**QEMU Internal – Tiny Code Generator (TCG) 1/2**

Copyright (c) 2011 陈韦任(Chen Wei-Ren)

前言

因为工作上的关系，必须接触QEMU。虽然网路上有不少文件，但总觉得讲得不够深入。QEMU 是一个仿真器(emulator)，可以process mode 或是system mode 运行。process mode 可以运行不同ISA 同一OS 的binary; system mode 可以在当前作业系统上运行另外一个OS。我在收集各方资料，阅读代码和在邮件列表上发问之后，觉得略有心得。在此对QEMU internal 作一个较为深入的介绍。凭我个人之力，难免有疏漏或是错误。权且当作抛砖引玉吧。希望各位不吝指教。

0. 术语、线上资源和技巧

对QEMU 而言，被仿真的平台被称为guest，又称target; 运行QEMU 的平台称为host。QEMU 是利用动态翻译(dynamic translation) 的技术将guest binary 动态翻译成host binary，并交由host 运行翻译所得的host binary。Tiny Code Generator (TCG) 是QEMU 中负责动态翻译的组件。对TCG 而言，target 有不同的含意，它代表TCG 是针对哪一个host 生成host binary。

网路上对QEMU 有较为完整描述的文件为:

* [QEMU, a Fast and Portable Dynamic Translator](http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf)
* [Porting QEMU to Plan 9: QEMU Internals and Port Strategy](http://gsoc.cat-v.org/people/nwf/paper-strategy-plus.pdf)

然而需要注意的是，上述文件在动态翻译的部分均是针对QEMU 0.9版。QEMU 0.9版以前是使用dyngen技术; QEMU 0.10版以后采用TCG。虽说如此，但在QEMU的其它部分差异不大，上述文件仍可供参考。[SOURCEARCHIVE.com](http://qemu.sourcearchive.com/)收集了自QEMU 0.6.1版至今的所有QEMU源代码。各位可以边看文件边看源代码。

QEMU极为依赖macro，这使得直接阅读源代码通常无法确定其函数呼叫，或是执行流程倒底为何。请在编译QEMU的时候加上--extra-cflags="-save-temps"，如此可得展开marco的\*.i档。

其余部分请见:

* [Getting started for developers](http://wiki.qemu.org/Documentation/GettingStartedDevelopers)
* [QEMU Internals](http://qemu.weilnetz.de/qemu-tech.html)
* QEMU 目录下的HACKING、CODING\_STYLE、tcg/README 和doc/\*
* ISA reference manual。由于本篇文章是以x86为例，关于x86指令请见  [x86/x64指令编码内幕](http://www.mouseos.com/x64/index.html)。

1. TCG

TCG 是QEMU 的核心。其基本流程如下:

guest binary -> TCG IR -> host binary

1.1 TCG IR

TCG 定义了一组IR (intermediate representation)，熟悉GCC 的各位对此应该不陌生。TCG IR 大致分成以下几类:

* Move Operation: mov, movi, …
* Logic Operation: and, or, xor, shl, shr, …
* Arithmetic peration: add, sub, mul, div, …
* Branch Operation: jmp, br, brcond
* Fuction call: call
* Memory Operation: ld, st
* QEMU specific Operation: tb\_exit, goto\_tb, qemu\_ld/qemu\_st

请见tcg/\*，特别是tcg.i，可以看到TCGOpcode。tcg/README也别忘了。TCG在翻译guest binary的时候是以一个translation block (tb)为单位，其结尾通常是分支指令。

target-ARCH/\* 定义了如何将ARCH binary 反汇编成TCG IR。tcg/ARCH 定义了如何将TCG IR 翻译成ARCH binary。

1.2 TCG Flow

先介绍一些资料结构:

* gen\_opc\_buf 和gen\_opparam\_buf (translate-all.c) 分别放置TCG Opcode 和Operand。
* 如果使用静态配置的缓冲区，static\_code\_gen\_buffer (exec.c) 即为code cache，放置host binary。
* 在跳入/出code cache 执行之前/后，要执行prologue/epilogue，请见code​​\_gen\_prologue (exec.c)。这边的prologue/epilogue 就是指function prologue/epilogue。QEMU 将跳至code cache (host binary) 执行的过程看成是函式呼叫，故有此prologue/epilogue。

以qemu-i386 为例，流程大致如下:

main (linux-user/main.c) -> cpu\_exec\_init\_all (exec.c)

-> cpu\_init/cpu\_x86\_init (target-i386/helper.c)

-> tcg\_prologue\_init (tcg/tcg.c) -> cpu\_loop (linux-user/main.c)

函式名之所以会出现cpu\_init/cpu\_x86\_init，是因为QEMU 经常使用#define 替换函式名。cpu\_init 是main 里呼叫的函式，经#define 替换后，实际上是cpu\_x86\_init (target-i386/helper.c)。GDB 下断点时请注意此种情况。

这边只介绍tcg\_prologue\_init (tcg/tcg.c) -> cpu\_loop (linux-user/main.c) 这一段，因为这一段跟TCG 较为相关。容我先讲cpu\_loop (linux-user/main.c)。

* cpu\_loop (linux-user/main.c) -> cpu\_x86\_exec/cpu\_exec (cpu-exec.c)。cpu\_exec 是主要执行回圈，其结构大致如下:

*/\* prepare setjmp context for exception handling \*/*

for ( ;; ) {

if ( setjmp ( env -> jmp\_env ) == 0 ) { *//例外处理。*

}

next\_tb = 0 ; */\* force lookup of first TB \*/*

for ( ;; ) {

*//判断是否有中断。若有，跳回例外处理。*

next\_tb = tcg\_qemu\_tb\_exec ( tc\_ptr ) ; *//跳至code cache执行。*

}

}

* tcg\_prologue\_init (tcg/tcg.c) -> tcg\_target\_qemu\_prologue (tcg/i386/tcg-target.c)。如前所述，QEMU 将跳至code cache (host binary) 执行的过程看成是函式呼叫。不同平台的calling convention 各有不同，tcg\_prologue\_init 将产生prologue/epilogue 的工作转交tcg\_target\_qemu\_prologue。

static void tcg\_target\_qemu\_prologue ( TCGContext \* s )

{

*/\* QEMU (cpu\_exec) ->入栈\*/*

*// OPC\_GRP5 (0xff)为call，EXT5\_JMPN\_Ev是其opcode extension。*

*// tcg\_target\_call\_iarg\_regs是函式呼叫负责传递参数的暂存器。*

*//跳至code cache执行。*

tcg\_out\_modrm ( s , OPC\_GRP5 , EXT5\_JMPN\_Ev , tcg\_target\_call\_iarg\_regs [ 0 ] ) ;

*//此时，s->code\_ptr指向code\_gen\_prologue中prologue和jmp to code cache*

*//之后的位址。*

*// tb\_ret\_addr是纪录code cache跳回code\_gen\_prologue的哪个地方。*

tb\_ret\_addr = s -> code\_ptr ;

*/\*出栈->返回QEMU (cpu\_exec)，确切的讲是返回tcg\_qemu\_tb\_exec \*/*

}

这边小结一下。

QEMU -> prologue -> code cache -> epilogue -> QEMU

tb\_ret\_addr 就是用来由code cache 返回至code\_gen\_prologue，执行epilogue，再返回QEMU。

在介绍cpu\_exec之前，我先介绍几个QEMU资料结构，请善用[SOURCEARCHIVE.com](http://qemu.sourcearchive.com/)。我们要知道所谓仿真或是虚 ​​拟化一个CPU (ISA)，简单来说就是用一个资料结构(struct)储存该CPU的状态。执行该虚拟CPU，就是从内存中读取该虚拟CPU的资料结构，运算后再存回去。

* CPUX86State: 保存x86 register，eflags，eip，cs，…。不同ISA 之间通用的资料结构被QEMU #define 成CPU\_COMMON。一般称此资料结构为CPUState。下文所提env 即为CPUState。QEMU 运行虚拟CPU 都会利用env 这个变数。
* TranslationBlock: 之前说过，QEMU 是以一个translation block 为单位进行翻译。其中保存此translation block 对应guest binary 的pc, cs\_bas​​e, eflags。另外，tc\_ptr 指向code cache (host binary)。其它栏位待以后再谈。
* PageDesc: 主要保存guest page 中的第一个tb (TranslationBlock \*)。这跟QEMU 内部运作机制有关。某些情况下，guest page (guest binary) 可能被替换或是被写。这个时候，QEMU 会以guest page (guest binary) 为单位，清空与它相关联的TB (code cache)。这时再回来讲 TranslationBlock。TranslationBlock 有底下两个栏位:
  + page\_addr[2]: 存放TranslationBlock 对应guest binary 所在的guest page。注意! guest binary 有可能跨guest page，故这里有两个栏位。
  + page\_next[2]: 当透过PageDesc->first\_tb 找到该guest page 的第一个tb，tb->page\_next 就被用来找寻该guest page 的下一个tb。

