging technology on dynamic binary translation platform, including reverse executing, debugging script and other new concepts. The debugger implemented on Crossbit is proved that it reduces the time for developer to locate bugs sharply.

【Key words】dynamic binary translation; Crossbit; debugger

1 概述

动态二进制翻译是虚拟机实现中应用最广泛的一种方法，是解决遗留代码和提高软件平台适应性的有效手段，它在不需要可执行程序源代码的情况下，可以动态地将运行于其他平台的二进制程序进行转换，使其运行在目标机器平台上。

为二进制翻译平台提供调试器，一方面可以调试运行在其上的程序，有助于程序的开发。特别是当该平台用于模拟实际机器开发系统程序(如操作系统)时，调试器对开发具有更大的促进作用。现代操作系统一般都是在模拟器中开发的，其中一个主要原因就是模拟器提供了强大的调试支持，而这是实际机器无法提供的，比如Bochs模拟的x86 平台、Skyeye模拟的arm平台。另一方面也有利于发现动态二进制翻译平台本身的潜在错误。本文提出了一种适用于动态二进制翻译系统的调试技术，并在动态二进制翻译系统Crossbit[1]上得到了验证。

现有的动态二进制翻译系统提供了一定的调试支持，但功能有限。Tdb[2]为动态翻译程序提供了一个源代码级调试器，Qemu 实现了GDB 的基本调试协议，Dynamo[3]和DynamoRIO实现了一个底层调试支持，还有Java的JPDA[4]架构。但它们有一个共同的缺点：所提供的功能过于简单，只有设置断点、单步、执行和查看寄存器与内存等基本功能。而且Qemu和Dynamo的调试器仅适用于它们自身的平台，JPDA更是只为Java平台JIT编译器所设计的。因此，这些调试器的错误定位能力不足。程序中产生错误数据的地方一般在程序失败之前。这类调试器只能通过设置断点(从后往前查找错误之处)，多次重新执行程序来定位产生错误的地方。如果程序执行时间较长，这个过程会消耗大量的时间。而本文引入的观察点、回退执行和调试脚本将大大改善这种情况。

现在已经有一些具有回退执行功能的调试器，但它们都与语言紧密相关，利用特殊的编译支持达到逆向执行的目的。比如 PROVIDE 支持 C 语言程序的逆向执行，但它只支持 C 语言的一个子集；Cook 只支持 Java 字节码的回退执行；LVM 需要用它的专用 C 编译器在编译时插入相关信息。这些方法在二进制翻译中都不适用，而本文提出了一种二进制级别的非精确回退执行算法，能回退执行在二进制翻译平台中运行的任意二进制代码。调试脚本功能则是对传统调试器条件断点的加强。

2 动态二进制翻译系统

动态二进制翻译系统的执行过程如图 1 所示。

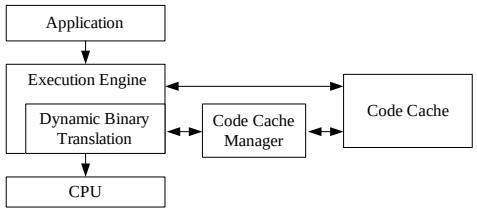

图 1 动态二进制翻译系统执行过程

应用程序的执行由执行引擎推动。在执行过程中，执行到的二进制程序由二进制翻译单元以代码块为单位翻译为本地代码，翻译好的本地代码被存储在代码 Cache 中，下一次被执行时就直接从代码 Cache 中获取。要运行的二进制代码

基金项目：国家自然科学基金资助项目(60773093)；国家“973”计划基金资助项目(2007CB316506)；国家“863”计划基金资助项目(2006AA01Z169)

作者简介：郑举育(1984－)，男，硕士研究生，主研方向：二进制翻译；管海兵，教授、博士生导师；梁阿磊，副教授

收稿日期：2008-04-30 **E-mail：**zhengjuyu@sjtu.edu.cn

所对应的本地代码是动态生成的，其地址也是运行时分配的。这使得现有的调试静态生成代码的调试器无法调试运行在动态二进制翻译平台上的程序。所以，有必要提供一种适用于动态翻译代码的调试架构。

