一个高可移植性的轻量级 x86 模拟器①

曹欢寅，张 妍

(复旦大学 软件学院，上海 200433)

摘 要: 介绍一个可以在多种处理器体系结构上运行的轻量级 x86 模拟器 PIT(Portable x86 Instruction Translator)。“动态二进制指令翻译”是一个可以让一种机器的指令运行在另一种机器上的技术。PIT 采用了可移植的动态二进制指令翻译技术，可以在多种体系结构（包括 x86,PowerPC,ARM, Sparc）的 CPU 上模拟执行英特尔 x86 指令。用户可以在 PIT 环境中加载 x86 系统上在用户态运行的常用格式的 16 位或 32 位可执行文件（包括 COM，EXE，ELF）运行并观察输出结果。PIT 采用了指令块动态缓存和条件码延迟计算的技术用于加速指令翻译的效率，使用虚拟 x86 MMU(内存管理单元)的方式支持指令在 PIT 虚拟环境中对 4GB 内存空间进行寻址。只需要通过简单的配置和重新编译，PIT 就可以在不同的 CPU 体系结构上模拟运行 x86 系统上的可执行文件。

关键词： x86 模拟器；动态二进制指令翻译；虚拟化技术

Highly Portable Light-Weight x86 Emulator

CAO Huan-Yin, ZHANG Yan

(Software School, Fudan University, Shanghai 200433, China)

Abstract: This paper presents a light-weight x86 emulator PIT (Portable x86 Instruction Translator) which is portable on multiple CPU architecture. “Dynamic binary translation” is a technique that can make instructions of one CPU architecture be capable of running on another CPU architecture. PIT utilizes portable dynamic binary translation technique to emulate Intel x86 instructions on multiple CPU architecture such as x86, PowerPC, ARM and Sparc. Users can load 16-bit or 32-bit user-level executable files of x86 system such as COM, EXE and ELF into PIT, execute them and observe the output. PIT utilizes Transferred Block Dynamic Cache and Condition Code Lazy Computation techniques to accelerate instruction translation. With virtual x86 MMU (Memory Management Unit), emulated instructions can address 4GB memory space in PIT virtual environment. With simple configuration and re-compiling, PIT can be easily ported and run x86 executable files on different CPU architectures.

Keywords: x86 emulator; dynamic binary translation; virtualization

1 引言

英特尔 x86 体系结构是目前最为被广泛使用的一种 CPU 架构，大量的现存软件都是为 x86 架构而编写的。为了让 x86 的程序可以在别的 CPU 架构上运行，x86 模拟器实现了在异种 CPU 架构的硬件环境中模拟出 x86 架构硬件环境的技术，使得 x86 的程序可以在不同 CPU 架构上的虚拟环境中被执行。Bochs[2]和 QEMU[3]是两个被广泛使用的 x86 模拟器。

本文要介绍一个可以在多种处理器体系结构上运行的轻量级 x86 模拟器 PIT(Portable x86 Instruction Translator)。它采用了可移植的动态二进制指令翻译技术，与其他 x86 模拟器相比，只需要通过少量的配置就可以让 PIT 的同一份代码在不同的 CPU 体系结构上运行（包括 x86, PowerPC, ARM, Sparc）。与 Bochs 和 QEMU 对 x86 环境的完全模拟不同，PIT 实现的是轻量级的 x86 模拟，使得 x86 可执行文件在 PIT 中可以

① 收稿时间:2010-09-03;收到修改稿时间:2010-11-13

直接高效地运行。PIT 采用了指令块动态缓存和条件码延迟计算的技术用于加速指令翻译的效率。PIT 在 Linux 环境中运行，通过虚拟的 x86 MMU 可以让被模拟执行的 x86 程序寻址 4GB 的虚拟内存空间。