再回来讲PageDesc。QEMU 替PageDesc 维护了一个二级页表l1\_map。page\_find 这个函式根据输入的address 搜寻l1\_map，返回PageDesc。这在以guest page (guest binary) 为单位，清空与它相关联的TB (code cache) 的时候会用到。有一个名字很像的资料结构叫PhysPageDesc，QEMU 也替它维护一个二级页表l1\_phys\_map。这是在system mode 做地址转换之用，这边不谈。

* TCGContext: 生成TCG IR 时会用到。
* DisasContext: 反汇编guest binary 时会用到。

# QEMU Internal – Tiny Code Generator (TCG) 2/2

Copyright (c) 2011 陈韦任(Chen Wei-Ren)

1.2 TCG Flow

介绍完一些资料结构之后，我开始介绍cpu\_exec 的流程。底下复习一下process mode 的流程:

cpu\_loop (linux-user/main.c) -> cpu\_x86\_exec/cpu\_exec (cpu-exec.c)

cpu\_exec 有两层for 回圈。我们先看内层:

next\_tb = 0 ; */\* force lookup of first TB \*/*

for ( ;; ) {

tb = tb\_find\_fast ( ) ;

tc\_ptr = tb -> tc\_ptr ;

next\_tb = tcg\_qemu\_tb\_exec ( tc\_ptr ) ;

}

* tb\_find\_fast 会先试图查看目前pc (guest virtual address) 是否已有翻译过的host binary 存放在code cache。

*// pc = eip + cs\_base*

cpu\_get\_tb\_cpu\_state ( env , & pc , & cs\_base , & flags ) ;

*// CPUState中的tb\_jmp\_cache即是做此用途。*

tb = env -> tb\_jmp\_cache [ tb\_jmp\_cache\_hash\_func ( pc ) ] ;

*//检查该tb是否合法。这是因为不同的eip + cs\_base可能会得到相同的pc。*

if ( unlikely ( ! tb || tb -> pc != pc || tb -> cs\_base != cs\_base ||

tb -> flags != flags ) ) {

tb = tb\_find\_slow ( pc , cs\_base , flags ) ;

}

*// code cache已有翻译过的host binary，返回TranslationBlock。*

return tb ;

* tb\_find\_slow 以pc 对映的物理位址(guest physcal address) 查找TB。如果成功，则将该TB 写入env->tb\_jmp\_cache; 若否，则进行翻译。

phys\_pc = get\_page\_addr\_code ( env , pc ) ;

phys\_page1 = phys\_pc & TARGET\_PAGE\_MASK ;

*//除了env->tb\_jmp\_cache这个以guest virtual address为索引的缓存之外，*

*// QEMU还维护了一个tb\_phys\_hash，这个是以guest physical address为索引。*

h = tb\_phys\_hash\_func ( phys\_pc ) ;

ptb1 = & tb\_phys\_hash [ h ] ;

for ( ;; ) {

not\_found :

found :

*// TranslationBlock中的phys\_hash\_next用在这里。*

*//如果phys\_pc索引到同一个tb\_phys\_hash栏位，用phys\_hash\_next串接起来。*

ptb1 = & tb -> phys\_hash\_next ;

}

not\_found :

tb = tb\_gen\_code ( env , pc , cs\_base , flags , 0 ) ;

found :

env -> tb\_jmp\_cache [ tb\_jmp\_cache\_hash\_func ( pc ) ] = tb ;

return tb ;

这里小结一下流程。

cpu\_exec (cpu-exec.c) -> tb\_find\_fast (cpu-exec.c)

-> tb\_find\_slow (cpu-exec.c)

QEMU 先以guest virtual address (GVA) 查找是否已有翻译过的TB，再以guest physical address (GPA) 查找是否已有翻译过的TB。

如果没有翻译过的TB，开始进行guest binary -> TCG IR -> host binary 的翻译。大致流程如下:

tb\_gen\_code (exec.c) -> cpu\_gen\_code (translate-all.c)

-> gen\_intermediate\_code (target-i386/translate.c)

-> tcg\_gen\_code (tcg/tcg.c) -> tcg\_gen\_code\_common (tcg/tcg.c)

* + tb\_gen\_code 配置内存给TB (TranslationBlock)，再交由cpu\_gen\_code。

*//注意!这里会将GVA转成GPA。phys\_pc将交给之后的tb\_link\_page使用。*

phys\_pc = get\_page\_addr\_code ( env , pc ) ;

tb = tb\_alloc ( pc ) ;

if ( ! tb ) {

*//清空code cache*

}

*// 初始tb*

*//开始guest binary -> TCG IR -> host binary的翻译。*

cpu\_gen\_code ( env , tb , & code\_gen\_size ) ;

*//将tb加入tb\_phys\_hash和二级页表l1\_map。*

*// phys\_pc和phys\_page2分别代表tb (guest pc)对映的GPA和所属的第二个*

*//页面(如果tb代表的guest binary跨页面的话)。*

tb\_link\_page ( tb , phys\_pc , phys\_page2 ) ;

return tb ;

我底下分别针对cpu\_gen\_code 和tb\_link\_page 稍微深入的介绍一下。

* + cpu\_gen\_code 负责guest binary -> TCG IR -> host binary 的翻译。

*//初始TCGContext的gen\_opc\_ptr和gen\_opparam\_ptr，使其分别指向*

*// gen\_opc\_buf和gen\_opparam\_buf。gen\_opc\_buf和gen\_opparam\_buf*

*//分别存放TCGOpcode和operand。*

tcg\_func\_start ( s ) ;

*//呼叫gen\_intermediate\_code\_internal产生TCG IR*

gen\_intermediate\_code ( env , tb ) ;

*// TCG IR -> host binary*

gen\_code\_size = tcg\_gen\_code ( s , gen\_code\_buf ) ;

* + gen\_intermediate\_code\_internal (target-\*/translate.c) 初始化并呼叫disas\_insn 反组译guest binary 成TCG IR。disas\_insn 呼叫tcg\_gen\_xxx (tcg/tcg-op.h) 产生TCG IR。分别将opcode 写入gen\_opc\_ptr 指向的缓冲区(translate-all.c 里的gen\_opc\_buf); operand 写入gen\_opparam\_ptr 指向的缓冲区(translate-all.c 里的gen\_opparam\_buf)。
  + tcg\_gen\_code (tcg/tcg.c) 呼叫tcg\_gen\_code\_common (tcg/tcg.c) 将TCG IR 转成host binary。

tcg\_reg\_alloc\_start ( s ) ;

s -> code\_buf = gen\_code\_buf ;

*// host binary会写入TCGContext s的code\_ptr所指向的缓冲区。*

s -> code\_ptr = gen\_code\_buf ;

至此，guest binary -> TCG IR -> host binary 算是完成了。剩下把TranslationBlock (TB) 纳入QEMU 的管理，这是tb\_link\_page 做的事。

* + tb\_link\_page (exec.c) 把新的TB 加进tb\_phys\_hash 和l1\_map 二级页表。tb\_find\_slow 会用pc 对映的GPA 的哈希值索引tb\_phys\_hash。

*/\*把新的TB加进tb\_phys\_hash \*/*

h = tb\_phys\_hash\_func ( phys\_pc ) ;

ptb = & tb\_phys\_hash [ h ] ;

*//如果两个以上的TB其phys\_pc的哈希值相同，则做chaining。*

tb -> phys\_hash\_next = \* ptb ;

\* ptb = tb ; *//新加入的TB放至chaining的开头。*

*//在l1\_map中配置PageDesc给TB，并设置TB的page\_addr和page\_next。*

tb\_alloc\_page ( tb , 0 , phys\_pc & TARGET\_PAGE\_MASK ) ;

if ( phys\_page2 != - 1 ) *// TB对应的guest binary跨页*

tb\_alloc\_page ( tb , 1 , phys\_page2 ) ;

else

tb -> page\_addr [ 1 ] = - 1 ;

