Shawn的专栏

个人资料

yearn520

KVM虚拟机代码揭秘——QEMU代码结构分析

虚拟机 cache 编译器 汇编 代码分析 struct

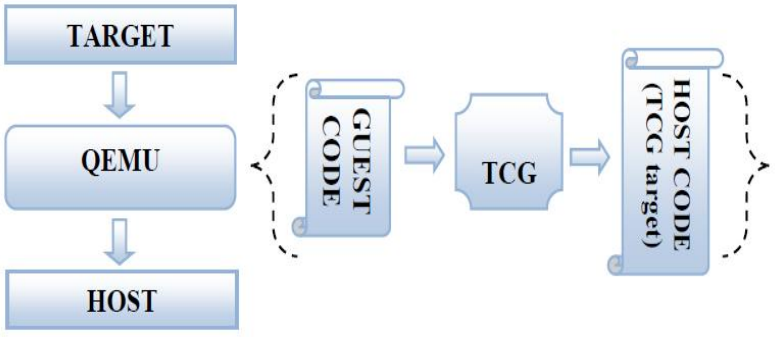
目录(?) [+]

前言：本文主要概括了QEMU的代码结构，特别从代码翻译的角度分析了QEMU是如何将客户机代码翻译成TCG代码和主机代码并且最终执行的过程。并且在最后描述了QEMU和KVM之间联系的纽带。

申明：本文前面部分从qemu detailed study第七章翻译而来。

1.代码结构

如我们所知，QEMU是一个模拟器，它能够动态模拟特定架构的CPU指令，如X86，PPC，ARM等等。QEMU模拟的架构叫目标架构，运行QEMU的系统架构叫主机架构，QEMU中有一个模块叫做微型代码生成器（TCG），它用来将目标代码翻译成主机代码。如下图所示。



我们也可以将运行在虚拟cpu上的代码叫做客户机代码，QEMU的主要功能就是不断提取客户机代码并且转化成主机指定架构的代码。整个翻译任务分为两个部分：第一个部分是将做目标代码（TB）转化成TCG中间代码，然后再将中间代码转化成主机代码。

QEMU的代码结构非常清晰但是内容非常复杂，这里先简单分析一下总体的结构

1. 开始执行：

主要比较重要的c文件有：/vl.c,/cpus.c, /exec-all.c, /exec.c, /cpu-exec.c.

QEMU的main函数定义在/vl.c中，它也是执行的起点，这个函数的功能主要是建立一个虚拟的硬件环境。它通过参数的解析，将初始化内存，需要的模拟的设备初始化，CPU参数，初始化KVM等等。接着程序就跳转到其他的执行分支文件如：/cpus.c, /exec-all.c, /exec.c, /cpu-exec.c.

2. 硬件模拟

所有的硬件设备都在/hw/ 目录下面，所有的设备都有独自的文件，包括总线，串口，网卡，鼠标等等。它们通过设备模块串在一起，在vl.c中的machine _init中初始化。这里就不讲每种设备是怎么实现的了。

3.目标机器

现在QEMU模拟的CPU架构有：Alpha, ARM, Cris, i386, M68K, PPC, Sparc, Mips, MicroBlaze, S390X and SH4.

我们在QEMU中使用./configure 可以配置运行的架构，这个脚本会自动读取本机真实机器的CPU架构，并且编译的时候就编译对应架构的代码。对于不同的QEMU做的事情都不同，所以不同架构下的代码在不同的目录下面。/target-arch/目录就对应了相应架构的代码，如/target-i386/就对应了x86系列的代码部分。虽然不同架构做法不同，但是都是为了实现将对应客户机CPU架构的TBs转化成TCG的中间代码。这个就是TCG的前半部分。

4.主机

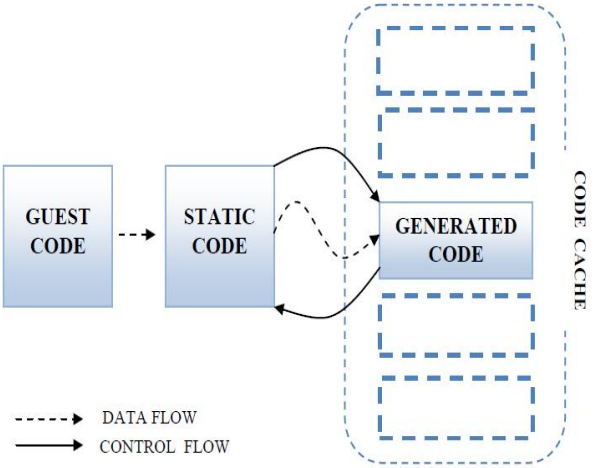
这个部分就是使用TCG代码生成主机的代码，这部分代码在/tcg/里面，在这个目录里面也对应了不同的架构，分别在不同的子目录里面，如i386就在/tcg/i386中。整个生成主机代码的过程也可以教TCG的后半部分。