2 系统设计

PIT 总体上可以分为输入输出控制、程序加载器、指令翻译器、x86 硬件环境模拟器这四部分。结构图如图 1。

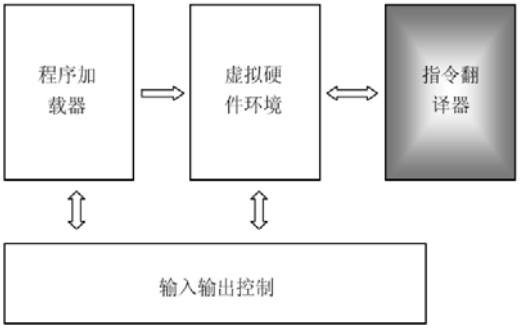

图 1　PIT 的功能模块图

输入输出控制模块可以通过命令行的方式同用户进行交互。用户可以通过它来让 PIT 加载指定的 x86 可执行文件运行并看到输出结果，可以用它来配置虚拟 CPU 的运行模式（实模式、保护模式、分页是否开启等），可以用它来配置模拟的物理内存大小，还可以进行 x86 程序的调试工作（包括下断点，单步执行，打印内存和寄存器等）。程序加载器用于加载常用格式的 16 位或 32 位可执行文件（可以是 COM，EXE 或者 ELF 格式）到模拟的物理内存中，并且设置虚拟 CPU 的 EIP 寄存器到程序入口地址，启动虚拟 CPU 进行程序模拟执行。虚拟硬件环境模块用于模拟虚拟 CPU，记录 CPU 状态，模拟内存和 MMU 等功能，也和输入输出模块进行交互。指令翻译器用于将 x86 指令转换为本地指令进行执行，并且在执行的过程中更新模拟的内存和 CPU 的状态。

PIT 的指令翻译分为三个阶段：取指令、转换为中间代码、翻译执行中间代码。第一阶段中，指令翻译寄存器会通过虚拟 CPU 的 EIP 寄存器找到下一条指令的内存地址，将指令逐个字节地读出进行分析。分析的过程中生成和平台不相关的中间代码（第二阶段）。通常一条 x86 指令会被转换成若干条中间代码，转换完成后进入第三阶段，针对每条中间代码用宿主 CPU 的指令进行翻译执行。

3 实现

3.1 可移植的指令翻译器

PIT 的指令翻译器通过取指令、转换为中间代码和翻译执行中间代码这三个阶段对 x86 指令进行翻译执行。通常一条 x86 指令会被转换成若干条中间代码，这样做有两个好处。首先，把 x86 指令转换成中间代码可以简化要翻译的指令的复杂度。x86 的 CISC 指令集是一套十分复杂的指令集，它们不定长，某些指令功能复杂而且寻址方式多样。如果直接把 x86 指令翻译成宿主 CPU 的本地指令，需要处理大量的指令操作码(opcode)和操作数的不同组合。而将 x86 指令首先分解成若干条简单的中间指令，可以大大的简化要实际翻译的指令集。把 x86 指令转换成中间指令的第二个好处是便于以后对指令执行块进行优化。

举一个 x86 指令翻译成中间代码的例子。假设 PIT 要执行 x86 指令“mov eax, [ebx]”，意为要将 ebx 寄存器所指的内存中的 32 位数据放到 eax 寄存器中。PIT 会将这条指令分解成如下的中间指令：

```
MOVE T0, EBX
LOAD T1, T0
MOVE EAX, T1
```

这三条指令的意思为：将虚拟 CPU 的 EBX 寄存器中的内容放到 T0 寄存器中；将 T0 寄存器所指的内存中数据加载到 T1 寄存器中；将 T1 寄存器的内容放到 EAX 寄存器中。T0 和 T1 分别为 PIT 维护在内存中的临时寄存器，任何 x86 内存或者寄存器之间的数据搬移都要通过这两个临时寄存器来实现，从而简化了指令操作数之间的组合数。虚拟硬件环境中寄存器与寄存器之间，寄存器与内存之间的数据搬移通过 MOVE 和 LOAD、STORE 这两组指令进行区分从而方便指令翻译器的实现和新 x86 指令的扩展。

PIT 翻译中间代码是用指令解释(instruction interpret)的方式的。这种指令翻译法广泛的被 Java 虚拟机（Hotspot[4], Dalvik[5]）和脚本语言解释器所使用。对于每一条中间指令，PIT 都有一个用 C 语言写的解释函数，在虚拟化环境的语境中实现该中间指令的功

能。比如中间指令 LOAD T1, T0 的解释函数如下：

```
void op_load_t1_t0()
{
    vcpu->t1 = get_data32_from_vram(vcpu->t0);
}
```

vcpu 是用于记录虚拟 CPU 当前状态的数据结构，临时寄存器 T0 和 T1 是它的数据成员。

get_data32_from_vram()函数用于返回模拟的内存中指定地址的 32 位数据，如果 CPU 开启分页则返回的是 4GB 虚拟内存空间中的数据，如果 CPU 不开启分页则直接返回模拟的物理内存中的数据。

