Simulate Invalid Opcode 的性能代价分析

前言

正象在《Simulate Invalid Opcode》一文中看到的，功能可以 simulate,但损失的是性能。在本文中想分析一下由模拟而损失的性能。我在此假设乘方运算指令 0XF089C8 (lock movl %eax, %ecx)如果由 CPU 来实现的话，它所花费的时钟周期与乘法运算指令 imul %eax, %ecx 是相当的（我想我的这个假设应该是合适的，因为乘法与除法指令是相当费时的指令）

前提条件

为了分析性能，我们必须统计执行某个动作花了多少时间，由于无论是执行 imul 指令还是模拟执行乘方运算指令，都是电光火石的时间，库函数提供的计时函数一般只能精确到毫秒级,有些能达微秒级，但现在的 CPU 速度太快，在一微秒之内已发生了太多事情。所以微秒级对我们的统计而言，粒度太粗。我们需要精确的纳秒级的计时器。

奔腾 4 CPU 的 RDTSC (ReaD Time Stamp Counter)指令可以给我们上述精度。

RDTSC 在 EDX:EAX 中记录了自 CPU 上电以来所经过的时钟周期数（CPU time stamp）

```
inline void get_time_stamp(unsigned int *time_high, unsigned int *time_low)
{

    /* ReaD Time Stamp Counter instruction */
    __asm__ __volatile__("rdtsc\n\t"
    /* The return value is 64-bit value in edx:eax register pair */
    "movl %%edx, %0\n\t"
    "movl %%eax, %1\n\t"
    :"=m"(*time_high), "=m"(*time_low)
    :
    :"%eax", "%edx"
    );
}
```

上面的函数返回一个 64 位的长整型，记录 CPU 的时间戳，高 32 位放入 time_high，低 32 位放入 time_low.如果你已经进入 64 位时代(比如用的是 AMD64)，那可能不需要这么麻烦。

```
get_time_stamp(&time_before_high, &time_before_low);

do something

get_time_stamp(&time_after_high, &time_after_low);
```

这时的 time 值就记录了做 do something 所花费的时间（多少时钟周期数），要把它转换成多少时间，只要除以你的 CPU 的主频即可。比如，CPU 主频是 1.8G, time 为 1000，则时间为 1000 / 1.8G = 1000 / 1.8e+9 = 1000 / 18000000000 = 555.6 纳秒

我们还是分 2 种情况测试

用户态的 application

测试代码如下

$cat invalid_op.c

```
1  #include <stdio.h>         //for printf
2  #include <sys/mman.h>  //for mlock & munlock system call
3  #include <assert.h>        //for assert
4
5
6  inline void get_time_stamp(unsigned int *time_high, unsigned int *time_low)
7  {
8
9          /* ReaD Time Stamp Counter instruction */
10         __asm__ __volatile__("rdtsc\n\t"
11         /* The return value is 64-bit value in edx:eax register pair */
12         "movl %%edx, %0\n\t"
13         "movl %%eax, %1\n\t"
14         :"=m"(*time_high), "=m"(*time_low)
15         :
16         :"%eax", "%edx"
17         );
18 }
19
20 inline unsigned long long make_64_bit_value(unsigned int high_32, unsigned int low_32)
21 {
22         /* make the 64-bit unsigned long long */
```

```
23          unsigned long long value_64 = 0;
24          value_64 = high_32;
25          value_64 <<= 32;
26          value_64 |= low_32;
27
28          return      value_64;
29  }
30
31  inline void involution_3_3(void)
32  {
33          /* calc 3 ** 3 */
34          __asm__("movl %0, %%eax\n\t"
35          "movl %1, %%ecx\n\t"
36          "lock movl %%ecx, %%eax\n\t"
37          :
38          :"i"(3), "i"(3)
39          );
40  }
41
42  int main()
43  {
44          unsigned int addr;
45          __asm__ __volatile__("call 1f\n\t"
46          "1:pop %%eax\n\t"
47          "movl %%eax, %0\n\t"
48          :"=r"(addr)
49          :
50          :"%eax"
51          );
52
53          void *paddr = (void*)addr;
54          assert(0 != mlock(paddr, 0x2000));
55
56          unsigned int time_before_high, time_before_low;
57          unsigned int time_after_high, time_after_low;
58
59          get_time_stamp(&time_before_high, &time_before_low);
60
61          /* produce 10 times invalid_opcode exception */
62          involution_3_3();   //1
63          involution_3_3();   //2
64          involution_3_3();   //3
```

```
65          involution_3_3();   //4
66          involution_3_3();   //5
67          involution_3_3();   //6
68          involution_3_3();   //7
69          involution_3_3();   //8
70          involution_3_3();   //9
71          involution_3_3();   //10
72
73          get_time_stamp(&time_after_high, &time_after_low);
74
75          assert(0 != munlock(paddr, 0x2000));
76
77          unsigned long long time_after = make_64_bit_value(time_after_high, time_after_low);
78          unsigned long long time_before = make_64_bit_value(time_before_high,
time_before_low);
79          unsigned int time_spend = time_after - time_before;
80
81          printf("time: %ld\n", time_spend);
82
83          return      0;
84  }
```