5.文件总结和补充：

/vl.c:	最主要的模拟循环，虚拟机机器环境初始化，和CPU的执行。
/target-arch/translate.c	将客户机代码转化成不同架构的TCG操作码。
/tcg/tcg.c	主要的TCG代码。
/tcg/arch/tcg-target.c	将TCG代码转化生成主机代码
/cpu-exec.c	其中的cpu-exec()函数主要寻找下一个TB（翻译代码块），如果没找到就请求得到下一个TB，并且操作生成的代码块。

2. TCG - 动态翻译

QEMU在 0.9.1版本之前使用DynGen翻译c代码.当我们需要的时候TCG会动态的转变代码，这个想法的目的是用更多的时间去执行我们生成的代码。当新的代码从TB中生成以后， 将会被保存到一个cache中，因为很多相同的TB会被反复的进行操作，所以这样类似于内存的cache，能够提高使用效率。而cache的刷新使用LRU算法。



编译器在执行器会从源代码中产生目标代码，像GCC这种编译器，它为了产生像函数调用目标代码会产生一些特殊的汇编目标代码，他们能够让编译器需要知道在调用函数。需要什么，以及函数调用以后需要返回什么，这些特殊的汇编代码产生过程就叫做函数的Prologue和Epilogue，这里就叫前端和后段吧。我在其他文章中也分析过汇编调用函数的过程，至于汇编里面函数调用过程中寄存器是如何变化的，在本文中就不再描述了。

函数的后端会恢复前端的状态，主要做下面2点：

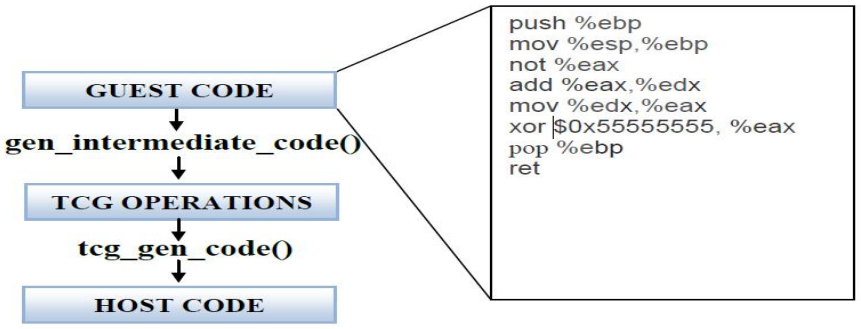
1. 恢复堆栈的指针，包括栈顶和基地址。

2. 修改cs和ip，程序回到之前的前端记录点。

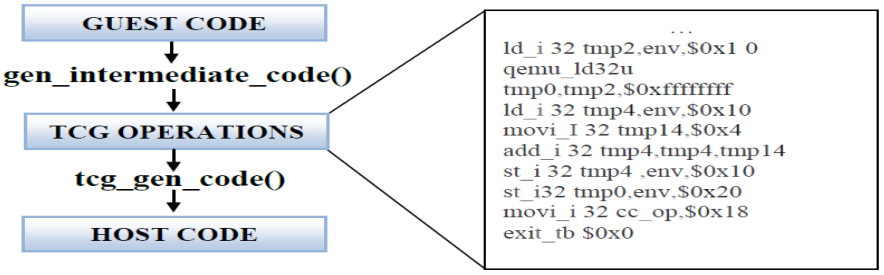
TCG就如编译器一样可以产生目标代码，代码会保存在缓冲区中，当进入前端和后端的时候就会将TCG生成的缓冲代码插入到目标代码中。

