QEMU内核探秘

2012-03-13 15:10 by debugger87, 2542 阅读, 2 评论, 收藏, 编辑

本文译自Fabrice Bellard大神的文章《QEMU, a Fast and Portable Dynamic Translator》，如有翻译不当之处，请斧正。

摘要

在本文中，我们将展示QEMU的内部机制。QEMU是一个快速的机器模拟器，它使用了独创的可移植动态翻译器。QEMU可以在若干种宿主机（x86，PowerPC，ARM以及Sparc）上模拟若干种CPU(x86，PowerPC，ARM以及Sparc)。它支持完全系统模拟和Linux用户模式模拟。对于完全系统模拟，完整且不经修改的操作系统可以运行在虚拟机之上；而对于Linux用户模式模拟，其意味着一个为某种目标CPU编译的Linux进程可以运行在另一种CPU之上。

1 绪论

QEMU是一种机器模拟器：它能使未经修改的目标操作系统（比如Windows或者Linux）及其所有的应用程序运行在虚拟机之上。QEMU自身运行在多种宿主操作系统之上，比如Linux，Windows以及Mac OS X。此外，宿主机和目标机的CPU可以不同。

QEMU的主要用途为在某个操作系统之上运行另一个操作系统，比如在Linux上运行Windows，或者在Windows上运行Linux。由于虚拟机易于关闭并且其状态可以被探测、保存与恢复，所以QEMU的另一个用途便是调试。除此之外，通过添加新的机器描述和模拟设备，可以模拟出一些特殊的嵌入式设备。

QEMU还集成了一个Linux特殊用户模式模拟器。它使为某个目标CPU编译的Linux进程可以运行在其他CPU之上。有了这样一个模拟器，我们不必启动一个完整的虚拟机就可以测试交叉编译结果是否正确或者测试CPU模拟器是否功能正确。

QEMU由若干子系统组成：

1. CPU模拟器（目前支持x86，PowerPC，ARM以及Sparc）
2. 模拟设备（比如VGA显示器，16450串口，PS/2鼠标和键盘，IDE硬盘，NE2000网卡，等等）
3. 通用设备（比如块设备，字符设备，网络设备），这些设备用来连接模拟设备与宿主机中相应的设备
4. 用于实例化模拟设备的机器描述（比如PC，PowerMac，Sun4m）
5. 调试器
6. 用户接口

本文阐述了QEMU中所使用的动态翻译器是如何实现的。动态翻译器在运行时将目标CPU指令转换为宿主机CPU指令。将翻译后得到的二进制代码保存到翻译缓存中，使之能够被重用。与解释器相比，动态翻译器的优势在于，提取和解码目标指令操作只需一次。

通常，由于需要重写整个代码生成器，动态翻译器很难从某一个宿主机移植

到另一个宿主机。这意味着其工作量相当于添加一个新的目标机器指令集到C编译器中。然而，QEMU就简便得多，它仅仅是把一系列由GCC离线生成的机器码片段链接起来。

一个CPU模拟器还需要面临其他更经典但是更困难的问题：

1. 翻译代码缓存管理
2. 寄存器分配
3. 直接块链接
4. 内存管理
5. 支持代码自修改
6. 支持异常处理
7. 硬件中断
8. 用户模式模拟

2 可移植动态翻译器

2.1 Description

第一步，将每一条目标CPU指令拆分成更小更简单的指令---微操作。每一个微操作由一小段C代码实现。这一小段C代码被GCC编译成一个目标文件。我们选择微操作的原因在于它们的数目比目标CPU中所有指令与操作的组合数要小得多（通常为几百）。将目标CPU指令翻译为微操作的工作完全通过手动编码实现。为了追求可读性和简洁性，可以对源代码进行优化，因为这个阶段对速度的要求并没有解释器那么严格。

dyngen是一个编译时工具，它将包含微操作的目标文件作为输入，用以生成一个动态代码生成器。动态代码生成器在运行时被调用，生成一个完整的链接了若干微操作的宿主机函数。