3 调试器框架

调试器一般拥有断点、单步、查看寄存器与内存值等功能。本文的调试器中引入了回退执行、观察点与调试脚本的功能。图 2 是调试器基本框架。调试器引擎主要由 4 个部分组成：异常处理器，断点管理器，观察点管理器和基本块分析器。异常处理器用来捕获 CPU 处理器产生的异常，它会根据异常的种类调用断点管理器或观察点管理器进行相应的处理，然后将调试器的结果反馈到动态二进制系统的执行引擎中。基本块分析器会分析当前执行的基本块，并且为之保存基本块上下文，为逆向执行保存必要信息。

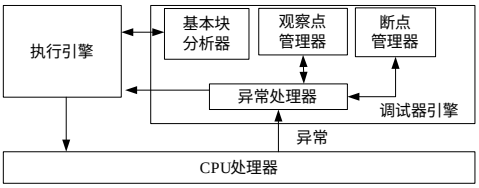

图 2 调试器框架

3.1 动态二进制系统中的断点

设置断点是调试器的基本功能之一。在传统调试器中，断点可以分为“硬件”断点和“软件”断点 2 大类。“硬件”断点的实现需要处理器的特殊支持。它的缺点非常明显，就是断点数目有限，与体系结构紧密结合。“软件”断点的实现一般是将程序指令替换为陷入指令、非法的除法指令或其他一些会产生异常的指令，执行到该地址后就会产生异常，再由调试器处理该异常。Linux 下的调试器 GDB 和 Windows 的调试器 OllyDbg 都采用这种方法。

动态二进制翻译系统的调试器一般采用软件断点方式，不过又与传统方法有所区别。可执行代码由翻译器生成，它们的地址是动态的。在程序的代码段中设置断点，翻译后的可执行代码中也必须有相应的断点。调试器需要一个映射表来支持原地址与实际动态地址的映射关系。在原程序中设置断点时，它对应的目标机器指令也应该替换为异常指令。当捕获到异常时，调试器通过映射表将实际地址转换为原地址，从而确定断点的位置。

3.2 单步和执行

单步运行时，调试器会创建一个只包含当前指令的基本块。然后交由执行引擎翻译并执行该基本块，执行结束后，将控制权还给调试器。如果选择继续运行，调试器只须将控制权交还给执行引擎，控制流即回到原来的状态，继续运行。

3.3 二进制级别的回退执行算法

本文提出了一种非精确回退执行的方法。非精确是指在某些特殊情况下，回退执行得到的数据不正确。但它只要满足大部分情况，就可以大大提高调试的效率。

回退执行有 2 个难点要解决：

(1)如何保存程序状态，包括寄存器值以及当前内存镜像。

(2)确定当前指令的前驱指令地址。因为当前指令不一定是顺序执行的结果，也可能是从某个跳转语句跳转而来的。

如果为每一条指令保存状态，则会消耗大量的时间与空间。其原因在于保存状态的粒度太小。本文提出了一种新的解决办法：扩大保存粒度，正向执行。扩大保存粒度就是以基本块为单位，只在基本块执行前保存程序状态。本方法的难点在于确定基本块内会修改的内存地址，特别是对于一些只能在执行时确定地址的内存。比如对局部变量的访问就是通过堆栈指针的简单运算而得到的地址。本文通过保存 BlockContext 结构(图 3)来解决。该结构由 4 个部分组成：(1)程序寄存器当前值。(2)当前栈顶的数据。在翻译基本块时，能很容易地确定该基本块访问的堆栈内存范围，这是因为对堆栈数据的访问都是基于堆栈指针与栈框指针的。(3)该基本块所用到的全局地址(以常数形式或全局寄存器加上常数地址形式使用)。(4)动态分配的内存地址。这类内存的地址是运行时产生的，决定了获取这类地址的复杂性。当检测到这种指令时，就用该指令作为起点构建新的基本块。前一个基本块执行结束时，可以计算出该指令所用到的内存地址，那么就可以将之保存在 BlockContext 中。这种方法有时会造成基本块太小，导致系统性能下降。为了提高系统性能，基本块分析器会在运行时对基本块进行分析，如果能通过调试器的运算得出该类地址，就可以不进行新基本块的构建，从而保持基本块的粗粒度。

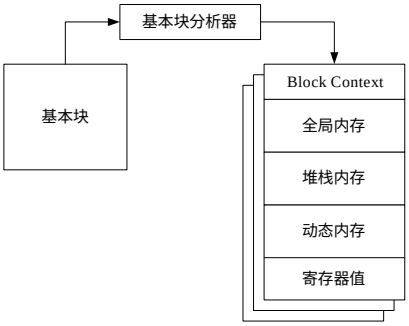

图 3 逆向执行中的基本块分析

为了确定前驱指令，调试器保存了最近被执行的基本块的 BlockContext 链表，并且按执行顺序存放，那么前驱指令就在最近的基本块中。当回退执行时，调试器先从该链表中取出最近被执行的基本块，根据它的 BlockContext 还原数据，然后用单步执行的方法执行到前驱指令(即回放的过程)。每执行一条指令就检查一下所修改的内存地址，如果该地址不在保存的内存行列中，就提示用户该地址的内存值可能不准确，由用户决定是否采用该结果。如果当前指令恰好是该基本块的第 1 条指令，就将该基本块从链表中删除，重新取出最近的基本块。由于基本块较小，因此单步执行所用的时间也较少。另外如果出现循环，可能造成实际可逆向执行的指令数大大减少。对此需做一定的优化：在 BlockContext 管理中，增加对循环的处理，如果连续保存的 2 个 BlockContext 属于同一个基本块，就将第 2 个舍弃，并在前一个 BlockContext 中记录被舍弃的数量。当回放时，该基本块必须被执行相应的次数。

由于系统调用与 I/O 操作的不可逆性，本文的方法无法恢复此类操作，且内存空间的有限性使得所能保存的 BlockContext 数目也很有限，因此不能无限制回退执行。被称为非精确回退执行的原因就在于此。但本文提供的回退执行功能主要是为调试服务的(这足以满足大部分程序的调试需求)，而不是为了完全逆向执行功能，这是逆向执行与调试目的的一个折中。后续的工作将对此做改进。本文的方法已在 Crossbit 中得到实现，并发挥了极大的作用。

3.4 观察点的支持

传统的调试器都有观察点这一功能，但由于动态二进制翻译系统的特性使得实现这一功能有一定的难度，因此现有的动态二进制翻译系统都不支持这一功能。而本文提出了3种在系统中实现观察点的方法：

(1)在有硬件支持的机器上(如IA32)采用硬件内存断点。设置观察点时，通过内存映射表，将源机器的内存地址转换为目标机器的内存地址，然后在该地址上设置内存断点。当监视的内存发生改变时就会产生异常，再由调试器捕获异常，进行处理。

(2)设置观察点后，调试器以单步的方式执行整个程序。当执行到某条指令改变了该内存值时，就暂停。

(3)利用内存映射过程。有些动态二进制翻译系统通过软件TLB的方式实现内存映射。使用一个表格记录已设置的观察点，软件TLB在翻译过程中查找该表，如果有匹配项，说明某个观察点被触发了，这时就将控制权转交给调试器。

第(1)种方法只能在特定机器上实现，但效率最高。第(2)种方法在所有二进制翻译系统中都能实现，但效率非常低，接近于解翻器的速度。第(3)种方法只能在使用了软件TLB的二进制翻译系统上实现。本文在Crossbit上分别实现了第(1)种和第(3)种方法。

3.5 调试脚本概念及其实现

本文的调试脚本功能是对传统调试器中条件断点的加强与改进。传统的条件断点功能有限，只能检查某些简单的条件，具体的分析工作必须由程序员手工完成。而调试脚本在此之上提供了一个更强的语义环境。调试脚本是指由程序员编写、帮助调试的程序片段。它对于调试器，就像shell脚本对于Linux一样。调试脚本中定义了一些关键词与内部变量。关键词主要有if, then, print, break等。内部变量有$r0-$r31(对于MIPS体系结构), $mem[], $regs[]。其中，基本运算符有+, -, *, /, =, ||, &&等。