基于这样一种指令解释的翻译方式，PIT 的可移植性得以实现。由于每条 x86 指令都会被翻译成中间代码，而中间代码又由 C 函数进行解释，这就省去了开发者通过人工编码的方式将被模拟的 x86 指令翻译成对应的宿主机指令这部分工作，将每条指令的转换工作交给了宿主机的 C 编译器。并且，PIT 中的虚拟 x86 CPU 的寄存器完全存储在内存中，所以也不涉及被模拟的 x86 寄存器和宿主 CPU 寄存器之间的映射问题。故对于不同的宿主 CPU 的体系结构，只需要重新编译一下 PIT 就可以生成可用的指令翻译器。而对于那些直接将要模拟的目标指令翻译成本地机器指令，在本地 CPU 直接执行的模拟器(如 QEMU)，则需要针对不同的宿主平台写出不同的指令翻译逻辑，这对于支持新的宿主平台大大地提高了门槛。

3.2 已转换指令块(TB)缓存

PIT 的指令翻译器每次不止将一条 x86 指令转换成中间代码。它会逐条地将 x86 指令翻译成中间代码，直到遇到一条跳转指令或者令 CPU 进入不可预知状态的指令(比如串操作指令,在翻译阶段不可预知 CPU 将会继续进行串操作还是执行下一条指令)。这些中间代码存在一个连续的内存区域中，称为“已转换指令块”(Transferred Block，以下简称 TB)。每次 PIT 都会将一整个 TB 进行中间代码的解释执行工作。PIT 的 TB 构造流程如图 2。

每次转换完成的 TB 都会存在一个连续的内存缓存区域中(TB Cache)，该区域大小可配置（默认为 20MB）。每次 PIT 要翻译一段 x86 代码时，都会根据 EIP 的值查找一个哈希表，该哈希表会告诉 PIT 这段代码是否已经存在于 TB Cache 中，如果已经存在则可以直接对这段 TB 进行解释执行，如果不存在则仍然需要进行 TB 构造的工作。

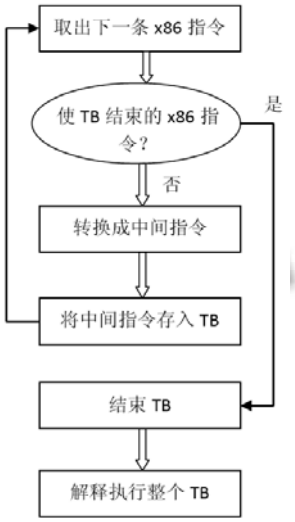

图 2 PIT 的 TB 构造流程

3.3 EFLAGS 条件码延迟计算

许多 x86 指令都会更新 EFLAGS 寄存器的条件标志位(ZF，SF，CF，OF，PF)，这些标志位用于在执行条件跳转指令时作为是否需要跳转的依据。计算每条指令对条件标志位的影响需要花费大量的开销，因此为了加速指令翻译速度，PIT 并不会在每执行完一条 x86 指令后都对 EFLAGS 的条件标志位进行更新。每当一条影响条件标志位的 x86 指令被执行后，PIT 都会在一组特殊的辅助寄存器中记录该条指令的操作码和操作数。当遇到一条 x86 指令（如条件跳转指令)需要对 EFLAGS 的条件标志位进行检查时，PIT 会通过这组特殊的寄存器找到最新的那条影响标志位的 x86 指令的操作码和操作数，计算出它对条件标志位的影响，然后更新 EFLGAS 寄存器。

3.4 内存管理

PIT 运行时需要模拟的物理内存大小是可以配置的。如果用户选择让虚拟 CPU 在保护模式下开启分页运行，则程序可以在 PIT 环境中对 4GB 内存空间进行寻址。PIT 模拟了 x86 的 MMU（内存管理单元），使得在虚拟 CPU 开启分页的情况下，程序对 4GB 内存空间的访问会像真实的 x86 环境中那样查询页表，对模拟的物理内存进行页替换。被换出的页会被存放到一个创建在 Linux 文件系统下的缓存文件中，以待下

次再被换入到模拟的物理内存中。

由于 x86 是小端机，故 PIT 模拟的物理内存中的数据都是以小端顺序存放的。如果宿主机是大端机的话，则每次被模拟的程序访问物理内存的时候 PIT 都会对数据的大小端进行转化。用户可以通过修改 Makefile 的配置参数，编译出适合在小端机和适合在大端机上运行的不同版本的 PIT。

3.5 自修改代码的支持

自修改代码是指程序中含有一部分用于修改程序代码段的代码。由于 PIT 用到了已转换指令块缓存的技术，所以当已经被缓存起来的 TB 所代表的那部分 x86 代码被修改时，这部分 TB 就必须被作废。

自修改代码分为两种情况，第一种是修改自身 TB 块对应的 x86 代码，第二种是修改其他 TB 块对应的 x86 代码。对于第二种情况，由于 PIT 中执行的 x86 程序所有对内存的访问都会通过 PIT 映射到模拟的物理内存上，故 PIT 如果发现程序试图修改一个内存地址的数据，而这个内存地址恰巧是属于某段已经被缓存的 x86 代码的，则 PIT 会作废那段 x86 代码所对应的 TB。下次这段被修改的 x86 代码要被执行时，PIT 会重新对其进行中间代码转换，从而不影响自修改代码的正确性。对于前面提到的第一种情况，也就是程序试图修改自身 TB 块对应的 x86 代码，则 PIT 会选择永远不对这段代码进行 TB 缓存，每次运行这段 x86 代码时都要对这段代码进行重新的中间代码转换。

4 性能

我们选择了 QEMU 和 Bochs 作为 PIT 性能的比较对象，QEMU 版本 0.12.1，Bochs 版本 2.4.2。比较的方法是评测这 3 个模拟器在执行用户态计算任务时的表现。由于 QEMU 和 Bochs 是对英特尔 x86 环境进行完全模拟的模拟器，故需要在它们创建的虚拟机中安装一个操作系统才能执行用户态程序。我们选择安装 debian 5.0，它是一个轻量级的 Linux 内核操作系统，可以尽量减少运行操作系统带给 QEMU 和 Bochs 的资源开销。我们的宿主机器选择的是英特尔酷睿 2-4300 处理器，带 2GB 物理内存，运行的是 Ubunbu 9.10 版本操作系统。QEMU，Bochs 和 PIT 在各自运行时都被分配了 256MB 的内存用来模拟虚拟机内的物理内存，PIT 运行时 CPU 使用保护模式并且开启了分页。

在 3 个模拟器中，我们运行的用户态程序包括了普通的 32 位整型数据的算术与逻辑运算，计算斐波那契数列，计算两个自然数的最大公约数，数组的快速排序和二分查找，堆排序以及用搜索树解决图论中节点覆盖问题（Vertex Cover）等一些列计算任务。最后我们得到了图 3 中的实验数据。

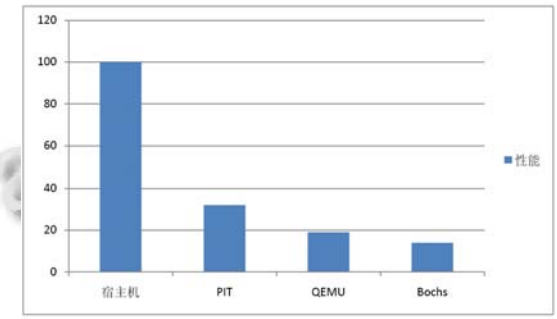

图 3　PIT 和其他模拟器之间的性能比较

我们将直接在宿主机上运行上述计算任务的性能得分标准化为 100 分，则 PIT 运行该计算任务的得分为 32 分，QEMU 和 Bochs 的得分分别为 19 分和 14 分。在模拟器中运行计算任务无疑要比直接在宿主机上运行慢很多。在各个模拟器之间的比较中，由于 PIT 实现的是轻量级的 x86 模拟，可以直接运行 x86 的用户态可执行程序，而 QEMU 和 Bochs 实现的是 x86 完全模拟，要运行计算任务必须首先要在模拟器中运行一个操作系统，故 PIT 在运行这个计算任务的得分上要胜过 QEMU 68%，比 Bochs 快一倍以上。

因此在不需要运行一整个 x86 的操作系统，而只是需要跑一个 x86 上的用户态程序时，PIT 的轻量级模拟比起 QEMU 和 Bochs 具有更好的计算性能。

5 结论

PIT 是一个轻量级的易于移植的 x86 模拟器。它可以高效地直接执行 x86 系统上用户态运行的程序。只需要通过简单的配置和重新编译，PIT 就可以在多种处理器架构上运行。比起一般的完全模拟 x86 环境的模拟器（比如 QEMU 和 Bochs），PIT 以其轻量级的模拟方式，在模拟执行一般的 x86 用户态程序的时候表现出更好的性能。

（下转第 143 页）

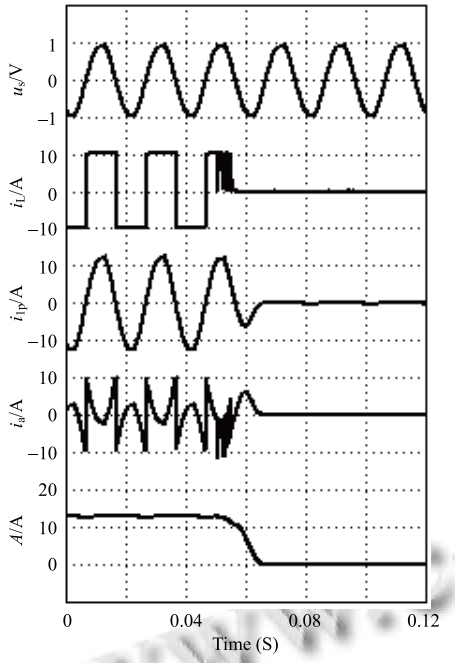

图 4 负载电流从 10 A 突变为 0 A 时的实验结果

4 结论

使用 UA206 A/D 数据采集卡通过 PCI 口与计算机相连，以电源电压和负载电流作为输入信号可以构成一种简单实用的 APF 谐波电流检测实验系统。该系统克服了传统的谐波电流检测实验系统存在的问题，它具有结构简单、稳定性好、可靠性高、程序设计比较容易等特点。该实验系统不仅适用于 APF 单相电路的谐波电流检测，而且适用于 APF 三相电路的谐波电流检测（具有 16 或者 32 单端模拟输入通道）。由于 UA206A/D 卡具有丰富的输入输出接口，该系统还可用于信号的分析与处理、智能实时控制等。

参考文献

1 Akagi H, Kanazawa Y, Nabae A. Instantaneous reactive power compensators comprising switching devices without energy storage components. IEEE Trans. Ind. Appl., 1984,20(3): 625–630.

2 Wang Q, Wu N, Wang ZA. A neuron adaptive detecting approach of harmonic current for APF and its realization of analog circuit. IEEE Trans. Instrum. Meas, 2001,50(1):77–84.

3 王群,周雒维,吴宁.一种基于神经网络的自适应谐波电流检测法.重庆大学学报:自然科学版, 1997,20(5):6–11.

4 王群,吴宁,苏向丰.有源电力滤波器谐波电流检测的一种新方法.电工技术学报, 1997,12(1):1–5.

5 周雒维,江泽佳,吴宁.基于补偿电流最小原理的谐波与无功电流检测方法.电工技术学报, 1998,13(3):33–36.

6 周雒维,李自成,吴宁.有源电力滤波器谐波及无功电流的一种检测方法.重庆大学学报:自然科学版, 2000,23(1):53–55.

7 于志豪,刘志珍,徐文尚.基于电路模型和神经网络的谐波电流检测方法.电工技术学报, 2004,19(9):86–89.

8 王群,谢品芳,吴宁,苏向丰.模拟电路实现的神经元自适应谐波电流检测方法.中国电机工程学报, 1999,19(6):42–46.

（上接第 104 页）

参考文献

1 李剑慧,马湘宁,朱传琪.动态二进制翻译与优化技术研究.计算机研究与发展,2007,44(1):161–168.

2 Mihocka D, Shwartsman S. Virtualization without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. ISCA'08: Proc. of the 35th International Symposium on Computer Architecture. Beijing, China, 2008:55–70.

3 Bellard F. QEMU, a Fast and Portable Dynamic Translator. TEC'05: Proc. of the USENIX Annual Technical Conference 2005. Berkeley, CA, USA: USENIX Association, 2005:41–46.

4 Java SE Hotspot at a Glance. Oracle. [2010-11-08]. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

5 Dalvik Virtual Machine. Google. [2010-11-08]. http://www.dalvikvm.com/