*//以下和block chaining有关，留待下次再讲，这边暂且不提。*

tb -> jmp\_first = ( TranslationBlock \* ) ( ( long ) tb | 2 ) ;

* + tb\_alloc\_page (exec.c) 设置TB 的page\_addr 和page\_next，并在l1\_map 中配置PageDesc 给TB。

static **inline**  void tb\_alloc\_page ( TranslationBlock \* tb ,

unsigned int n , tb\_page\_addr\_t page\_addr )

{

*//代表tb (guest binary)所属guest page。*

tb -> page\_addr [ n ] = page\_addr ;

*//在l1\_map中配置一个PageDesc，返回该PageDesc。*

p = page\_find\_alloc ( page\_addr >> TARGET\_PAGE\_BITS , 1 ) ;

*//将该页面目前第一个TB串接到此TB。将来有需要将某页面所属所有TB清空。*

tb -> page\_next [ n ] = p -> first\_tb ;

*// n为1代表tb对应的guest binary跨page。*

p -> first\_tb = ( TranslationBlock \* ) ( ( long ) tb | n ) ;

*// PageDesc会维护一个bitmap，这是给SMC之用。这里不提。*

invalidate\_page\_bitmap ( p ) ;

}

这里先回顾一下，QEMU 查看当前env->pc 是否已翻译过。若否，则进行翻译。

tb\_find\_fast (cpu-exec.c) -> tb\_find\_slow (cpu-exec.c)

-> tb\_gen\_code (exec.c)

tb\_gen\_code 讲到这里，guest binary -> host binary 已翻译完成，相关资料结构也已设置完毕。返回TB (TranslationBlock \*) 给tb\_find\_fast。

tb = tb\_find\_fast ( ) ;

tc\_ptr = tb -> tc\_ptr ; *// tc\_ptr指向code cache (host binary)*

next\_tb = tcg\_qemu\_tb\_exec ( tc\_ptr ) ;

很好，我们准备从QEMU 跳入code cache 开始执行了。:-) tcg\_qemu\_tb\_exec 被定义在tcg/tcg.h。

#define tcg\_qemu\_tb\_exec(tb\_ptr) ((long REGPARM (\*)(void \*))code\_gen\_prologue)(tb\_ptr)

(long REGPARM (\*)(void \*))将code\_gen\_prologue转型成函式指针，void \*为该函式的参数，返回值为long。REGPARM指示GCC此函式透过暂存器而非栈传递参数。至此，(long REGPARM (\*)(void \*))将数组指针code\_gen\_prologue转型成函式指针。tb\_ptr为该函式指针的参数。综合以上所述，code\_gen\_prologue被视为一函式，其参数为tb\_ptr，返回当前TB (tc\_ptr代表的TB，等讲到block chaining会比较清楚)。code\_gen\_prologue所做的事为一般函式呼叫前的prologue，之后将控制交由tc\_ptr指向的host binary并开始执行。

# QEMU Internal – Block Chaining 1/3

Copyright (c) 2011 陈韦任(Chen Wei-Ren)

2. Block Chaining

先复习一下QEMU 的流程，目前的情况如底下这样:

QEMU -> code cache -> QEMU -> code cache -> ...

之前提到，QEMU 是以一个translation block 为单位进行翻译并执行。这也就是说，每当在code cache 执行完一个translation block 之后，控制权必须交还给QEMU。这很没有效率。理想情况下，大部分时间应该花在code cache 中执行，只要在必要情况下才需要回到QEMU。基本想法是把code cache 中的translation block 串接起来。只要translation block 执行完之后，它的跳跃目标确定且该跳跃目标也已经在code cache 里，那我们就把这两个translation block 串接起来。这个就叫做block chaining/linking。

在QEMU 中，达成block chaining 有两种做法: 第一，采用direct jump。此法直接修改code cache 中分支指令的跳跃目标，因此依据host 有不同的patch 方式。第二，则是透过修改TranslationBlock 的tb\_next 栏位达成block chaining。exec-all.h 中定义那些host 支援使用direct jump。

这里只介绍direct jump。我们先回到cpu\_exec (cpu-exec.c) 的内层回圈。

tb = tb\_find\_fast ( env ) ;

if ( tb\_invalidated\_flag ) {

next\_tb = 0 ; *//注意! next\_tb也被用来控制是否要做block chaining。*

tb\_invalidated\_flag = 0 ;

}

*//注意!! next\_tb的名字会让人误解。block chaining的方向为: next\_tb -> tb。*

*// next\_tb不为NULL且tb (guest binary)不跨guest page的话，做block*

*// chaining。原因之后再讲。*

if ( next\_tb != 0 && tb -> page\_addr [ 1 ] == - 1 ) {

*//这边利用TranlationBlock指针的最低有效位后两位指引block chaining的方向。*

tb\_add\_jump ( ( TranslationBlock \* ) ( next\_tb & ~ 3 ) , next\_tb & 3 , tb ) ;

}

*//执行TB，也就是tc\_ptr所指到的位址。注意，产生TCG IR的过程中，在block的最后会是*

*// exit\_tb addr，此addr是正在执行的TranslationBlock指针，同时也是tcg\_qemu\_tb\_exec*

*//的回传值。该位址后两位会被填入0、1或2以指示block chaining的方向。*

next\_tb = tcg\_qemu\_tb\_exec ( tc\_ptr ) ;

是不是有点晕头转向?我们来仔细检验guest binary -> TCG IR -> host binary到底怎么做的。在一个translation block的结尾，TCG都会产生TCG IR exit\_tb。我们来看看。:-)

* tcg\_gen\_exit\_tb (tcg/tcg-op.h)呼叫tcg\_gen\_op1i生成TCG IR，其opcode为INDEX\_op\_exit\_tb (还记得tcg.i里的TCGOpcode吗?)，operand为val。

*//一般是这样呼叫: tcg\_gen\_exit\_tb((tcg\_target\_long)tb + tb\_num);*

*//注意!请留意其参数: (tcg\_target\_long)tb + tb\_num。*

static **inline**  void tcg\_gen\_exit\_tb ( tcg\_target\_long val )

{

*//将INDEX\_op\_exit\_tb写入gen\_opc\_buf; val写入gen\_opparam\_buf。*

tcg\_gen\_op1i ( INDEX\_op\_exit\_tb , val ) ;

}

* tcg/ARCH/tcg-target.c 根据TCG IR 产生对应host binary。以i386 为例: (关于每一行的作用，我是凭经验用猜的。如有错请指正。)

static **inline**  void tcg\_out\_op ( TCGContext \* s , int opc ,

const TCGArg \* args , const int \* const\_args )

{

*//总和效果:返回QEMU。*

*// next\_tb = tcg\_qemu\_tb\_exec(tc\_ptr);*

*//*

*//图示:*

*// struct TranslationBlock\* code cache*

*// next\_tb->tc\_ptr -> tb*

*//*

*// next\_tb的末两位编码next\_tb条件分支的方向。*

*//等待下一次回圈取得tb = tb\_find\_fast()，*

*//试图做block chaining: next\_tb -> tb*

*//*

case INDEX\_op\_exit\_tb :

*// QEMU把跳至code cache执行当作是函式呼叫，EAX存放返回值。*

*//将val写进EAX，val是(tcg\_target\_long)tb + tb\_num。*

tcg\_out\_movi ( s , TCG\_TYPE\_I32 , TCG\_REG\_EAX , args [ 0 ] ) ;

*// e9是jmp指令，后面的operand为相对偏移量，将会加上eip。*

*//底下两条的总和效果是跳回code\_gen\_prologue中prologue以后的位置。*

tcg\_out8 ( s , 0xe9 ) ; */\* jmp tb\_ret\_addr \*/*

*// tb\_ret\_addr在tcg\_target\_qemu\_prologue初始成指向*

*// code\_gen\_prologue中prologue以后的位置。*

*//生成host binary的同时，s->code\_ptr会移向下一个code buffer的位址。*

*//所以要减去4。*

tcg\_out32 ( s , tb\_ret\_addr - s -> code\_ptr - 4 ) ;

**break** ;

}

* + tcg\_out\_movi 将arg 移至ret 代表的暂存器。

static **inline**  void tcg\_out\_movi ( TCGContext \* s , TCGType type ,

int ret , int32\_t arg )

{

if ( arg == 0 ) {

*/\* xor r0,r0 \*/*

tcg\_out\_modrm ( s , 0x01 | ( ARITH\_XOR << 3 ) , ret , ret ) ;