调试脚本示例如下：

```
for( i = 0; i < 32; i ++ )
{if( $regs[i] == 0x12345678)
  break; }
if( $mem[0x10000] == 0x12345678)
break;
```

它表示如果有某一个寄存器值或者内存0x10000的内容变成0x12345678，就将程序暂停。当然还可以利用各种组合写出更复杂、更有用的调试脚本。比如根据PC寄存器统计执行次数，在某些条件下修改寄存器或者内存。

调试器读入调试脚本后，对脚本进行分析，生成相应的代码片段。调试器会在2个基本块执行间的空隙执行该代码片段，然后执行相应的动作。为了提高调试脚本的精度，调试器同时会对代码块代分进行细化。

4 实验效果对比

4.1 时间性能分析

图4给出了无调试器二进制翻译系统与有调试器版本的性能对比。由于调试器需要保存一些附加的状态信息，因此在性能上与无调试器的系统有一定的差别。增加回退执行与调试脚本功能之后，需保存的信息更多，还增加了程序分析的消耗，性能的损失不可避免，但仍与基本调试器处于同一个数量级上，说明本调试器是非常高效的。

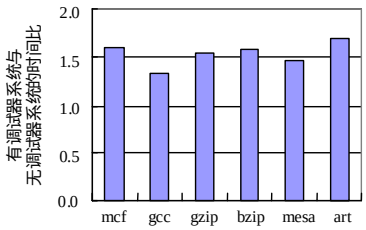

图4 有调试器系统与无调试器系统时间比

4.2 调试器在运行基准测试程序CPU2000中的应用

在动态二进制翻译系统Crossbit中实现了本文的调试器架构，并测试与定位到了一个与CFP2000 Mesa测试程序不一致的地方。Mesa中使用了浮点数取整功能，CFP的结果是以截去取整为标准，但是在不同的体系结构中，浮点数取整并没有一个统一的标准。比如Intel IA32就是使用一个特殊寄存器来设置取整模式，在不同模式下，得到的数据并不一样。而Crossbit模拟的PISA体系架构，实现为舍入取整。在测试Spec2000 mesa的过程中，所得数据与CFP提供的标准输出的所有数值都相差1。为了定位该错误，使用各种调试器的运行次数如下，这里的次数是指找到产生错误数据的指令(本例中为取整指令)所需的程序执行遍数：

调试器版本	原始调试器	使用回退执行功能	使用调试脚本
次数	> 20	1 (回退约1 000指令)	2

调试的第1步是找到存储错误数据的内存地址。然后以此往前分析调试程序，找出产生错误数据的指令。使用基本调试器需要设置多次断点并执行多次，才能最终找到该指令。使用回退执行功能只需直接在该地址回退执行，很容易就能找到出错指令。而使用调试脚本只需在执行第2遍时，以错误数据为判断条件，就可以在错误数据产生的第一时间暂停程序。

由此可见，引入的观察点、回退执行、调试脚本大大提高了程序员定位错误的速度。

5 结束语

动态二进制翻译与优化技术推动了计算机系统的发展。它作为某些处理器的虚拟机，解决了软件的移植问题。本文设计的调试器不但可以作为手工验证动态二进制翻译正确与否的工具，而且能解决在其之上开发或者移植系统软件所存在的问题。与现有的动态二进制翻译器的调试器相比，本调试器大大提高了程序员定位错误的速度，为虚拟机开发人员的排错定错提供了极大的便利。本调试器已经在Crossbit中实现，并在测试CPU2000的过程中起到了巨大的作用。

参考文献

[1] Bao Yuncheng. Building Process Virtual Machine via Dynamic Binary Translation[D]. Shanghai: Shanghai Jiaotong University, 2006.

[2] Kumar N, Childers B R, Soffa M L. TDB: A Source-level Debugger for Dynamically Translated Programs[C]//Proc. of AADEBUG'05. Monterey, California, USA: ACM Press, 2005.

[3] Bala V, Duesterwald E, Banerjia S. Dynamo: A Transparent Dynamic Optimization System[C]//Proc. of Conference on Programming Language Design and Implementation. [S. l.]: ACM Press, 2000.

[4] Cook J J. Reverse Execution of Java Bytecode[J]. Computer Journal, 2002, 45(6): 608-619.