这个工具的做法同[1]类似，但为了获得更好地性能，在编译时又做了更多工作。特别地，在QEMU中有一个关键思想，常量参数可以被传递给微操作。为了达到这个目的，针对每个常量参数都使用GCC对伪代码进行重定位。这使得dyngen可以对伪代码进行重定位，并在创建动态代码的时候生成适当的C代码来解决这些问题。重定位还支持对静态数据和其他微操作中函数的引用。

2.2 Example

这里有一个例子，我们需要把下面的PowerPC指令转换为x86代码：

```
addl   r1, r1, -16          # r1 = r1-16
```

PowerPC代码翻译器将会产生如下微操作：

```
movl_T0_r1           # T0 = r1

addl_T0_im  -16      # T0 = T0 - 16

movl_r1_T0            # r1 = T0
```

在微操作的数目减到最小的同时，并没有对生成代码的质量产生负面影响。例如，我们只是生成从一些临时寄存器之间的move操作，而不是32个PowerPC之间所有可能得move操作。通过使用GCC静态寄存器变量扩展，这些T0，T1，T2临时寄存器通常被指定为宿主机寄存器。

微操作movl _T0_ r1通常由如下代码实现：

```
void op_movl_T0_r1(void)

{

T0 = env->regs[1];

}
```

env是一个包含了目标CPU状态的结构体。32个PowerPC寄存器被保存在数组env->regs[32]中。

addl_T0_im 更有意思，因为它使用了一个常量参数。该常量参数的数值在运行时确定。

```
extern int __op_param1;

void op_addl_T0_im(void)

{

    T0 = T0 + ((long)(&__op_param1));

}
```

dyngen生成的代码生成器提取由opc_ptr指定的微操作流，输出gen_code_ptr位置的宿主机代码。微操作参数由opparameter_ptr指定：

```
[...]

for(;;) {

switch(*opc_ptr++) {

[...]

case INDEX_op_movl_T0_r1:

{

extern void op_movl_T0_r1();

memcpy(gen_code_ptr,

(char *)&op_movl_T0_r1+0,

3);

gen_code_ptr += 3;

break;

}

case INDEX_op_addl_T0_im:
```

```
{

long param1;

extern void op_addl_T0_im();

memcpy(gen_code_ptr,

(char *)&op_addl_T0_im+0,

6);

param1 = *opparam_ptr++;

*(uint32_t *)(gen_code_ptr + 2) =

param1;

gen_code_ptr += 6;

break;

}

[...]

}

}

[...]

}
```

对大多数微操作比如movl_T0_r1而言，GCC生成的宿主机代码仅仅需要拷贝即可。当需要使用常量参数时，dyngen实际上是这样处理的：GCC对__op_param1进行重定位，使用运行时参数对需要生成的代码打补丁。

当代码生成器在运行的时候，将会输出如下宿主机代码：

```
# movl_T0_r1

# ebx = env->regs[1]

mov 0x4(%ebp),%ebx

# addl_T0_im -16

# ebx = ebx - 16

add $0xfffffff0,%ebx

# movl_r1_T0

# env->regs[1] = ebx

mov %ebx,0x4(%ebp)
```