} else {

*// move arg至ret*

*// 0xb8为move，ret代表目地暂存器。*

*// 0xb8 + ret合成一个opcode。*

tcg\_out8 ( s , 0xb8 + ret ) ;

tcg\_out32 ( s , arg ) ;

}

}

小结一下，QEMU 函式名称命名惯例为:

* tcg\_gen\_xxx – 产生TCG Opcode 和operand 至gen\_opc\_buf 和gen\_opparam\_buf。
* tcg\_out\_xxx – 产生host binary 至TCGContext 所指的code cache。

# QEMU Internal – Block Chaining 2/3

Copyright (c) 2011 陈韦任(Chen Wei-Ren)

2. Block Chaining

我们再回到cpu\_exec (cpu-exec.c) 的内层回圈。这边主要看tb\_add\_jump。

if ( next\_tb != 0 && tb -> page\_addr [ 1 ] == - 1 ) {

*//这边利用TranlationBlock指针的最低有效位后两位指引block chaining的方向。*

*// next\_tb -> tb*

tb\_add\_jump ( ( TranslationBlock \* ) ( next\_tb & ~ 3 ) , next\_tb & 3 , tb ) ;

}

*//这边要注意到，QEMU利用TranslatonBlock指针后两位必为零的结果*

*//做了一些手脚。QEMU将其末两位编码成0、1或2来指引block chaing*

*//的方向。这种技巧在QEMU用得非常娴熟。*

next\_tb = tcg\_qemu\_tb\_exec ( tc\_ptr ) ;

* tb\_add\_jump 呼叫tb\_set\_jmp\_target 做block chaining 的patch。另外，会利用tb 的jmp\_next 和tb\_next 的jmp\_first 把block chaining 中的TranslationBlock 串成一个circular list。这是以后做block \*unchaining\* 之用。再复习一下TranslationBlock，
* struct TranslationBlock\* code cache (host binary)
* tb->tc\_ptr -> TB

请注意! 当我们讲TB，可能是指TranslationBlock，也可能是指code cache 中的TB。好! 当我们patch code cache 中的TB，将它们串接在一起。我们同时也要把它们相对应的TranslationBlock 做点纪录，利用jmp\_first 和jmp\_next 这两个栏位。

这边我先回到tb\_link\_page，还记得我留下一些地方没讲嘛? 开始吧! ;-)

*// jmp\_first代表跳至此tb的其它TB中的头一个。jmp\_first初值为自己，末两位为10 (2)。*

*//将来做block chaining时，jmp\_first指向跳至此tb的其它TB中的头一个，tb1，末两位*

*//为00或01，这代表从tb1的哪一个分支跳至此tb。*

tb -> jmp\_first = ( TranslationBlock \* ) ( ( long ) tb | 2 ) ;

*// jmp\_next[n]代表此tb条件分支的目标TB。*

*//注意!如果目标TB，tb2，孤身一人，jmp\_next就真的指向tb2 (符合初次看到jmp\_next所想*

*//的语意)。*

*//*

*//如果已有其它TB，tb3，跳至tb2，则赋值给tb->jmp\_next的是tb2的jmp\_first，也就是*

*// tb3 (末两位编码tb3跳至tb2的方向)。*

tb -> jmp\_next [ 0 ] = NULL ;

tb -> jmp\_next [ 1 ] = NULL ;

*// tb\_next\_offset代表此TB在code cache中分支跳转要被patch的位址(相对于其code cache*

*//的偏移量)，为了direct block chaining之用。*

if ( tb -> tb\_next\_offset [ 0 ] != 0xffff )

tb\_reset\_jump ( tb , 0 ) ;

if ( tb -> tb\_next\_offset [ 1 ] != 0xffff )

tb\_reset\_jump ( tb , 1 ) ;

我们再回来看看tb\_add\_jump 做了什么。:-)

*// block chaining方向为: tb -> tb\_next。n用来指示tb条件分支的方向。*

static **inline**  void tb\_add\_jump ( TranslationBlock \* tb , int n ,

TranslationBlock \* tb\_next )

{

*// jmp\_next[0]/jmp\_next[1]代表tb条件分支的目标。*

if ( ! tb -> jmp\_next [ n ] ) {

*/\* patch the native jump address \*/*

tb\_set\_jmp\_target ( tb , n , ( unsigned long ) tb\_next -> tc\_ptr ) ;

*// tb\_jmp\_remove 会用到jmp\_next 做block unchaining，这个以后再讲。*

*// tb\_next->jmp\_first初值为自己，末两位设为10 (2)。*

*//如果已有其它TB，tb1，跳至tb\_next，则下面这条语句会使得*

*// tb->jmp\_next指向tb1 (末两位代表tb1跳至tb\_next的方向)。*

*// tb\_next->jmp\_first由原本指向tb1改指向tb。*

*//有没有circular list浮现在你脑海中?;-)*

tb -> jmp\_next [ n ] = tb\_next -> jmp\_first ;

*// tb\_next的jmp\_first指回tb，末两位代表由tb哪一个条件分支跳至tb\_next。*

tb\_next -> jmp\_first = ( TranslationBlock \* ) ( ( long ) ( tb ) | ( n ) ) ;

}

}

我们再来看是怎么patch code cache 中分支指令的目标地址。依据是否采用direct jump，tb\_set\_jmp\_target (exec-all.h) 有不同做法。采用direct jump 的话，tb\_set\_jmp\_target 会根据host 呼叫不同的tb\_set\_jmp\_target1。tb\_set\_jmp\_target1 会用到TB 的tb\_jmp\_offset。如果不采用direct jump 做block chaining，tb\_set\_jmp\_target 会直接修改TB 的tb\_next。

* tb\_set\_jmp\_target (exec-all.h)。

static **inline**  void tb\_set\_jmp\_target ( TranslationBlock \* tb ,

int n , unsigned long addr )

{

unsigned long offset ;

*// n可以为0或1，代表分支跳转的分向taken或not taken。*

offset = tb -> tb\_jmp\_offset [ n ] ; *// tb要patch的位址相对于tb->tc\_ptr的偏移量。*

tb\_set\_jmp\_target1 ( ( unsigned long ) ( tb -> tc\_ptr + offset ) , addr ) ;

}

* tb\_set\_jmp\_target1 (exec-all.h)。

#elif defined(\_\_i386\_\_) || defined(\_\_x86\_64\_\_)

static **inline**  void tb\_set\_jmp\_target1 ( unsigned long jmp\_addr , unsigned long addr )

{

*/\* patch the branch destination \*/*

*// jmp的参数为jmp下一条指令与目标地址的偏移量。*

\* ( uint32\_t \* ) jmp\_addr = addr - ( jmp\_addr + 4 ) ;

*/\* no need to flush icache explicitly \*/*

}

你会有这个疑问吗? 在分支跳转位址被patch 到分支跳转指令之前，它是要跳去哪里? :-)

[QEMU – block chaining](http://www.hellogcc.org/wp-content/uploads/2011/10/QEMU-block-chaining.ppt)可以帮助你比较清楚了解block chaining如何运作。

# QEMU Internal – Block Chaining 3/3

Copyright (c) 2011 陈韦任(Chen Wei-Ren)

2. Block Chaining

由guest binary -> TCG IR 的过程中，gen\_goto\_tb 会做block chaining 的准备。我们先来看何时会呼叫到gen\_goto\_tb。以i386 为例，遇到guest binary 中的条件分支和直接跳转都会呼叫gen\_goto\_tb (target-i386/translate.c)。这边以条件分支当例子:

static **inline**  void gen\_jcc ( DisasContext \* s , int b ,

target\_ulong val , target\_ulong next\_eip )

{

int l1 , l2 , cc\_op ;

cc\_op = s -> cc\_op ;

gen\_update\_cc\_op ( s ) ;

if ( s -> jmp\_opt ) { *// use direct block chaining for direct jumps*

l1 = gen\_new\_label ( ) ;

gen\_jcc1 ( s , cc\_op , b , l1 ) ;

gen\_goto\_tb ( s , 0 , next\_eip ) ; *//我猜是taken*

gen\_set\_label ( l1 ) ;

gen\_goto\_tb ( s , 1 , val ) ; *//我猜是not taken*

s -> is\_jmp = DISAS\_TB\_JUMP ;

} else {

*/\* 忽略不提\*/*

}

}

* gen\_goto\_tb。强烈建议阅读[Porting QEMU to Plan 9: QEMU Internals and Port Strategy](http://gsoc.cat-v.org/people/nwf/paper-strategy-plus.pdf) 2.2.3和2.2.4节，也别忘了[SOURCEARCHIVE.com](http://qemu.sourcearchive.com/)。

*// tb\_num代表目前tb block linking分支情况。eip代表跳转目标。*

static **inline**  void gen\_goto\_tb ( DisasContext \* s , int tb\_num , target\_ulong eip )

{

TranslationBlock \* tb ;

target\_ulong pc ;

*// s->pc代表翻译至目前guest binary的所在位址。tb->pc表示guest binary的起始位址。*

*//注意!这里s->cs\_base + eip代表跳转位址; s->pc代表目前翻译到的guest pc。*

pc = s -> cs\_base + eip ; *//计算跳转目标的pc*

tb = s -> tb ; *//目前tb*

*// http://lists.nongnu.org/archive/html/qemu-devel/2011-08 /msg02249.html*

*// http://lists.gnu.org/archive/html/qemu-devel/2011-09/msg03065.html*

*//满足底下两个条件之一，则可以做direct block linking*

*//第一，跳转目标和目前tb起始的pc同属一个guest page。*

*//第二，跳转目标和目前翻译到的pc同属一个guest page。*

if ( ( pc & TARGET\_PAGE\_MASK ) == ( tb -> pc & TARGET\_PAGE\_MASK ) ||

( pc & TARGET\_PAGE\_MASK ) == ( ( s -> pc - 1 ) & TARGET\_PAGE\_MASK ) ) {

*//如果guest jump指令和其跳转位址同属一个guest page，则做direct block linking。*

tcg\_gen\_goto\_tb ( tb\_num ) ; *//生成准备做block linking的TCG IR。详情请见之后描述。*

*//更新env的eip，使其指向此tb之后欲执行指令的位址。*

*// tb\_find\_fast会用eip查找该TB是否已被翻译过。*

gen\_jmp\_im ( eip ) ;

*//最终回到QEMU tcg\_qemu\_tb\_exec，赋值给next\_tb。*

*//注意! tb\_num会被next\_tb & 3取出，由此可以得知block chaining的方向。*

tcg\_gen\_exit\_tb ( ( tcg\_target\_long ) tb + tb\_num ) ;

} else {

*/\* jump to another page: currently not optimized \*/*

gen\_jmp\_im ( eip ) ;

gen\_eob ( s ) ;

}

}

* + tcg\_gen\_goto\_tb 生成TCG IR。

static **inline**  void tcg\_gen\_goto\_tb ( int idx )

{

tcg\_gen\_op1i ( INDEX\_op\_goto\_tb , idx ) ;

}

* + tcg\_out\_op (tcg/i386/tcg-target.c) 将TCG IR 翻成host binary。注意! 这边利用patch jmp 跳转位址达成block linking。

static **inline**  void tcg\_out\_op ( TCGContext \* s , TCGOpcode opc ,

const TCGArg \* args , const int \* const\_args )

{

case INDEX\_op\_goto\_tb :

if ( s -> tb\_jmp\_offset ) {

*/\* direct jump method \*/*

tcg\_out8 ( s , OPC\_JMP\_long ) ; */\* jmp im \*/*

*//纪录将来要patch的地方。*

s -> tb\_jmp\_offset [ args [ 0 ] ] = s -> code\_ptr - s -> code\_buf ;

*// jmp的参数为jmp下一个指令与目标的偏移量。*

*//如果还没做block chaining，则jmp 0代表fall through。*

tcg\_out32 ( s , 0 ) ;

} else {

*/\* 在此忽略\*/*

}

*//留待将来"reset" direct jump之用。*

s -> tb\_next\_offset [ args [ 0 ] ] = s -> code\_ptr - s -> code\_buf ;

**break** ;

}

回答上一篇最后留下的问题。在还未patch code cache 中的分支跳转指令的跳转位址，它会fall through，还记​​得jmp 0 吗? 我这边在列出gen\_goto\_tb 的部分内容:

tcg\_gen\_goto\_tb ( tb\_num ) ;

*// Fall through*

*//更新env的eip，使其指向此tb之后欲执行指令的位址。*

*// tb\_find\_fast会用eip查找该TB是否已被翻译过。*

gen\_jmp\_im ( eip ) ;

*//最终回到QEMU tcg\_qemu\_tb\_exec，赋值给next\_tb。*

*//注意! tb\_num会被next\_tb & 3取出，由此可以得知block chaining的方向。*

tcg\_gen\_exit\_tb ( ( tcg\_target\_long ) tb + tb\_num ) ;

目前执行的tb 会赋值给next\_tb (末两位编码block chaining 的方向)。等待下一次回圈，tb\_find\_fast 回传next\_tb 的下一个tb。

if ( next\_tb != 0 && tb -> page\_addr [ 1 ] == - 1 ) {

*//这边利用TranlationBlock指针的最低有效位后两位指引block chaining的方向。*

*// next\_tb -> tb*

tb\_add\_jump ( ( TranslationBlock \* ) ( next\_tb & ~ 3 ) , next\_tb & 3 , tb ) ;

}

That's it! That's how direct block chaining is done in QEMU, I think…

# QEMU Internal – Precise Exception Handling 1/5

Copyright (c) 2012 陈韦任(Chen Wei-Ren)

一般计算机架构会定义当例外/中断发生时，当下的CPU 状态应当为何，这个称之为precise exception。假设当下流水线是底下这样:

A. addl %eax,(%esp)

B. addl %ebx,(%esp)

C. movl %esi,(%ebp)

D. subl %ecx,5

当执行到指令(C)，欲从%ebp 所指的内存位址读取资料至%esi 时，发生页缺失例外。以x86 对precise exception 的定义，在指令(C) 之前的指令，即(A) 和(B) 其结果必须完成。也就是说，暂存器/内存的内容应该更新; 在指令(C) 之后的指令，即(D) 的结果必须舍弃。此范例取自[1]。(注: 对此段汇编的解读，我直接引用[1]。如果单从AT&T 语法来看，指令(C) 应为将%esi 其值写至%ebp 所指位址。)

Precise exception 在binary translation 中占有重要地位。在此以QEMU 为例，说明​​QEMU​​ 如何确保pecise exception。由以上对pecise exception 的说明，我们可以知道precise exception 必须考量暂存器和内存。在binary translation 中，我们关注的是客户(guest) 的precise exception。因此，我们必须确保当guest 代码发生例外时，guest 的暂存器和内存其内容必须满足precise exception 的要求。这样guest 的exception handler 才能正确处理该例外。

就guest 暂存器而言，在QEMU 中需要考量的是CPUState。QEMU 在每一个可能会发生例外的指令或是helper function 之前，会将CPUState 中的大部分内容更新，少数未更新的内容会在guest 真正发生例外时重新再计算。以x86 为例，pc 和condition code 属于后者。在binary translation 中，出于效能上的考量，通常会在一个basic block 的结尾才更新guest pc，而非每翻译一个guest 指令就更新guest pc。

就guest 内存而言，我们将重点放在guest memory store operation 上，因为只有memory store operation 会修改guest 内存的内容。第一，QEMU 会依照guest 原本memory store operation 的顺序进行翻译。第二，针对所有潜在会发生例外的guest 指令，QEMU 保留其相对于guest memory store operation 的顺序。简单来说，QEMU 不会reorder guest 指令顺序。这简化了QEMU 维护precise exception 的复杂度，但同时也丧失了一些潜在可能的优化[2][3]。硬体对异常的处理可能还有别的行为，只要满足precise exception 的规范，同样是可以接受的。在QEMU，我们选择了一个最为保守的实现方式。

接下来以guest 页缺失例外为例，让我们来观察当guest 发生页缺失时，QEMU 是如何维护正确的guest 暂存器和内存。

[1] The Technology Behind Crusoe™ Processor   
[2] http://lists.gnu.org/archive/html/qemu-devel/2012-02/msg04138.html   
[3] http://people.cs.nctu. edu.tw/~chenwj/log/QEMU/agraf-2012-03-02.txt

# QEMU Internal – Precise Exception Handling 2/5

Copyright (c) 2012 陈韦任(Chen Wei-Ren)

这里以linux-0.11 当范例，请至[1][2] 下载代码和硬盘映像编译。我们观察以guest pc 0xe4c0 为开头的basic block。使用QEMU 0.13 [3] 运行linux-0.11。

$ **mkdir** build; **cd** build

$ .. **/** qemu-0.13.0 **/** configure --prefix = $INSTALL --target-list =i386-softmmu \

--enable-debug --extra-cflags = "--save-temps"

$ **make**  **install**

$ **gdb** qemu

**( gdb )** r -boot a -fda Image -hda hdc- 0.11 -new.img -vnc 0.0.0.0: 1 -d in\_asm,op,out\_asm

登入后，下`ls`。观察qemu.log 并定位至0xe4c0。首先，我们可以看到如下内容:

IN:

0x0000e4c0: sub $0x4,%esp

0x0000e4c3: mov 0x8(%esp),%eax

0x0000e4c7: mov %al,(%esp)

0x0000e4ca: movzbl (%esp),%eax

0x0000e4ce: mov 0xc(%esp),%edx

0x0000e4d2: mov %al,%fs:(%edx)

0x0000e4d5: add $0x4,%esp

0x0000e4d8: ret

OP:

---- 0xe4c0

movi\_i32 tmp1,$0x4

mov\_i32 tmp0,esp

sub\_i32 tmp0,tmp0,tmp1

mov\_i32 esp,tmp0

mov\_i32 cc\_src,tmp1

mov\_i32 cc\_dst,tmp0

... 略...

OUT: [size=450]

0x40bbeff0: mov 0x10(%r14),%ebp

0x40bbeff4: sub $0x4,%ebp

... 略...

0x4011813b: callq 0x54d38a

0x40118140: mov 0x10(%r14),%ebp

... 略...

这是QEMU 第一次遇到尚未翻译，以guest pc 0xe4c0 开头的basic block 时所产生的输出，这包括guest binary (IN: 以下内容)、TCG IR (OP: 以下内容) 和host binary (OUT:以下内容)。

再继续往下搜寻0xe4c0，会看到以下内容:

IN:

0x0000e4c0: sub $0x4,%esp

0x0000e4c3: mov 0x8(%esp),%eax

0x0000e4c7: mov %al,(%esp)

0x0000e4ca: movzbl (%esp),%eax

0x0000e4ce: mov 0xc(%esp),%edx

0x0000e4d2: mov %al,%fs:(%edx)

0x0000e4d5: add $0x4,%esp

0x0000e4d8: ret

OP:

---- 0xe4c0

movi\_i32 tmp1,$0x4

mov\_i32 tmp0,esp

sub\_i32 tmp0,tmp0,tmp1

mov\_i32 esp,tmp0

mov\_i32 cc\_src,tmp1

mov\_i32 cc\_dst,tmp0

... 略...

RESTORE:

0x0000: 0000e4c0

0x0007: 0000e4c3

0x000d: 0000e4c7

0x0011: 0000e4ca

0x0015: 0000e4ce

0x001b: 0000e4d2

spc=0x4011813f pc\_pos=0x1b eip=0000e4d2 cs\_bas​​e=0

这里就是重点了。spc 指的是发生例外的host pc，eip 指的是与其相对映发生例外的guest pc。这边请注意，由于我们将guest binary 翻译成host binary 并执行，真正发生例外的是host binary (位于code cache)，但是我们必须将它映射回guest pc，查出哪一​​条guest 指令发生例外，并做后续处理。我们看一下第一次翻译0xe4d2 所得的host binary。

0x4011813b: callq 0x54d38a

0x40118140: mov 0x10(%r14),%ebp

我们可以看到spc 0x4011813f == 0×40118140 – 1，也就是callq 0x54d38a 下一条指令所在位址减去1。这里做点弊，我们在gdb 下print \_\_stb\_mmu。

(gdb) print \_\_stb\_mmu

$1 = {void (target\_ulong, uint8\_t, int)} 0x54d38a

可以得知，我们在呼叫\_\_stb\_mmu 的时候发生例外。\_\_{ld,st}{b,w,l,q}\_{cmmu,mmu} 是用来存取guest 内存的helper function。它们首先会查找TLB (env1->tlb\_table) 试图取得guest virtual address 相对映的host virtual address。如果TLB 命中，可直接利用该host virtual address 存取guest 内存内容。如果TLB 不命中，则会呼叫tlb\_fill (target-i386/op\_helper.c)。tlb\_fill 会呼叫cpu\_x86\_handle\_mmu\_fault 查找客户页表。如果命中，代表该guest virtual address 所在的页已存在，tlb\_fill 会将该页项目填入TLB 以便后续查找。如果不命中，代表发生页缺失，QEMU 会回复guest CPUState，并拉起guest exception index (env->exception\_index) 通知guest 页缺失发生。最后交由guest 页缺失handler 将该页载入。

我先给出一个precise exception handling 的大致流程，之后再透过阅读代码有更深的体会。底下给出关键的资料结构TranslationBlock，它负责掌管guest binary 和host binary 的关系，其中pc 代表此basic block 起始的guest pc，tc\_ptr 指向翻译好的host binary 在code cache 中的位址。

code cache

guest binary TranslationBlock (host binary)

0x0000e4c0: sub $0x4,%esp 0x40bbeff0: mov 0x10(%r14),%ebp

0x0000e4ca: movzbl (%esp),%eax 0x40bbeff4: sub $0x4,%ebp

0x0000e4ce: mov 0xc(%esp),%edx

0x0000e4d2: mov %al,%fs:(%edx) ... 略...

0x0000e4d5: add $0x4,%esp

0x0000e4d8: ret 0x4011813b: callq 0x54d38a

(3) offset --> 0x40118140: mov 0x10(%r14),%ebp (1)

... 略...

当QEMU 发现例外是发生在code cache 里，这代表需要处理precise exception。首先，QEMU 会透过host pc (0x4011813f) 查出相对应的TranslationBlock – (1)。接下来，QEMU 会回复guest CPUState。主要概念是这样，透过之前查找到的TranslationBlock 的pc，我们从该pc 所指的guest binary 重新再翻译一次，同时产生额外的资讯以便回复guest CPUState – (2)。那要如何得知我们已经翻译到发生exception 的guest binary? 这里的重点在于，我们重新翻译guest binary，直到host binary 地址到达了出现异常的位置，这个时候guest pc 就是产生异常的指令。我们便可以回复guest CPUState – (3)。

接下来各小节的主题分别是:

\* Precise Exception Handling 3/5 – (1)

\* Precise Exception Handling 4/5 – (2) (3)

\* Precise Exception Handling 5/5 – 验证我们对QEMU 的理解。

接下来，我们来看代码。;)

[1] http://www.oldlinux.org/oldlinux/viewthread.php?tid=13681&extra=page%3D1   
[2] http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-060625 .zip   
[3] http://wiki.qemu.org/download/qemu-0.13.0.tar.gz

# QEMU Internal – Precise Exception Handling 3/5

Copyright (c) 2012 陈韦任(Chen Wei-Ren)

首先我们来看\_\_stb\_mmu。请看${SRC}/softmmu\_template.h 和${BUILD}/i386-softmmu/op\_helper.i。SOFTMMU 相关的helper function 是透过softmmu\_\* 档案内的巨集加以合成。这里只挑部分加以描述。

SUFFIX 可以是b (byte, 8)、w (word, 16)、l (long word, 32) 和q (quad word, 64)，代表资料大小。MMUSUFFIX 可以是cmmu 或是mmu，分别代表欲读取的是code 或是data。mmu\_idx 代表索引的是内核态亦或是用户态的TLB。addr 代表guest virtual address。

*// ${SRC}/softmmu\_template.h*

void REGPARM glue ( glue ( \_\_st , SUFFIX ) , MMUSUFFIX ) ( target\_ulong addr ,

DATA\_TYPE val ,

int mmu\_idx )

{

...

}

接着看展开巨集后的函式体。

*// ${BUILD}/i386-softmmu/op\_helper.i*

void \_\_stb\_mmu ( target\_ulong addr , uint8\_t val , int mmu\_idx )

{

redo :

*//查找TLB*

tlb\_addr = env -& gt ; tlb\_table [ mmu\_idx ] [ index ] . addr\_write ;

if ( ... ) {

*// TLB 命中*

} else {

*// TLB 不命中*

*/\* the page is not in the TLB : fill it \*/*

*// retaddr = GETPC();*

retaddr = ( ( void \* ) ( ( unsigned long ) \_\_builtin\_return\_address ( 0 ) - 1 ) ) ;

*//试图填入TLB entry。*

tlb\_fill ( addr , 1 , mmu\_idx , retaddr ) ;

goto redo ;

}

}

这里QEMU 利用GCC 的\_\_builtin\_return\_address 扩展[1] 来判定tlb\_fill 是从一般C 函式或是code cache 中被呼叫。retaddr 若为0，表前者，retaddr 若不为0，表后者。之后，我们会透过GDB 更加清楚前面所述所代表的意思。我们关注retaddr 不为0，也就是从code cache 中呼叫tlb\_fill 的情况。

在看tlb\_fill 之前，我们先偷看cpu\_x86\_handle\_mmu\_fault (target-i386/helper.c) 的注解。我们关注返回值为1，也就是页缺失的情况。

*/\* return value:*

*-1 = cannot handle fault*

*0 = nothing more to do*

*1 = generate PF fault*

*\*/*

int cpu\_x86\_handle\_mmu\_fault ( CPUX86State \* env , target\_ulong addr , ... )

{

...

}

我们来看tlb\_fill。

void tlb\_fill ( target\_ulong addr , int is\_write , int mmu\_idx , void \* retaddr )

{

ret = cpu\_x86\_handle\_mmu\_fault ( env , addr , is\_write , mmu\_idx , 1 ) ;

if ( ret ) {

if ( retaddr ) {

*//当客户发生页缺失(ret == 1)且tlb\_fill是从code cache中被*

*//呼叫(retaddr != 0)，我们会在这里。*

*/\* now we have a real cpu fault \*/*

pc = ( unsigned long ) retaddr ;

tb = tb\_find\_pc ( pc ) ;

if ( tb ) {

*/\* the PC is inside the translated code. It means that we have*

*a virtual CPU fault \*/*

cpu\_restore\_state ( tb , env , pc , NULL ) ;

}

}

raise\_exception\_err ( env -& gt ; exception\_index , env -& gt ; error\_code ) ;

}

env = saved\_env ;

}

请注意! 如果retaddr != 0，其值代表的(几乎) 是发生例外的host binary 所在位址。QEMU 利用它来查找是哪一个TranslationBlock 中的host binary 发生例外。tb\_find\_pc (exec.c) 利用该host binary pc 进行查找，取得tb。

TranslationBlock \* tb\_find\_pc ( unsigned long tc\_ptr )

{

*// tbs是TranslationBlock \*数组。每一个在code cache中(已翻译好的)*

*// basic block都有相对应的TranslationBlock存放其相关资讯。*

*/\* binary search (cf Knuth) \*/*

m\_min = 0 ;

m\_max = nb\_tbs - 1 ;

while ( m\_min & gt ; 1 ;

tb = & ; tbs [ m ] ;

*// tc\_ptr代表host binary在code cache的起始位址。*

v = ( unsigned long ) tb -& gt ; tc\_ptr ;

if ( v == tc\_ptr )

return tb ;

else if ( tc\_ptr & lt ; v ) {

m\_max = m - 1 ;

} else {

m\_min = m + 1 ;

}

}

return & amp ; tbs [ m\_max ] ;

}

一但找到该负责的tb，QEMU 就会回复guest CPUState 以便guest exception handler 处理guest 的页缺失例外。

if ( tb ) {

*/\* the PC is inside the translated code. It means that we have*

*a virtual CPU fault \*/*

cpu\_restore\_state ( tb , env , pc , NULL ) ;

}

接着我们看cpu\_restore\_state (translate-all.c)。

[1] http://gcc.gnu.org/onlinedocs/gcc/Return-Address.html

# QEMU Internal – Precise Exception Handling 4/5

Copyright (c) 2012 陈韦任(Chen Wei-Ren)

好! 我们现在找到例外(本范例是页缺失) 是发生在某个TranslationBlock 里头，但是到底是哪一条guest 指令触发页缺失? 我们需要从头翻译该TranslationBlock 对应的guest binary 来揪出罪魁祸首。一般情况下，QEMU 在翻译guest binary 时不会记录guest pc 资讯。这时，为了定位guest pc，QEMU 在翻译guest binary 会记录额外的资讯，包含guest pc。

QEMU 会用到底下定义在translate-all.c 资料结构:

target\_ulong gen\_opc\_pc [ OPC\_BUF\_SIZE ] ; *//纪录guest pc。*

uint8\_t gen\_opc\_instr\_start [ OPC\_BUF\_SIZE ] ; *//当作标记之用。*

针对x86，又在target-i386/translate.c 定义以下资料结构:

static uint8\_t gen\_opc\_cc\_op [ OPC\_BUF\_SIZE ] ; *//纪录condition code。*

现在来看cpu\_restore\_state (translate-all.c)。sea​​rched\_pc 传入的(几乎) 是发生例外的host pc。

int cpu\_restore\_state ( TranslationBlock \* tb ,

CPUState \* env , unsigned long searched\_pc ,

void \* puc )

{

tcg\_func\_start ( s ) ; *//初始gen\_opc\_ptr和gen\_opparam\_ptr*

*//转呼叫gen\_intermediate\_code\_internal，要求在生成TCG IR*

*//的同时，为其生成相关的guest pc和其它资讯于下列资料结构。*

*//*

*// gen\_opc\_pc, gen\_opc\_instr\_start,和gen\_opc\_cc\_op*

*//*

gen\_intermediate\_code\_pc ( env , tb ) ;

*//转呼叫tcg\_gen\_code\_common (tcg/tcg.c)将TCG IR翻成host binary。*

*//返回TCG gen\_opc\_buf index。*

j = tcg\_gen\_code\_search\_pc ( s , ( uint8\_t \* ) tc\_ptr , searched\_pc - tc\_ptr ) ;

*// gen\_opc\_instr\_start[j]若为1，代表gen\_opc\_pc[j]和gen\_opc\_cc\_op[j]*

*//正是我们所要的资讯。*

while ( gen\_opc\_instr\_start [ j ] == 0 )

j --;

*//回复CPUState。*

gen\_pc\_load ( env , tb , searched\_pc , j , puc ) ;

}

gen\_intermediate\_code\_pc 是gen\_intermediate\_code\_internal 的包装，search\_pc 设为1。当search\_pc 为true，在翻译guest binary 的同时，生成额外资讯。

static **inline**  void gen\_intermediate\_code\_internal ( CPUState \* env ,

TranslationBlock \* tb ,

int search\_pc )

{

*// guest binary -> TCG IR*

for ( ;; ) {

if ( search\_pc ) {

*// gen\_opc\_ptr为TCG opcode buffer目前位址，gen\_opc\_buf为*

*// TCG opcode buffer的起始位址。*

j = gen\_opc\_ptr - gen\_opc\_buf ;

if ( lj & lt ; j ) {

lj ++;

while ( lj cc\_op ; *//纪录condition code。*

gen\_opc\_instr\_start [ lj ] = 1 ; *//填1作为标记。*

gen\_opc\_icount [ lj ] = num\_insns ;

}

*//针对pc\_ptr代表的guest pc进行解码并生成TCG IR，返回下一个guest pc。*

pc\_ptr = disas\_insn ( dc , pc\_ptr ) ;

}

}

tcg\_gen\_code\_search\_pc 是tcg\_gen\_code\_common 的包装，search\_pc (应命名为offset) 设为发生例外的host binary 与其所属basic block 在code cache 开头(tc\_ptr) 的offset。注意! 此时传入gen\_code\_buf 的是触发例外的TranslationBlock 其tc\_ptr。也就是说，现在TCG IR -> host binary 中的host binary 是写在发生例外host binary 所属basic block 在code cache 的开头。我们把这段host binary 覆写了! 当然写的内容和被覆写的内容一模一样。我们只想要透过这个方式反推触发例外的guest pc。

static **inline**  int tcg\_gen\_code\_common ( TCGContext \* s , uint8\_t \* gen\_code\_buf ,

long search\_pc )

{

for ( ;; ) {

switch ( opc ) {

case INDEX\_op\_nopn :

args += args [ 0 ] ;

goto next ;

case INDEX\_op\_call :

dead\_args = s -& gt ; op\_dead\_args [ op\_index ] ;

args += tcg\_reg\_alloc\_call ( s , def , opc , args , dead\_args ) ;

goto next ;

}

args += def -& gt ; nb\_args ;

next :

*//如果offset (search\_pc)落在tc\_ptri ( gen\_code\_buf)和code cache*

*//目前存放host binary的位址之间，返回TCG gen\_opc\_buf index。*

if ( search\_pc & gt ;= 0 & amp ;& ; search\_pc < s > code\_ptr - gen\_code\_buf ) {

return op\_index ;

}

op\_index ++;

}

}

此时，gen\_opc\_pc 和gen\_opc\_cc\_op 已存放发生例外的guest pc 和当时的condition code。gen\_pc\_load 负责回复CPUState。

void gen\_pc\_load ( CPUState \* env , TranslationBlock \* tb ,

unsigned long searched\_pc , int pc\_pos , void \* puc )

{

env -& gt ; eip = gen\_opc\_pc [ pc\_pos ] - tb -& gt ; cs\_base ;

cc\_op = gen\_opc\_cc\_op [ pc\_pos ] ;

}

至此，CPUState 已完全回复，我们回来看tlb\_fill。raise\_exception\_err (target-i386/op\_helper.c) 这时候拉起虚拟CPU 的exception\_index (env->exception\_index)，并设置error\_code (env->error\_code)。

void tlb\_fill ( target\_ulong addr , int is\_write , int mmu\_idx , void \* retaddr )

{

ret = cpu\_x86\_handle\_mmu\_fault ( env , addr , is\_write , mmu\_idx , 1 ) ;

if ( ret ) {

if ( retaddr ) {

*//当客户发生页缺失(ret == 1)且tlb\_fill是从code cache中被*

*//呼叫(retaddr != 0)，我们会在这里。*

*/\* now we have a real cpu fault \*/*

pc = ( unsigned long ) retaddr ;

tb = tb\_find\_pc ( pc ) ;

if ( tb ) {

*/\* the PC is inside the translated code. It means that we have*

*a virtual CPU fault \*/*

cpu\_restore\_state ( tb , env , pc , NULL ) ;

}

}

raise\_exception\_err ( env -& gt ; exception\_index , env -& gt ; error\_code ) ;

}

env = saved\_env ;

}

raise\_exception\_err 实际上是raise\_interrupt 的包装(wrapper)。QEMU\_NORETURN 前缀代表此函式不会返回。它其实是GCC 扩展\_\_attribute\_\_ ((\_\_noreturn\_\_))，定义在qemu-common.h [1]。

static void QEMU\_NORETURN raise\_interrupt ( int intno , int is\_int , int error\_code ,

int next\_eip\_addend )

{

... 略...

env -& gt ; exception\_index = intno ;

env -& gt ; error\_code = error\_code ;

env -& gt ; exception\_is\_int = is\_int ;

env -& gt ; exception\_next\_eip = env -& gt ; eip + next\_eip\_addend ;

cpu\_loop\_exit ( ) ;

}

cpu\_loop\_exit (cpu-exec.c) 用longjmp 返回至cpu\_exec (cpu-exec.c) 中处理例外的分支。

void cpu\_loop\_exit ( void )

{

env -& gt ; current\_tb = NULL ;

longjmp ( env -& gt ; jmp\_env , 1 ) ;

}

来看cpu\_exec。cpu\_exec 里用到许多#ifdef，强烈建议查看经过预处理之后结果，即${BUILD}/i386-softmmu/cpu-exec.i 中的cpu\_x86\_exec。

int cpu\_exec ( CPUState \* env )

{

*//进行翻译并执行的回圈。*

*/\* prepare setjmp context for exception handling \*/*

for ( ;; ) {

if ( setjmp ( env -& gt ; jmp\_env ) == 0 ) { *//正常流程。*

*/\* if an exception is pending, we execute it here \*/*

if ( env -& gt ; exception\_index & gt ;= 0 ) {

*/\* 2. 再来到这里，处理例外。\*/*

}

next\_tb = 0 ; */\* force lookup of first TB \*/*

for ( ;; ) {

} */\* inner for(;;) \*/*

}

*/\* 1. 我们先来到这里。\*/*

} */\* outer for(;;) \*/*

}

OK，到这里就是一个循环。:-) 接着，我们来验证一下我们对QEMU 的理解。

[1] http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

# QEMU Internal – Precise Exception Handling 5/5

Copyright (c) 2012 陈韦任(Chen Wei-Ren)

最后，如​​同我之前所承诺的。我们来看tlb\_fill 从一般C 函式和code cache 被呼叫是什么意思。

我们可以看到retaddr == 0 时，tlb\_fill 是从一般C 函式被呼叫。

(gdb) b tlb\_fill

(gdb) r -boot a -fda Image -hda hdc-0.11-new.img -vnc 0.0.0.0:1 -d in\_asm,op,out\_asm

Breakpoint 1, tlb\_fill (addr=4294967280, is\_write=2, mmu\_idx=0, retaddr=0x0) at

/tmp/chenwj/qemu-0.13.0/target-i386/op\_helper.c:4816

4816 {

(gdb) bt

#0 tlb\_fill (addr=4294967280, is\_write=2, mmu\_idx=0, retaddr=0x0) at /tmp/chenwj/qemu-0.13.0/target-i386/op\_helper.c:4816

#1 0x000000000050ee86 in \_\_ldb\_cmmu (addr=4294967280, mmu\_idx=0) at /tmp/chenwj/qemu-0.13.0/softmmu\_template.h:134

#2 0x000000000051045e in ldub\_code (ptr=4294967280) at /tmp/chenwj/qemu-0.13.0/softmmu\_header.h:87

#3 0x000000000051054b in get\_page\_addr\_code (env1=0x110e390, addr=4294967280) at /tmp/chenwj/qemu-0.13.0/exec-all.h:325

#4 0x0000000000510986 in tb\_find\_slow (pc=4294967280, cs\_bas​​e=4294901760, flags=68) at /tmp/chenwj/qemu-0.13.0/cpu-exec.c:139

#5 0x0000000000510b9d in tb\_find\_fast () at /tmp/chenwj/qemu-0.13.0/cpu-exec.c:188

#6 0x00000000005112db in cpu\_x86\_exec (env1=0x110e390) at /tmp/chenwj/qemu-0.13.0/cpu-exec.c:575

#7 0x000000000040aabd in qemu\_cpu\_exec (env=0x110e390) at /tmp/chenwj/qemu-0.13.0/cpus.c:767

#8 0x000000000040abc4 in cpu\_exec\_all () at /tmp/chenwj/qemu-0.13.0/cpus.c:795

#9 0x000000000056e417 in main\_loop () at /tmp/chenwj/qemu-0.13.0/vl.c:1329

#10 0x00000000005721cc in main (argc=11, argv=0x7fffffffe1a8, envp=0x7fffffffe208) at /tmp/chenwj/qemu-0.13.0/vl.c:2992

我们可以看到retaddr != 0 时，tlb\_fill 是从code cache 中被呼叫。

Breakpoint 1, tlb\_fill (addr=28668, is\_write=1, mmu\_idx=0, retaddr=0x4000020c)

at /tmp/chenwj/qemu-0.13.0/target-i386/op\_helper.c:4816

4816 {

(gdb) bt

#0 tlb\_fill (addr=28668, is\_write=1, mmu\_idx=0, retaddr=0x4000020c) at /tmp/chenwj/qemu-0.13.0/target-i386/op\_helper.c:4816

#1 0x000000000054e511 in \_\_stl\_mmu (addr=28668, val=982583, mmu\_idx=0) at /tmp/chenwj/qemu-0.13.0/softmmu\_template.h:272

#2 0x000000004000020d in ?? () <--- 我们在code cache 里!

我们来看一下qemu.log 验证一下我们对QEMU 的了解。;) 既然retaddr = 发生例外的host binary 下一条指令位址减去1，我们定位到0x4000020d。

0x40000208: callq 0x54e3a0

0x4000020d: movzwl %bx,%ebp

瞧瞧\_\_stl\_mmu 的位址，果然是0x54e3a0。这代表我们在code cache 呼叫\_\_stl\_mmu。\_\_stl\_mmu 再去呼叫tlb\_fill 的时候发生例外。

(gdb) p \_\_stl\_mmu

$1 = {void (target\_ulong, uint32\_t, int)} 0x54e3a0

这里我们可以看到SOFTMMU 相关的helper function 在各个地方都会被用到，不论是QEMU 本身的函式(一般C 函式) 或是code cache 都会调用\_\_{ld,st}{b,w,l ,q}\_{cmmu,mmu}。这些helper function 又会调用tlb\_fill。tlb\_fill 就是透过retaddr 来判定是否需要回复guest CPUState。