involution_3_3()是我要测试的乘方运算 (3 ** 3),我测 10 次 3 ** 3 的乘方运算，即发生 10 次 invalid opcode exception 所花费的时间。

```
44          unsigned int addr;
45          __asm__ __volatile__("call 1f\n\t"
46          "1:pop %%eax\n\t"
47          "movl %%eax, %0\n\t"
48          :"=r"(addr)
49          :
50          :"%eax"
51          );
52
53          void *paddr = (void*)addr;
54          assert(0 != mlock(paddr, 0x2000));
```

上面 line 44 到 line 54 是为了防止被测试的代码，也即是 line 59 到 line 73, 产生 page fault, 从而影响测试结果，所以把被测试代码 lock 在内存。当然如果吹毛求疵的话，似乎应该在执行被测试代码时禁止进程切换(schedule),但在用户态我实在想不出有什么办法做到，而在核心态实现起来又太脏(通过临时接管 timer interrupt,或许你知道有漂亮的做法，请告诉我)，所以我忽略这种可能性。

make_64_bit_value()是把两个 32 位值合成一个 64 位值,我的 gcc 不支持 -mlong64 选项，所以只能手工完成。

核心态 LKM 测试

cat invalid.c

```
1  #include <linux/module.h>
2  #include <linux/moduleparam.h>
3  #include <linux/kernel.h>
4  #include <linux/init.h>
5  #include <linux/stat.h>
6
7  MODULE_LICENSE("GPL");
8  MODULE_AUTHOR("Walter Zhou");
9
10 inline void get_time_stamp(unsigned int *time_high, unsigned int *time_low)
11 {
12
13         /* ReaD Time Stamp Counter instruction */
14         __asm__ __volatile__("rdtsc\n\t"
15         /* The return value is 64-bit value in edx:eax register pair */
16         "movl %%edx, %0\n\t"
17         "movl %%eax, %1\n\t"
18         :"=m"(*time_high), "=m"(*time_low)
19         :
20         :"%eax", "%edx"
21         );
22 }
23
24 inline unsigned long long make_64_bit_value(unsigned int high_32, unsigned int low_32)
25 {
26         /* make the 64-bit unsigned long long */
27         unsigned long long value_64 = 0;
28         value_64 = high_32;
29         value_64 <<= 32;
30         value_64 |= low_32;
31
32         return      value_64;
33 }
34
35 inline void involution_3_3(void)
36 {
37         /* calc 3 ** 3 */
38         __asm__("movl %0, %%eax\n\t"
```

```
39          "movl %1, %%ecx\n\t"
40          "lock movl %%ecx, %%eax\n\t"
41          :
42          :"i"(3), "i"(3)
43          );
44  }
45
46  static int __init invalid_op_init(void)
47  {
48          unsigned int time_before_high, time_before_low;
49          unsigned int time_after_high, time_after_low;
50
51          get_time_stamp(&time_before_high, &time_before_low);
52
53          involution_3_3();   //1
54          involution_3_3();   //2
55          involution_3_3();   //3
56          involution_3_3();   //4
57          involution_3_3();   //5
58          involution_3_3();   //6
59          involution_3_3();   //7
60          involution_3_3();   //8
61          involution_3_3();   //9
62          involution_3_3();   //10
63
64          get_time_stamp(&time_after_high, &time_after_low);
65
66          unsigned long long time_after = make_64_bit_value(time_after_high, time_after_low);
67          unsigned long long time_before = make_64_bit_value(time_before_high,
time_before_low);
68          unsigned int time_spend = time_after - time_before;
69
70          printk(KERN_ALERT "Time: %ld\n", time_spend);
71          return 0;
72  }
73
74
75  static void __exit invalid_op_exit(void)
76  {
77          printk(KERN_ALERT "exit invalid_op module\n");
78  }
79
80  module_init(invalid_op_init);
```

```
81    module_exit(invalid_op_exit);
```

代码同用户态程序大同小异，只不过用户态的一些担心不需要了。

1. 由于是 kernel mode 代码，自然是 lock 在内存的
2. 我运行在自己编译的非抢占式内核上，所以在运行被测试代码时不会被切换

有了模拟指令的执行测试，自然要有相应的对比

对比测试

```
 1  #include <stdio.h>         //for printf
 2  #include <sys/mman.h> //for mlock & munlock system call
 3  #include <assert.h>        //for assert
 4
 5  inline void get_time_stamp(unsigned int *time_high, unsigned int *time_low)
 6  {
 7
 8          /* ReaD Time Stamp Counter instruction */
 9          __asm__ __volatile__("rdtsc\n\t"
10          /* The return value is 64-bit value in edx:eax register pair */
11          "movl %%edx, %0\n\t"
12          "movl %%eax, %1\n\t"
13          :"=m"(*time_high), "=m"(*time_low)
14          :
15          :"%eax", "%edx"
16          );
17  }
18
19  inline unsigned long long make_64_bit_value(unsigned int high_32, unsigned int
low_32)
20  {
21          /* make the 64-bit unsigned long long */
22          unsigned long long value_64 = 0;
23          value_64 = high_32;
24          value_64 <<= 32;
25          value_64 |= low_32;
26
27          return      value_64;
28  }
29
```

```
30  inline void mul_3_3(void)
31  {
32          /* calc 3 ** 3 */
33          __asm__("movl %0, %%eax\n\t"
34          "movl %1, %%ecx\n\t"
35          "imul %%eax, %%ecx\n\t"
36          :
37          :"i"(3), "i"(3)
38          :"%eax", "%ecx"
39          );
40  }
41
42  int main()
43  {
44          unsigned int time_before_high, time_before_low;
45          unsigned int time_after_high, time_after_low;
46
47          unsigned int addr;
48          __asm__ __volatile__("call 1f\n\t"
49          "1:pop %%eax\n\t"
50          "movl %%eax, %0\n\t"
51          :"=r"(addr)
52          :
53          :"%eax"
54          );
55
56          void *paddr = (void*)addr;
57          assert(0 != mlock(paddr, 0x2000));
58
59          get_time_stamp(&time_before_high, &time_before_low);
60
61          mul_3_3();      //1
62          mul_3_3();      //2
63          mul_3_3();      //3
64          mul_3_3();      //4
65          mul_3_3();      //5
66          mul_3_3();      //6
67          mul_3_3();      //7
68          mul_3_3();      //8
69          mul_3_3();      //9
70          mul_3_3();      //10
71
```

```
72          get_time_stamp(&time_after_high, &time_after_low);
73
74          assert(0 != munlock(paddr, 0x2000));
75
76          unsigned long long time_after = make_64_bit_value(time_after_high,
time_after_low);
77          unsigned long long time_before = make_64_bit_value(time_before_high,
time_before_low);
78          unsigned int time_spend = time_after - time_before;
79
80          printf("time: %ld\n", time_spend);
81          return      0;
82      }
```

大部分代码同用户态的测试代码雷同，区别是不再是用模拟乘方指令的 lock movl %ecx, %eax，而是合法的乘法指令 imul %eax, %ecx.同样执行 10 次。

测试结果

	User App	LKM	Imul
1	19536	21988	112
2	19676	19424	112
3	19936	19880	112
4	19632	18424	112
5	19844	19356	112
6	19824	21492	112
7	18688	19532	112
8	19416	21196	112
9	18884	19892	124
10	19440	19916	124
执行 10 次的平均值	19487	20110	114.4
执行单次的平均值	1949	2011	11.5
纳秒数	1082.9	1117.3	6.389

本来由于一个是乘法，一个是乘方，实在不是可以等而论之的，但我这里主要分析一下模拟指令的代价，并不是要精确的分析某功能的指令如用硬件执行是多少时间，而等价的功能用软件模拟要多少时间。从上表可看出，代价是极其巨大的，差不多是近 **200** 倍的代价。

疑问

LKM 测试的时间竟然比 User App 要长，这是出乎我意料的。理论上 LKM 的测试应该比 User App 略短，因为在同样发生 Trap 6 的情况下，User App 比 LKM 多了一个从 User Mode 切

换到Kernel Mode，处理完后还要从Kernel Mode切换回User Mode. 但从测试结果反而LKM所花的时间略多于User App. 不解 ？

附录

上面的代码请用第3级优化编译 （-O3）

User App 的被测试代码的汇编如下

```
204  80484cc:  0f 31              rdtsc
205  80484ce:  89 55 ec           mov    %edx,0xffffffec(%ebp)
206  80484d1:  89 45 e8           mov    %eax,0xffffffe8(%ebp)
207  80484d4:  b8 03 00 00 00     mov    $0x3,%eax
208  80484d9:  b9 03 00 00 00     mov    $0x3,%ecx
209  80484de:  f0 89 c8           lock    mov %ecx,%eax
210  80484e1:  b8 03 00 00 00     mov    $0x3,%eax
211  80484e6:  b9 03 00 00 00     mov    $0x3,%ecx
212  80484eb:  f0 89 c8           lock    mov %ecx,%eax
213  80484ee:  b8 03 00 00 00     mov    $0x3,%eax
214  80484f3:  b9 03 00 00 00     mov    $0x3,%ecx
215  80484f8:  f0 89 c8           lock    mov %ecx,%eax
216  80484fb:  b8 03 00 00 00     mov    $0x3,%eax
217  8048500:  b9 03 00 00 00     mov    $0x3,%ecx
218  8048505:  f0 89 c8           lock    mov %ecx,%eax
219  8048508:  b8 03 00 00 00     mov    $0x3,%eax
220  804850d:  b9 03 00 00 00     mov    $0x3,%ecx
221  8048512:  f0 89 c8           lock    mov %ecx,%eax
222  8048515:  b8 03 00 00 00     mov    $0x3,%eax
223  804851a:  b9 03 00 00 00     mov    $0x3,%ecx
224  804851f:  f0 89 c8           lock    mov %ecx,%eax
225  8048522:  b8 03 00 00 00     mov    $0x3,%eax
226  8048527:  b9 03 00 00 00     mov    $0x3,%ecx
227  804852c:  f0 89 c8           lock    mov %ecx,%eax
228  804852f:  b8 03 00 00 00     mov    $0x3,%eax
229  8048534:  b9 03 00 00 00     mov    $0x3,%ecx
230  8048539:  f0 89 c8           lock    mov %ecx,%eax
231  804853c:  b8 03 00 00 00     mov    $0x3,%eax
232  8048541:  b9 03 00 00 00     mov    $0x3,%ecx
233  8048546:  f0 89 c8           lock    mov %ecx,%eax
234  8048549:  b8 03 00 00 00     mov    $0x3,%eax
235  804854e:  b9 03 00 00 00     mov    $0x3,%ecx
236  8048553:  f0 89 c8           lock    mov %ecx,%eax
237  8048556:  0f 31              rdtsc
```

```
238    8048558:    89 55 e4              mov    %edx,0xfffffe4(%ebp)
239    804855b:    89 45 e0              mov    %eax,0xfffffe0(%ebp)
```

LKM 的被测试代码的汇编如下

```
64     9:    0f 31                 rdtsc
65     b:    89 55 f0              mov    %edx,0xffffff0(%ebp)
66     e:    89 45 ec              mov    %eax,0xfffffec(%ebp)
67     11:   b8 03 00 00 00        mov    $0x3,%eax
68     16:   b9 03 00 00 00        mov    $0x3,%ecx
69     1b:   f0 89 c8              lock    mov %ecx,%eax
70     1e:   b8 03 00 00 00        mov    $0x3,%eax
71     23:   b9 03 00 00 00        mov    $0x3,%ecx
72     28:   f0 89 c8              lock    mov %ecx,%eax
73     2b:   b8 03 00 00 00        mov    $0x3,%eax
74     30:   b9 03 00 00 00        mov    $0x3,%ecx
75     35:   f0 89 c8              lock    mov %ecx,%eax
76     38:   b8 03 00 00 00        mov    $0x3,%eax
77     3d:   b9 03 00 00 00        mov    $0x3,%ecx
78     42:   f0 89 c8              lock    mov %ecx,%eax
79     45:   b8 03 00 00 00        mov    $0x3,%eax
80     4a:   b9 03 00 00 00        mov    $0x3,%ecx
81     4f:   f0 89 c8              lock    mov %ecx,%eax
82     52:   b8 03 00 00 00        mov    $0x3,%eax
83     57:   b9 03 00 00 00        mov    $0x3,%ecx
84     5c:   f0 89 c8              lock    mov %ecx,%eax
85     5f:   b8 03 00 00 00        mov    $0x3,%eax
86     64:   b9 03 00 00 00        mov    $0x3,%ecx
87     69:   f0 89 c8              lock    mov %ecx,%eax
88     6c:   b8 03 00 00 00        mov    $0x3,%eax
89     71:   b9 03 00 00 00        mov    $0x3,%ecx
90     76:   f0 89 c8              lock    mov %ecx,%eax
91     79:   b8 03 00 00 00        mov    $0x3,%eax
92     7e:   b9 03 00 00 00        mov    $0x3,%ecx
93     83:   f0 89 c8              lock    mov %ecx,%eax
94     86:   b8 03 00 00 00        mov    $0x3,%eax
95     8b:   b9 03 00 00 00        mov    $0x3,%ecx
96     90:   f0 89 c8              lock    mov %ecx,%eax
97     93:   0f 31                 rdtsc
98     95:   89 55 e8              mov    %edx,0xfffffe8(%ebp)
99     98:   89 45 e4              mov    %eax,0xfffffe4(%ebp)
```

Imul 被测试代码的汇编如下

```
204    80484cc:    0f 31              rdtsc
205    80484ce:    89 55 ec           mov    %edx,0xffffffec(%ebp)
206    80484d1:    89 45 e8           mov    %eax,0xffffffe8(%ebp)
207    80484d4:    b8 03 00 00 00     mov    $0x3,%eax
208    80484d9:    b9 03 00 00 00     mov    $0x3,%ecx
209    80484de:    0f af c8           imul   %eax,%ecx
210    80484e1:    b8 03 00 00 00     mov    $0x3,%eax
211    80484e6:    b9 03 00 00 00     mov    $0x3,%ecx
212    80484eb:    0f af c8           imul   %eax,%ecx
213    80484ee:    b8 03 00 00 00     mov    $0x3,%eax
214    80484f3:    b9 03 00 00 00     mov    $0x3,%ecx
215    80484f8:    0f af c8           imul   %eax,%ecx
216    80484fb:    b8 03 00 00 00     mov    $0x3,%eax
217    8048500:    b9 03 00 00 00     mov    $0x3,%ecx
218    8048505:    0f af c8           imul   %eax,%ecx
219    8048508:    b8 03 00 00 00     mov    $0x3,%eax
220    804850d:    b9 03 00 00 00     mov    $0x3,%ecx
221    8048512:    0f af c8           imul   %eax,%ecx
222    8048515:    b8 03 00 00 00     mov    $0x3,%eax
223    804851a:    b9 03 00 00 00     mov    $0x3,%ecx
224    804851f:    0f af c8           imul   %eax,%ecx
225    8048522:    b8 03 00 00 00     mov    $0x3,%eax
226    8048527:    b9 03 00 00 00     mov    $0x3,%ecx
227    804852c:    0f af c8           imul   %eax,%ecx
228    804852f:    b8 03 00 00 00     mov    $0x3,%eax
229    8048534:    b9 03 00 00 00     mov    $0x3,%ecx
230    8048539:    0f af c8           imul   %eax,%ecx
231    804853c:    b8 03 00 00 00     mov    $0x3,%eax
232    8048541:    b9 03 00 00 00     mov    $0x3,%ecx
233    8048546:    0f af c8           imul   %eax,%ecx
234    8048549:    b8 03 00 00 00     mov    $0x3,%eax
235    804854e:    b9 03 00 00 00     mov    $0x3,%ecx
236    8048553:    0f af c8           imul   %eax,%ecx
237    8048556:    0f 31              rdtsc
238    8048558:    89 55 e4           mov    %edx,0xffffffe4(%ebp)
239    804855b:    89 45 e0           mov    %eax,0xffffffe0(%ebp)
```

由上可知即使在 machine code 级别，被测试代码也极其相近。该测试还是有一定说服力的。

本测试在我的塞扬 1.8 G, 最新 Linux Kernel 2.6.20 下编译并测试。

如有指教请发 mail 给我。

zhoulong@sh163.net