在x86机器上，T0将会被映射为ebx寄存器，CPU状态上下文将会被映射到ebp寄存器。

2.3 Dyngen implementation

QEMU翻译的关键是dyngen。在使用dyngen处理包含有微操作的目标文件时，需要完成如下任务：

1. 目标文件（包含微操作）将被解析以获取它的符号表，重定位入口点，及其代码段。这个过程依赖于宿主机目标文件格式（dyngen支持ELF（Linux），PE-COFF（Windows），MACH-O（Mac OS X））。
2. 微操作位于代码段中，代码段使用符号表。有一个特殊的宿主机方法用于获取拷贝代码的起始点与结束点。通常，函数prologue与epilogue会被忽略。
3. 检查每个微操作的重定位，获取常量参数的个数。通过使用特殊符号__op_paramN来检测常量参数重定位。
4. C代码中的内存拷贝函数用来拷贝微操作。对每个微操作的代码进行重定位，以此对拷贝的代码进行打补丁。如此便可使之被恰当地重定位。此外，重定位操作由宿主机定义。
5. 对一些宿主机例如ARM而言，常量必须存放在生成代码附近，因为需要通过相对位置访问他们。宿主机通过一个特殊程序对生成代码中的常量进行重定位。

当编译微操作时，使用一系列GCC标志来操作函数prologue和epilogue代码的生成，使之生成的形式易于解析。一个伪汇编宏强制GCC对每个微操作相关的函数编译产生以唯一指令。如果若干输出指令由单一微操作生成，代码将链接失败。

3 实现细节

3.1 Translated Blocks and Translation Cache

当QEMU第一次取得一段目标机器码时，它将其翻译为宿主机代码。然后链接后续的跳转代码或指令。通过该指令，可以通过一种翻译时无法推导出的方式修改静态CPU状态。我们把这种基本模块叫做Translated Blocks（TBs）。

一个16MB的cache保存最近最常使用的TBs。简单起见，当该缓存用满时，它将被清空。

静态CPU状态就是在编译过程中进入TB时已经知道的那部分CPU状态。例如，编译时，所有目标机器上的PC(程序计数器)是已知的。在x86机器上，静态CPU状态包括更多可以用来产生更优代码的数据。例如，很重要的是，知晓CPU是处于保护模式、实模式、用户模式还是核心模式，同时知道默认指令大小为16位还是32位。

3.2 Register allocation

QEMU使用固定的寄存器分配机制。这意味着每个目标CPU寄存器将被映射到宿主机固定的寄存器或内存地址。在大多数宿主机上，我们只是简单地将目标机器寄存器映射到宿主机内存，并且只是保存一些临时变量到宿主机寄存器中。临时变量的分配情况在目标机描述文件中指定。这种方法的好处在于简洁且可移植性好。

在QEMU未来的版本中，将会使用一个动态的临时寄存器分配器，把目标寄存器直接保存在宿主机寄存器中，以避免一些不必要的move操作。

3.3 Condition code optimizations

要想获得良好的性能，则必须有良好的CPU条件码模拟（eflags register on x86）。QEMU使用惰性条件码估值：它只保存某个指令中的源操作数（CC_SRC

），目标操作数（CC_DST）和操作类型（CC_OP），而不是在每条x86指令执行之后计算条件码。对于32位的加法计算，例如R=A+B，我们可以得到如下表达式：

CC_SRC=A

CC_DST=R

CC_OP=CC_OP_ADDL

由于可以通过存放在CC_OP中的常数知道有一个32位的加法，我们可以通过CC_SRC和CC_DST恢复A，B和R。然后，如果下一条指令需要，所有相关的条件码，例如零结果（ZF），非负结果（SF），进位（CF）或者溢出（OF）都可以很容易获得。

由于一个完整TB代码会在一段时间内生成，条件码估算可以在翻译时得到进一步优化。在生成的代码上，有一个看似保守的过程，用于检测CC_OP，CC_SRC或者CC_DST是否未被

后续代码使用。在TB结尾，我们认为这些变量已被使用了。然后，我们删除那些后续代码不再使用的变量。

3.4 Direct block chaning

在每个TB执行之后，QEMU通过一个hash表使用模拟PC以及静态CPU状态中的其他信息，来查找下一个TB。如果下一个TB还未翻译，那么将启动一个新的翻译进程。否则，跳转到下一个TB的操作到此为止。

为了在一些很普通场景（新的模拟PC值已知，比如条件跳转指令）下进行加速，QEMU可以对TB打补丁程序，以致其能直接跳转到下一个TB。

移植性最好的代码使用非直接跳转。在某些宿主机（例如x86或者PowerPC）上，将会直接附加一个子程序调用指令，以免block链产生开销。

3.5 Memory management

对于系统模拟，QEMU使用mmap()这个系统调用来模拟目标机器MMU（内存管理单元）。只要模拟操作系统没有使用宿主机操作系统占用的内存区域，这个模拟MMU都会正常工作。

为了能够启动任何操作系统，QEMU还支持一个软件MMU。在这种模式下，每一次内存访问都要进行MMU虚拟地址到物理地址的转换。QEMU通过使用地址转换缓存来加速转换。

为了避免每次MMU映射改变时都清空地址转换缓存，QEMU使用了一个物理索引转换缓存。这意味着每个TB用它的物理地址进行索引。

当MMU映射改变时，因为跳转目标的物理地址可能会改变，索引TB链被重置（例如，将不能从一个TB直接跳转到另一个TB）。

3.6 Self-modifying code and translated code invalidation

在大多数CPU上，自修改代码很容易处理。通过执行一条特殊的代码缓存废弃指令，可以发出信号指示出该代码已被修改。这足以废弃相应的翻译代码。

然而，在一些CPU例如x86上，当代码被修改时，应用程序不能发出信号以废弃指令缓存。所以，自修改代码是一个特殊的挑战。

当生成了一个TB的翻译代码时，如果相应的宿主机页不是只读的，那么它将会被设置为写保护。如果有一个针对该页的写访问产生，QEMU会废弃该页中所有的翻译代码，并使该页重置为可写。

通过维护一个包含给定页中所有TB的链表，可以有效完成正翻译代码的废弃任务。除此之外，还有其他链表用来取消直接block链。

当使用软件MMU时，代码废弃将更加高效：如果某个代码页由于写访问而频繁做废弃代码操作，将会创建一个展示该页内部代码的bitmap。每次往该页的

存储操作都将检查bitmap，以知晓该页的代码是否需要废弃。这避免了该页仅作数据修改时就进行代码废弃操作。

3.7 Exception support

当发生异常例如除零操作时，使用longjmp()跳转到异常处理代码。没有使用软件MMU时，宿主机信号处理器被用来捕获无效内存访问。

QEMU支持精确异常，这意味着在异常发生时可以获取目标CPU的精确状态。目标CPU的大多数状态都会被翻译代码显式存储与修改。对于那些未被显式存储的目标CPU状态S（例如当前程序计数器），它们将通过重翻译TB获取。在这个TB中发生了异常，并且状态S在每条目标机器指令翻译之前已被记录。发生了异常的宿主机程序计数器被用于查找相关目标机器指令以及状态S。

3.8 Hardware interrupts

为了运行速度更快，QEMU并不在每个TB中检测硬件中断是否处于未处理状态。相反，用户必须异步调用一个特殊的函数以获知某个中断未处理。该函数重置正在执行的TB链。这确保正在执行的TB链可以从CPU模拟器的主循环中立即返回。然后主循环会测试是否有某个中断未处理，并对处理该中断。

3.9 User mode emulation

为了能够让针对某种CPU编译的Linux进程可以运行在另一种CPU上，QEMU还支持用户模式模拟。

在CPU级别，用户模式模拟仅仅是完全系统模拟的一个子集。因为QEMU假定用户内存映射是由宿主机操作系统处理的，所以用户模式模拟并没有MMU模拟。QEMU包含了一个用于处理字节序问题和32/64bit转换的通用Linux系统调用转换器。因为QEMU支持异常处理，所以它显然也模拟了目标机器的信号机制。每个目标机器的线程以宿主机线程的形式运行。

4 移植工作

为了将QEMU移植到新的CPU宿主机上，需要完成以下事情：

1. 必须移植dyngen
2. 为了优化性能，微操作所使用的临时变量可以被映射到宿主机中某些特定的寄存器中。
3. 为了维持指令缓存与内存的一致性，大多数宿主机CPU需要特殊指令的支持。
4. 如果直接block链由分支指令实现，需要提供某些特殊的汇编宏。

QEMU整个移植工作的复杂度估计和动态连接器相当。

5 性能

为了测试因模拟而带来的系统开销，我们比较了BYTEmark benchmark在x86 Linux系统宿主机本地模式下的性能与x86目标机器用户模式模拟下的性能。

经测试，就整型代码（定点运算）而言，QEMU用户模式比本地代码模式慢四倍；就浮点代码（浮点运算）而言，用户模式则要慢十倍。可以理解的是，这个结果是由x86静态CPU状态缺少FPU栈指针而导致的。

对于完全系统模拟，QEMU比Bochs[4]几乎快了30倍。

而用户模式的QEMU则比valgrind –skin=none version 1.9.6[6]快了1.2倍。后者是一个手动编码的x86 to x86的动态翻译器，通常用于调试程序。--skin none选项确保Valgrind不会产生调试代码。

6 结论及未来的工作

QEMU已经可以用于日常工作，特别是商业x86操作系统（如Windows）的模拟。使用PowerPC作为目标机器，在其上几乎已经可以启动Mac OS X。此外，在Sparc目标机器上，已经可以启动Linux。目前还没有那个动态翻译器可以在如此多的宿主机上支持如此多的目标机器，这主要还是因为其他动态翻译器的移植复杂度难以估量。而QEMU似乎在性能与复杂度之间做到了很好的平衡。

未来仍需要处理如下问题：

移植：QEMU已经很好地支持PowerPC和x86宿主机。其他宿主机（Sparc，Alpha，ARM，MIPS等）上的QEMU还需要进一步的工作。此外，QEM对用于编译微操作C代码的GCC为哪个版本有很大依赖。

完整系统模拟：ARM和MIPS目标机还未添加支持。（目前已经支持，我之前使用QEMU模拟过ARMv5处理器）

性能：软件MMU的性能还有提升空间。一些紧要的微操作使用汇编语言手动编码，而不是在当前的翻译框架中做大量修改。同时，CPU主循环也可以用汇编语言手动编码。

虚拟化：当宿主机和目标机器CPU相同时，可以在目标机上运行大多数代码。最简单的实现方法是，像通常一样模拟目标机器内核代码，但在目标机器上运行用户代码。

调试：可以添加缓存模拟和循环计数器，得到类似SIMICS[3]中的调试器。

7 Availability

目前可以通过http://bellard.org/qemu获取可用的QEMU。

参考文献

[1] Ian Piumarta, Fabio Riccardi, Optimizing direct threaded code by selective inlining, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[2] Mark Probst, Fast Machine-Adaptable Dynamic binary Translation, Workshop on Binary Translation 2001.

[3] Peter S. Magnusson et al., SimICS/sun4m: A Virtual Workstation, Usenix Annual Technical Conference, June 15-18, 1998.

[4] Kevin Lawton et al., the Bochs IA-32 Emulator Project, http://bochs.sourceforge.net.

[5] The Free Software Foundation, the GNU Compiler Collection, http://gcc.gnu.org.

[6] Julian Seward et al., Valgrind, an open-source memory debugger for x86-GNU/Linux, http://valgrind.kde.org/.

[7] The BYTEmark benchmark program, BYTE Magazine, Linux version available at http://www.tux.org/~mayer/linux/bmark.html.

« 上一篇：Xen工作原理
» 下一篇：找亲戚

标签: QEMU

#1楼 T库科技
2012-03-13 15:17
代码写的不错，关注一下
支持(0) 反对(0)

#2楼[楼主] debugger87
2012-03-13 15:22
@T库科技
代码来自大神的原文。。。
支持(0) 反对(0)