时间点(time location)问题

参考http://code.google.com/p/vmplayer/wiki/Introduction

Revert相关：

1. 概念

时间点(time location) ：它是在程序运行过程中，指令流的某个具体位置。它是一个动态的概念。

具体而言，在 Revirt 中，时间点描述了一个事件发生的时间，在重放阶段需要在相同的时间点重放这个事件。

在 x86 系统上时间点可以用 3 个属性唯一表示：EIP(指令指针，指令计数器)、BC(分支计数器，又可以称作性能计数器 PC-performance counter)、ECX(ECX 寄存器)。

需要特别说明的是为什么需要 ECX，这是因为存在一些串操作指令，它们结合 rep 前缀可以重复执行直到 ECX 变成 0，并且在重复执行期间 EIP 和 BC 没有变化，只有结合 ECX 才能在指令流中区分串操作指令的每一次执行。

理解时需要注意的问题，在这个记录重放系统中，被监控的对象是一个完整的 guest os，所以我们只需考虑这个 guest os 的指令流对时间点的贡献，而不需要考虑 hypervisor 以及其它guest os(Dom0)对时间点的影响;另外时间点中的分支字段是自启动 RECORD 以来在被监控的guest os 上分支计数器累加的值，即下一个事件记录的分支字段累加了上一个事件记录的分支字段的值。

2. 如何获取时间点

EIP和ECX的获取很简单，可以从 guest os 的用户寄存器组直接读取。下面主要说明分支计数器如何获取。

2.1 关于分支计数器

performance monitor 是用于度量分析程序性能的硬件支持，分支计数(performance counter)是其中一个功能。目前 Intel 和 AMD 都提供了相应支持。performance monitor 在 Intel 不同处理器系列上不兼容，因为原始 revirt 是基于 p4 系列，这里我只关心 P4 系列。在 P4 系列上主要涉及通过 wrmsr 指令对MSR\_TBPU\_ESCR0、MSR\_MS\_CCCR0 和 MSR\_MS\_COUNTER0 寄存器访问。

如：if(!cpu\_isset(pid, counter\_programmed)) {

cpu\_set(pid, counter\_programmed); //避免重复设置

......

/\* Set the ESCR \*/

wrmsrll(0x08001e05, ESCR); /\*ESCR，事件选择\*/

/\* Set the CCCR \*/

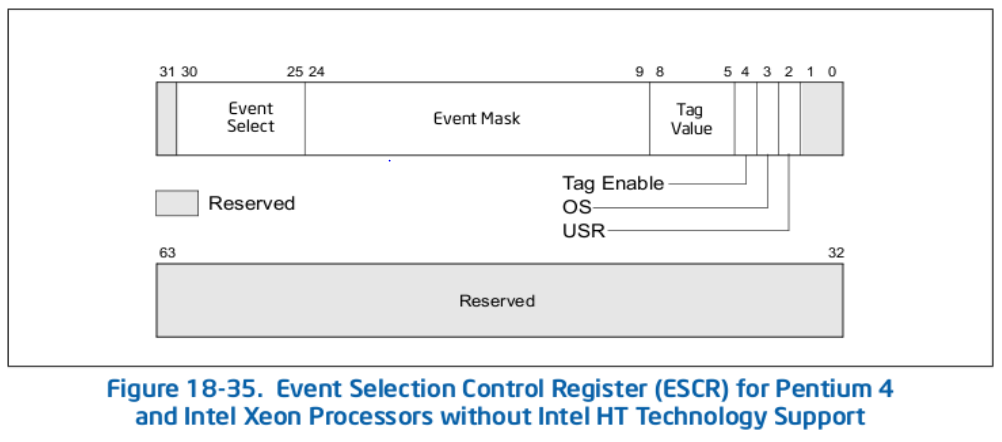
wrmsrll(0x04035000, CCCR); /\*CCCR，ESCR 选择\*/

ESCR：0000 1000 0000 0000 0001 1110 0000 0101

紫色 bit0 是 USR 位，bit2 是 OS 位，所以表示只记录非特权态(ring1-ring3)发生的事件;黑色部分是 Event Select(事件类别)位值为 04H 表示只记录retired\_branch\_type事件；黄色部分是 Event Mask 位（表示一个事件类别中被选择的事件类型）,一共有 4 种类型，分别是黄色部分的 bit1 表示 CONDITIONAL 类型(条件跳转)、bit2 表示 CALL 类型(直接和间接 CALL)、bit3 表示 RETURN 类型(Return 分支)、bit4 表示 INDIRECT 类型Return ,indirect calls , indirect jumps），这里 bit4-bit1=0111，所以不记录 CONDITIONAL 类型（从 0 索引,参考 Intel 文档18.18.6.1 ）。

retired\_branch\_type 不包含异常与中断。

可以参照下图理解（最新Intel手册上的图没有变，为Figure 18-37.）



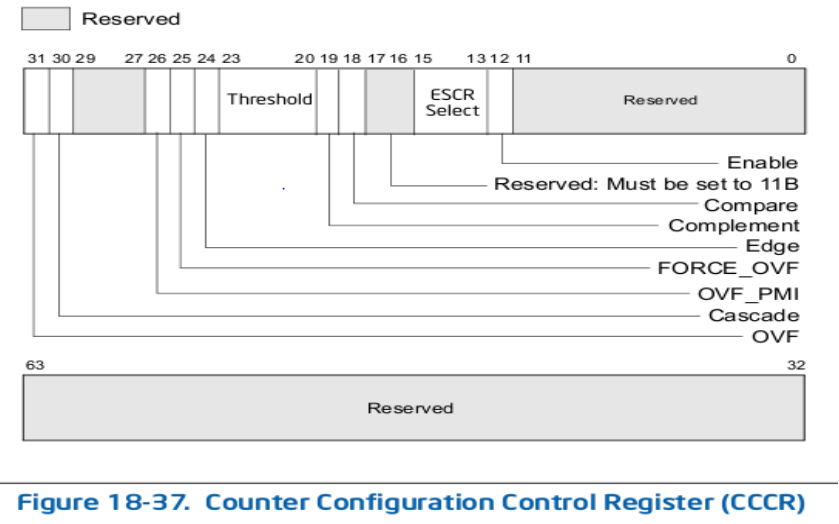
CCCR：0000 0100 0000 0011 01010000 0000 0000

黑色部分是 ESCR Select 位，值为 02H 表示选择 MSR\_TBPU\_ESCR0;黄色是 Enable 位，1 表示启动这个 CCCR 对应性能计数器；绿色位是预留位且必需置 11；紫色是 OVF\_PMI 位 1 表示当对应的计数器 MSR\_MS\_COUNTER0 溢出时启用中断，于是 local APIC 产生一个 PMI 中断，当然首先要设置好 local APIC 的 LVT 的性能计数器项；

这里 CCCR 是 MSR\_MS\_CCCR0 所以对应的是 4 号计数器即 MSR\_MS\_COUNTER0。

注意：当有多个性能计数器设置 OVF\_PMI 位时，PMI 中断服务程序可以通过 CCCR 的 OVF 位判断当前是哪个计数器溢出产生了 PMI；另外当产生性能计数器中断时，local APIC 设置相应 LVT项的 mask 位从而不接收相应中断（mask 位是 LVT 项的一部分）,所以离开中断服务例程前清对应 mask 位。

可以参照下图理解（最新的Intel手册上为Figure 18-39.）



2.3 performance counter 模块逻辑分析

这里描述 performance counter 模块的几个重要函数，因为同一个计数器一直处于计数状，而被监控 dom 只关心在它运行期间计数器累加的值，所以当被监控 dom 要被调入时忽略掉计数器在其它 dom 中累加的值，当 dom 要被调出时把计数器累加的值保存起来以免调出后丢失。

2.3.1 record 初始化阶段

\*xen/arch/x86/perfctr.c\*

void start\_performance\_counters(struct vcpu \*v)

{

long long cur\_pctr\_val;

v->perfctrs.counter\_enabled = 1;

v->perfctrs.break\_enabled=0;

if(v == current) {// 这个分支初始化时执行不到

setup\_performance\_counters\_cpu();

cur\_pctr\_val = read\_perf\_counter();

} else {

ASSERT(!test\_bit(\_VCPUF\_running, &v->vcpu\_flags));

cur\_pctr\_val=0xdeadbeef; /////////////////////////////

}

v->perfctrs.consumed = PERFCTR\_INIT; //被监控 dom 时间点分支从 1 开始

v->perfctrs.sched\_start = cur\_pctr\_val;/\*最开始就是 0xdeadbeef,在这里设置的这个

值的初值应该不影响结果，因为 update\_count 会更新这个值\*/

}

这个函数在 record 初始化阶段被调用，它不触发真正的硬件设置，只是为 schedule 阶段的硬件设置做好准备。

v 是被设置 dom 的某个 vcpu，在 domain create 阶段和 record init 阶段调用这个函数,v 都不会是当前 vcpu，所以真正的硬件设置工作在这两个阶段没有做,真正的硬件设置工作在调度阶段才去做。

另外 dom 创建阶段也会调用 start\_performance\_counters，完成功能和这里一样，它同样会使得在 schedule 阶段为性能计数器进行真正的硬件设置，但是只有进行 record 初始化后才开始收集事件记录，这时才关心分支计数器的值。

2.3.2 调度阶段(schedule)

\* xen/arch/x86/perfctr.c \*

void do\_perfctrs\_scheduler(struct vcpu \*from, struct vcpu \*to)

{/\*from 是 current(就是 dom0 的某个 vcpu-现在假设系统中只有两个 dom-dom0 和一个

guest)，to 是被调入的 dom 的某个 vcpu

\*/

if (from == to)

BUG();

/\* Make sure performance counters have been set up on this CPU properly \*/

if(to->perfctrs.counter\_enabled) {//在 Record 初始化阶段已经设置被监控 dom 的 vcpu

的 perfctrs.counter\_enabled 字段为 1，所以这里会真实的执行硬件设置。

setup\_performance\_counters\_cpu(); //设置硬件

setup\_perfctr\_interrupt(); //设置对应的中断处理函数

}

update\_count(from,to); // schedule 阶段一个重要功能是调用 update\_count 函数。

......

}

\*xen/arch/x86/perfctr.c\*

static inline void update\_count(struct vcpu \* from, struct vcpu \* to) {

/\*在 do\_perfctrs\_scheduler 调用时,to 是当前被监控 dom,from 是 Dom0 或者 idle dom但是当被 sync\_count 调用而 sync\_count 被 get\_pcounter\_val 调用时，from 是当前 dom,to 是NULL\*/

long long pct, pct\_start;

int flags;

......

pct = read\_perf\_counter();

//硬件计数器上当前值

//core2 与 p4 关于计数器的概念不变

if(pct > 0x1000000000LL) { /\*如果计数器值 pct 已经达到较大值，把它设置回 1,这是

个相当大的数但不是计数器表示最大数\*/

pct\_start = 1; //

#ifndef REPLAY\_HT\_ENABLE //这条分支,如果计数器的值已经比较大，则设置为 1

wrmsrll(IA32\_PERFEVTSEL\_VAL3, IA32\_PERFEVTSEL0);

wrmsrll(pct\_start, COUNTER);

wrmsrll(IA32\_PERFEVTSEL\_VAL, IA32\_PERFEVTSEL0);

#else

wrmsrll(pct\_start, COUNTER\_HT[smp\_processor\_id()%smp\_num\_siblings]);

#endif

} else {

//否则不需要重新设置计数器，pct\_start 取计数器当前值。

pct\_start = pct;

}

if(!is\_idle\_domain(from->domain)) {

from->perfctrs.consumed += pct – from->perfctrs.sched\_start;/\*cousumed 字

段是直接供其它模块读取消费的，从这里可以看出分支计数字段记录的是基于 1 开始的累加值，

而不是绝对值,并且 from->perfctrs.consumed 是个不断累加的过程\*/

from->perfctrs.sched\_start = pct\_start; /\*重新设置 sched\_start。\*/

}

if(to && !is\_idle\_domain(to->domain)) //

to->perfctrs.sched\_start = pct\_start; /\*忽略其它 dom 为计数器贡献的值\*/

spin\_unlock\_irqrestore(&update\_count\_lock, flags);

}

这个函数在 do\_perfctrs\_scheduler 被调用，另外#define sync\_count(\_v)

update\_count(\_v,NULL)导致调用 sync\_count 就是调用 update\_count(\_v,NULL)，所以真正直接调用的地方只有 do\_perfctrs\_scheduler。

schedule 阶段会不断调用这个函数，当一个 dom 要被调出时(from 将要被调出)通过

from->perfctrs.consumed 把这阶段计数器的值累加保存起来，并且通过from->perfctrs.sched\_start = pct\_start 更新下次累加的起点为当前计数器值。

当一个 dom 要被调入(to)时忽略掉其它 dom 为计数器贡献的值

to->perfctrs.sched\_start = pct\_start;

2.3.3 接口函数 get\_pcounter\_val

\*xen/arch/x86/perfctr.c\*

unsigned long long get\_pcounter\_val(struct vcpu \*v)

{

if (v == current) {

sync\_count(v); /\*读之前要同步一下被监控 dom 的值 #define sync\_count(\_v)

update\_count(\_v,NULL)\*/

} else {

assert\_vcpu\_not\_running(v);

}

return v->perfctrs.consumed; //

}

虽然性能技术器模块有很多琐碎的内容，但是其它模块或得服务时很简单，只需调用

get\_pcounter\_val 就可以或得当前分支计数器的值。

以上是性能技术器模块的几个关键函数分析，还有其它函数不再做详细分析，它们的主要代码都在 xen/arch/x86/perfctr.c 中。

关于最新版本xen中设置代码：

Op\_model\_p4.c (xen\arch\x86\oprofile) 18654 2013/11/18

struct p4\_event\_binding {

int escr\_select; /\* value to put in CCCR \*/

int event\_select; /\* value to put in ESCR \*/

struct {

int virt\_counter; /\* for this counter... \*/

int escr\_address; /\* use this ESCR \*/

} bindings[2];

};

struct op\_counter\_config {

unsigned long count; //放入计数器的数值

unsigned long enabled;

unsigned long event;

unsigned long kernel;

unsigned long user;

unsigned long unit\_mask;

};

static const struct p4\_event\_binding p4\_events[NUM\_EVENTS] = {

{ /\* BRANCH\_RETIRED \*/

0x05, 0x06,

{ {CTR\_IQ\_4, MSR\_P4\_CRU\_ESCR2},

{CTR\_IQ\_5, MSR\_P4\_CRU\_ESCR3} }

},

……………………………………………………..

}

static void pmc\_setup\_one\_p4\_counter(unsigned int ctr){

int i;

int const maxbind = 2;

uint64\_t cccr = 0;

uint64\_t escr = 0;

unsigned int counter\_bit;

const struct p4\_event\_binding \*ev = NULL;

unsigned int stag;

stag = get\_stagger();

……………………………………………………

counter\_bit = 1 << VIRT\_CTR(stag, ctr);

ESCR\_WRITE(escr, ev, i);

/\* modify CCCR \*/

…………………………………………….

CCCR\_WRITE(cccr, VIRT\_CTR(stag, ctr));

return;

}

}

}

static void p4\_setup\_ctrs(struct op\_msrs const \* const msrs)

{

………………………………………………………….

/\* setup all counters \*/

for (i = 0 ; i < num\_counters ; ++i) {

if (counter\_config[i].enabled) {

reset\_value[i] = counter\_config[i].count;

pmc\_setup\_one\_p4\_counter(i);

CTR\_WRITE(counter\_config[i].count, VIRT\_CTR(stag, i));

} else {

reset\_value[i] = 0;

}

}

}

static int p4\_check\_ctrs(unsigned int const cpu,

struct op\_msrs const \* const msrs,

struct cpu\_user\_regs \* const regs)

{

unsigned long ctr, stag, real;

uint64\_t msr\_content;

int i;

int ovf = 0;

unsigned long eip = regs->eip;

int mode = xenoprofile\_get\_mode(current, regs);

stag = get\_stagger();

for (i = 0; i < num\_counters; ++i) {

if (!reset\_value[i])

continue;

real = VIRT\_CTR(stag, i);

CCCR\_READ(msr\_content, real);

CTR\_READ(ctr, real);

if (CCCR\_OVF\_P(msr\_content) || CTR\_OVERFLOW\_P(ctr)) {

xenoprof\_log\_event(current, regs, eip, mode, i);

CTR\_WRITE(reset\_value[i], real);

CCCR\_CLEAR\_OVF(msr\_content);

CCCR\_WRITE(msr\_content, real);

CTR\_WRITE(reset\_value[i], real);

ovf = 1;

}

}

/\* P4 quirk: you have to re-unmask the apic vector \*/

apic\_write(APIC\_LVTPC, apic\_read(APIC\_LVTPC) & ~APIC\_LVT\_MASKED);

return ovf;

}

注：在Intel手册里关于设置计数器overflow引发中断：

18.12.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter over-flows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel®VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR\_PMI flag in the counter’s associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 18.12.5.6, “Cascading Counters.” Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead ofwaiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE\_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.