接下来我们就来看下如何翻译代码的：

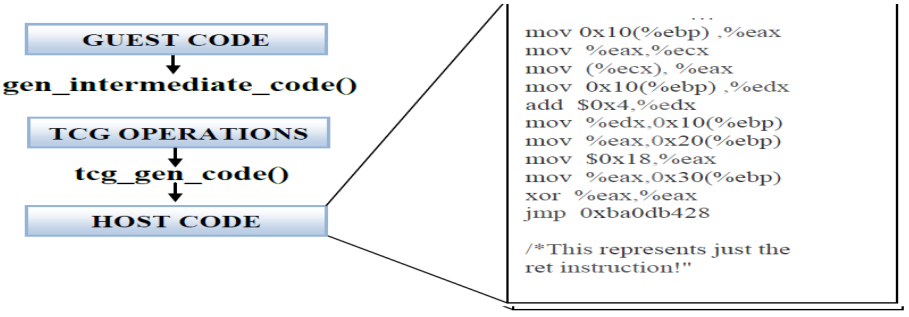
客户机代码



TCG中间代码

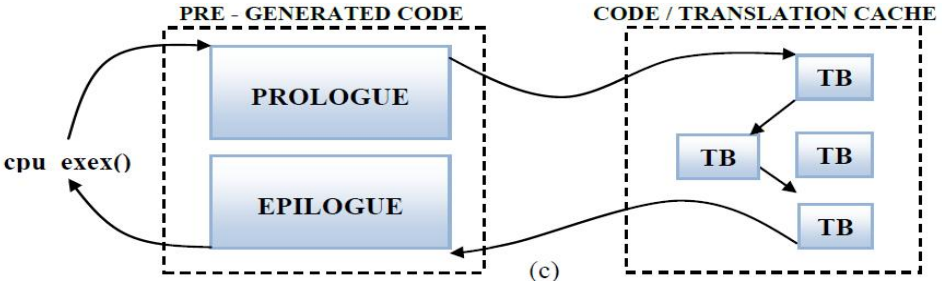


主机代码



3. TB链

在QEMU中，从代码cache到静态代码再回到代码cache，这个过程比较耗时，所以在QEMU中涉及了一个TB链将所有TB连在一起，可以让一个TB执行完以后直接跳到下一个TB，而不用每次都返回到静态代码部分。具体过程如下图：



4. QEMU的TCG代码分析

接下来来看看QEMU代码中中到底怎么来执行这个TCG的，看看它是如何生成主机代码的。

main_loop(...){/vl.c} :

函数main_loop 初始化qemu_main_loop_start() 然后进入无限循环cpu_exec_all() ， 这个是QEMU的一个主要循环，在里面会不断的判断一些条件，如虚拟机的关机断电之类的。

qemu_main_loop_start(...){/cpus.c} :

函数设置系统变量 qemu_system_ready = 1并且重启所有的线程并且等待一个条件变量。

cpu_exec_all(...){/cpus.c} :

它是cpu循环，QEMU能够启动256个cpu核，但是这些核将会分时运行，然后执行qemu_cpu_exec() 。

struct CPUState{/target-xyz/cpu.h} :

它是CPU状态结构体，关于cpu的各种状态，不同架构下面还有不同。

cpu_exec(...){/cpu-exec.c}:

这个函数是主要的执行循环，这里第一次翻译之前说道德TB，TB被初始化为(TranslationBlock *tb) ，然后不停的执行异常处理。其中嵌套了两个无限循环 find tb_find_fast() 和tcg_qemu_tb_exec().

cantb_find_fast()为客户机初始化查询下一个TB，并且生成主机代码。

tcg_qemu_tb_exec()执行生成的主机代码

struct TranslationBlock {/exec-all.h}:

结构体*TranslationBlock*包含下面的成员：PC, CS_BASE, Flags （表明TB）, tc_ptr (指向这个TB翻译代码的指针), tb_next_offset[2], tb_jmp_offset[2] (接下去的Tb), *jmp_next[2], *jmp_first (之前的TB).

tb_find_fast(...){/cpu-exec.c} :

函数通过调用获得程序指针计数器，然后传到一个哈希函数从 tb_jmp_cache[]（一个哈希表）得到TB的所以，所以使用tb_jmp_cache可以找到下一个TB。如果没有找到下一个TB，则使用tb_find_slow。

tb_find_slow(...){/cpu-exec.c}:

这个是在快速查找失败以后试图去访问物理内存，寻找TB。

tb_gen_code(...){/exec.c}:

开始分配一个新的TB，TB的PC是刚刚从CPUstate里面通过using get_page_addr_code()找到的

phys_pc = get_page_addr_code(env, pc);

tb = tb_alloc(pc);

ph当调用cpu_gen_code() 以后，接着会调用tb_link_page() ，它将增加一个新的TB，并且指向它的物理页表。

cpu_gen_code(...){translate-all.c}:

函数初始化真正的代码生成，在这个函数里面有下面的函数调用：

gen_intermediate_code(){/target-arch/translate.c}->gen_intermediate_code_internal(){/target-arch/translate.c }->disas_insn(){/target-arch/translate.c}

disas_insn(){/target-arch/translate.c}

函数disas_insn() 真正的实现将客户机代码翻译成TCG代码，它通过一长串的switch case，将不同的指令做不同的翻译，最后调用tcg_gen_code。

tcg_gen_code(...){/tcg/tcg.c}:

这个函数将TCG的代码转化成主机代码，这个就不细细说明了，和前面类似。

#define tcg_qemu_tb_exec(...){/tcg/tcg.g}:

通过上面的步骤，当TB生成以后就通过这个函数进行执行.

next_tb = tcg_qemu_tb_exec(tc_ptr) :

extern uint8_t code_gen_prologue[];

#define tcg_qemu_tb_exec(tb_ptr) ((long REGPARM(*)(void *)) code_gen_prologue)(tb_ptr)

通过上面的步骤我们就解析了QEMU是如何将客户机代码翻译成主机代码的，了解了TCG的工作原理。接下来看看QEMU与KVM是怎么联系的。

5. QEMU中的IOCTL

在QEMU-KVM中，用户空间的QEMU是通过IOCTL与内核空间的KVM模块进行通讯的。

1. 创建KVM

在/vl.c中通过kvm_init()将会创建各种KVM的结构体变量，并且通过IOCTL与已经初始化好的KVM模块进行通讯，创建虚拟机。然后创建VCPU，等等。

2. KVM_RUN

这个IOCTL是使用最频繁的，整个KVM运行就不停在执行这个IOCTL，当KVM需要QEMU处理一些指令和IO等等的时候就会退出通过这个IOCTL退回到QEMU进行处理，不然就会一直在KVM中执行。

它的初始化过程：

vl.c中调用machine->init初始化硬件设备接着调用pc_init_pci，然后再调用pc_init1。

接着通过下面的调用初始化KVM的主循环，以及CPU循环。在CPU循环的过程中不断的执行KVM_RUN与KVM进行交互。

pc_init1->pc_cpus_init->pc_new_cpu->cpu_x86_init->qemu_init_vcpu->kvm_init_vcpu->ap_main_loop->kvm_main_loop_cpu->kvm_cpu_exec->kvm_run

3.KVM_IRQ_LINE

这个IOCTL和KVM_RUN是不同步的，它也是个频率非常高的调用，它就是一般中断设备的中断注入入口。当设备有中断就通过这个IOCTL最终调用KVM里面的kvm_set_irq将中断注入到虚拟的中断控制器。在kvm中会进一步判断属于什么中断类型，然后在合适的时机写入vmcs。当然在KVM_RUN中会不断的同步虚拟中断控制器，来获取需要注入的中断，这些中断包括QEMU和KVM本身的，并在重新进入客户机之前注入中断。

总结: 通过这篇文章能够大概的了解QEMU的代码结构，其中主要包括TCG翻译代码的过程以及QEMU和KVM的交互过程。

更多 0

上一篇 QEMU, a Fast and Portable Dynamic Translator

下一篇 2.6.36 内核模块时间同步函数汇总

主题推荐 虚拟机 结构 kvm qemu 系统架构

查看评论

7楼 zdcs 2013-12-26 20:04发表

qemu detailed study 只是第七章,其他的有吗???

6楼 onfoot 2012-11-12 11:50发表

走到这里，我对qemu-kvm的理解又深入了

5楼 廖伟强-廖哥 2012-10-10 17:23发表

入门的好文章！！

4楼 xingyu3604 2012-04-27 10:55发表

qemu detailed study 可以从哪里下载或者楼主发我一份

Re: 廖伟强-廖哥 2012-10-10 17:23发表

回复xingyu3604：http://lists.gnu.org/archive/html/qemu-devel/2011-04/pdfhC5rVdz7U8.pdf

3楼 viktor 2012-02-29 15:18发表

这篇文章好像是GNU的。

2楼 viktor 2012-02-29 15:15发表

同问~

1楼 abin116179 2012-02-23 10:14发表

qemu detailed study 现在从哪里可以下载?

Re: 廖伟强-廖哥 2012-10-10 17:23发表

回复abin116179：http://lists.gnu.org/archive/html/qemu-devel/2011-04/pdfhC5rVdz7U8.pdf

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap