如何让NEMU跑得更快

内存篇

14307130078

张博洋

衡量性能的标准

- 据我观察，当在游戏中调出“物品栏”时，游戏的运行速度最慢。因此以此情况下的每秒执行指令数作为衡量NEMU性能的标准。

- 大家写完PA4之后的第一反应肯定是——慢
- 哪里慢？——内存

Terminal 终端 - zby@macbookair: ~/src/fdu-ics/programming-assignment

文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)

```
Samples: 472K of event 'cycles', Event count (approx.): 199887966645
 79.44%  nemu  nemu                  [.] ddr3_read
  5.35%  nemu  nemu                  [.] dram_read
  2.32%  nemu  nemu                  [.] hwaddr_read
  2.00%  nemu  nemu                  [.] is_mmio
  1.93%  nemu  nemu                  [.] page_translate_nocache
  1.80%  nemu  nemu                  [.] ddr3_write
  1.71%  nemu  nemu                  [.] swaddr_read_slow
  0.50%  nemu  nemu                  [.] decode_r2rm_l
  0.48%  nemu  nemu                  [.] read_ModR_M_l
  0.46%  nemu  nemu                  [.] seg_translate
  0.30%  nemu  nemu                  [.] cpu_exec
  0.28%  nemu  nemu                  [.] exec
  0.25%  nemu  [kernel.kallsyms]     [k] 0xffffffff8105144a
  0.21%  nemu  nemu                  [.] load_addr
  0.20%  nemu  nemu                  [.] mov_r2rm_v
  0.20%  nemu  nemu                  [.] swaddr_read
  0.17%  nemu  nemu                  [.] dram_write
  0.12%  nemu  libSDL-1.2.so.0.11.4  [.] 0x00000000000119e4
  0.11%  nemu  nemu                  [.] group1_sx_v
  0.10%  nemu  nemu                  [.] do_update_screen_graphic_mode
  0.10%  nemu  nemu                  [.] swaddr_write_slow
  0.09%  nemu  nemu                  [.] mov_rm2r_v
  0.09%  nemu  nemu                  [.] cmp_r2rm_v
  0.09%  nemu  nemu                  [.] add_r2rm_v
  0.08%  nemu  nemu                  [.] push_r_v
Press '?' for help on key bindings
```

初始速度

- 关闭DEBUG
- 关闭TLB
- 关闭Cache
- -O2
- 关闭PIE

- 0.8952 MIPS　后面的速度均以此作为基准

```
[1467708198.145457622] speed: 885531.18 instrs/sec, 0.00 hlts/sec, jitter = 1986
00.37 ms (1986003.73%), eip = PAL_RLEBlitToSurface+0074
[1467708273.022076945] speed: 897085.88 instrs/sec, 0.00 hlts/sec, jitter = 2531
85.05 ms (2531850.50%), eip = PAL_DrawCharOnSurface+0042
[1467708347.986776147] speed: 896387.77 instrs/sec, 0.00 hlts/sec, jitter = 3076
36.79 ms (3076367.86%), eip = PAL_RLEBlitToSurface+00D6
[1467708422.530221008] speed: 900988.41 instrs/sec, 0.00 hlts/sec, jitter = 3618
83.49 ms (3618834.92%), eip = PAL_RLEBlitToSurface+00DA
[1467708497.327920335] speed: 898390.24 instrs/sec, 0.00 hlts/sec, jitter = 4161
52.37 ms (4161523.66%), eip = PAL_RLEBlitToSurface+00B0
```

回顾一下内存相关的函数调用流程

swaddr_read()

段转换：seg_translate()

lnaddr_read()

页转换：page_translate()

TLB（已关闭）

hwaddr_read()

L1 Cache, L2 Cache（已关闭）

dram_read()
ddr3_read()

```
79.44%  [.] ddr3_read
 5.35%  [.] dram_read
```

- **dram_read(), ddr3_read() 很慢，怎么优化？**
- **这两个函数是用来模拟DRAM的工作方式的。**

```
/* Simulate the (main) behavor of DRAM.
 * Although this will lower the performace of NEMU, it makes
 * you clear about how DRAM perform read/write operations.
 * Note that cross addressing is not simulated.
 */
```

- **实际上，dram_read(addr, len)的实际作用只是把位于hw_mem+addr处的len个字节读出来**

- **dram_write()同理**

改进的dram_read(), dram_write()

```c
#define HW_MEM_SIZE (128 * 1048576)
uint8_t fast_mem[HW_MEM_SIZE];
uint8_t *hw_mem = fast_mem;

uint32_t dram_read(hwaddr_t addr, size_t len)
{
    return (*(uint32_t *)(hw_mem + addr)) & ((1LL << (len << 3)) - 1);
}

void dram_write(hwaddr_t addr, size_t len, uint32_t data)
{
    if (len == 4) {
        *(uint32_t *)(hw_mem + addr) = data;
    } else if (len == 2) {
        *(uint16_t *)(hw_mem + addr) = (uint16_t) data;
    } else {
        assert(len == 1);
        *(uint8_t *)(hw_mem + addr) = (uint8_t) data;
    }
}

void init_ddr3()
{
    // do nothing
}
```

- 效果立竿见影
- **6.162 MIPS (6.88x)**

```
[1467709216.725212988] speed: 6156069.98 instrs/sec, 0.00 hlts/sec, jitter = 2.6
9 ms (26.92%), eip = PAL_RLEBlitToSurface+00B8
[1467709227.627364006] speed: 6161122.65 instrs/sec, 0.00 hlts/sec, jitter = 2.7
0 ms (26.99%), eip = PAL_DrawCharOnSurface+00B4
[1467709238.539393670] speed: 6155700.11 instrs/sec, 0.00 hlts/sec, jitter = 2.6
2 ms (26.19%), eip = PAL_RLEBlitToSurface+007C
[1467709249.402685877] speed: 6182790.18 instrs/sec, 0.00 hlts/sec, jitter = 2.6
7 ms (26.72%), eip = PAL_RLEBlitToSurface+00A7
[1467709260.282862406] speed: 6173449.93 instrs/sec, 0.00 hlts/sec, jitter = 2.6
1 ms (26.13%), eip = PAL_RLEBlitToSurface+00A2
```

能不能更快？

- 先看 perf 结果再决定优化什么。
- 时间耗费比较平均
- 需要仔细深入思考了

Terminal 终端 - zby@macbookair: ~/src/fdu-ics/programming-assignment

文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)

```
Samples: 94K of event 'cycles', Event count (approx.): 39748414959
 16.10%  nemu  nemu                    [.] hwaddr_read
 13.70%  nemu  nemu                    [.] is_mmio
 12.92%  nemu  nemu                    [.] page_translate_nocache
 12.59%  nemu  nemu                    [.] dram_read
 11.00%  nemu  nemu                    [.] swaddr_read_slow
  3.59%  nemu  nemu                    [.] read_ModR_M_l
  3.09%  nemu  nemu                    [.] seg_translate
  2.16%  nemu  nemu                    [.] exec
  2.15%  nemu  nemu                    [.] cpu_exec
  1.82%  nemu  nemu                    [.] load_addr
  1.41%  nemu  nemu                    [.] swaddr_read
  1.40%  nemu  nemu                    [.] mov_r2rm_v
  1.02%  nemu  nemu                    [.] decode_r2rm_l
  0.73%  nemu  nemu                    [.] test_r2rm_v
  0.69%  nemu  nemu                    [.] group1_sx_v
  0.66%  nemu  [kernel.kallsyms]       [k] 0xffffffff8105144a
  0.65%  nemu  nemu                    [.] mov_rm2r_v
  0.65%  nemu  nemu                    [.] swaddr_write_slow
  0.59%  nemu  libSDL-1.2.so.0.11.4    [.] 0x00000000000192c2
  0.57%  nemu  nemu                    [.] push_r_v
  0.53%  nemu  nemu                    [.] add_r2rm_v
  0.50%  nemu  nemu                    [.] cmp_r2rm_v
  0.46%  nemu  nemu                    [.] decode_r_l
Press '?' for help on key bindings
```

怎么办

- 原来的代码为什么慢？

- 因为在实际读写数据前，有一大堆额外工作。

- “段转换”
- “页转换”
- “内存映射I/O”的判断

哪个是大头？

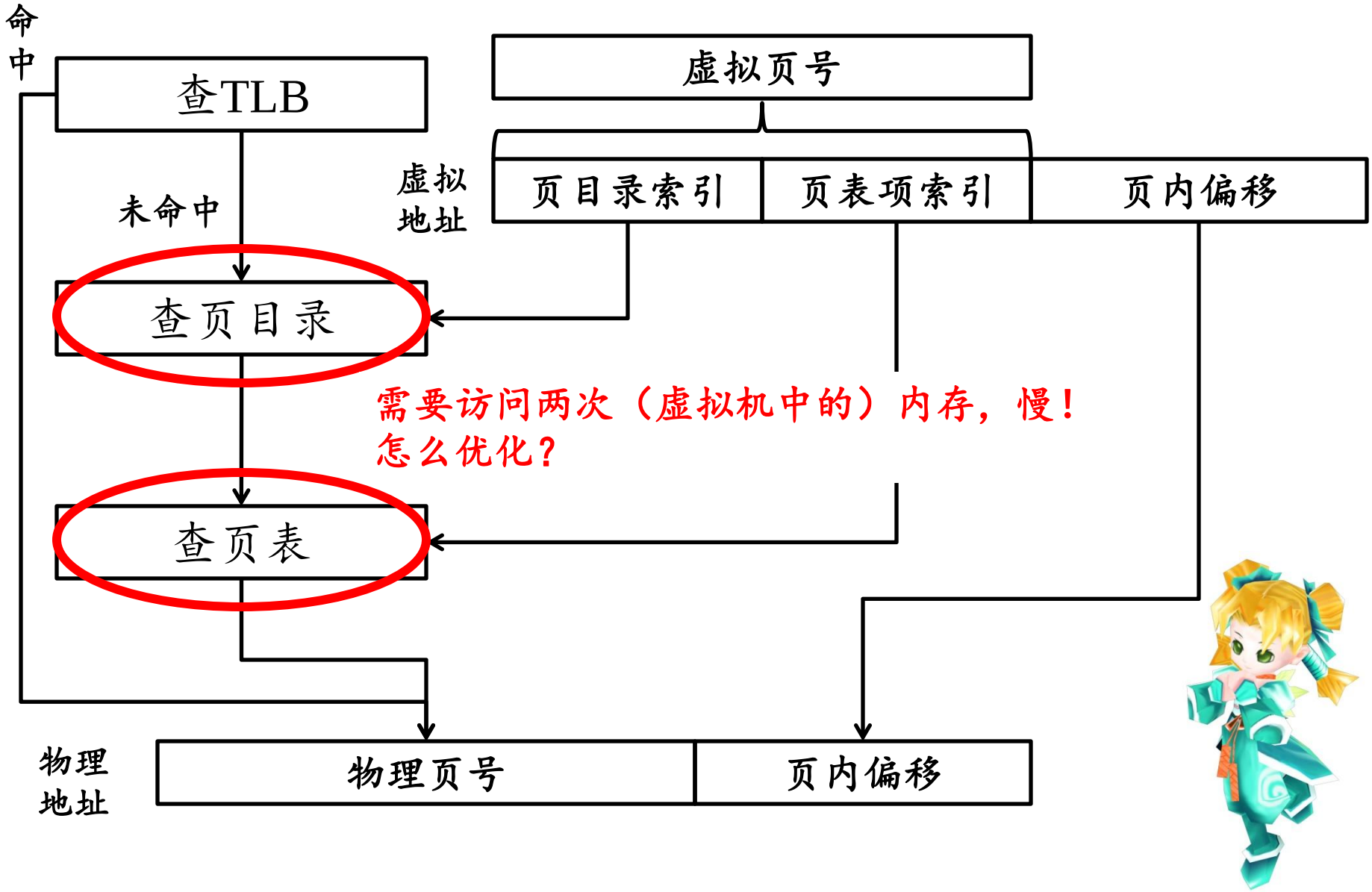
- perf 得出的结果很重要
- 但是也不能完全看哪个慢就去“大干快上”

```
Samples: 94K of event 'cycles', Event count (approx.): 39748414959
 16.10%  nemu  nemu                  [.] hwaddr_read
 13.70%  nemu  nemu                  [.] is_mmio
 12.92%  nemu  nemu                  [.] page_translate_nocache
 12.59%  nemu  nemu                  [.] dram_read
 11.00%  nemu  nemu                  [.] swaddr_read_slow
```

- 实际上这里慢的是“页转换”，为什么？
- 因为在页转换过程中，会多次访问内存。
- 这样hwaddr_read()的调用次数会增加很多。

- 怎么优化？

先回顾一下页转换

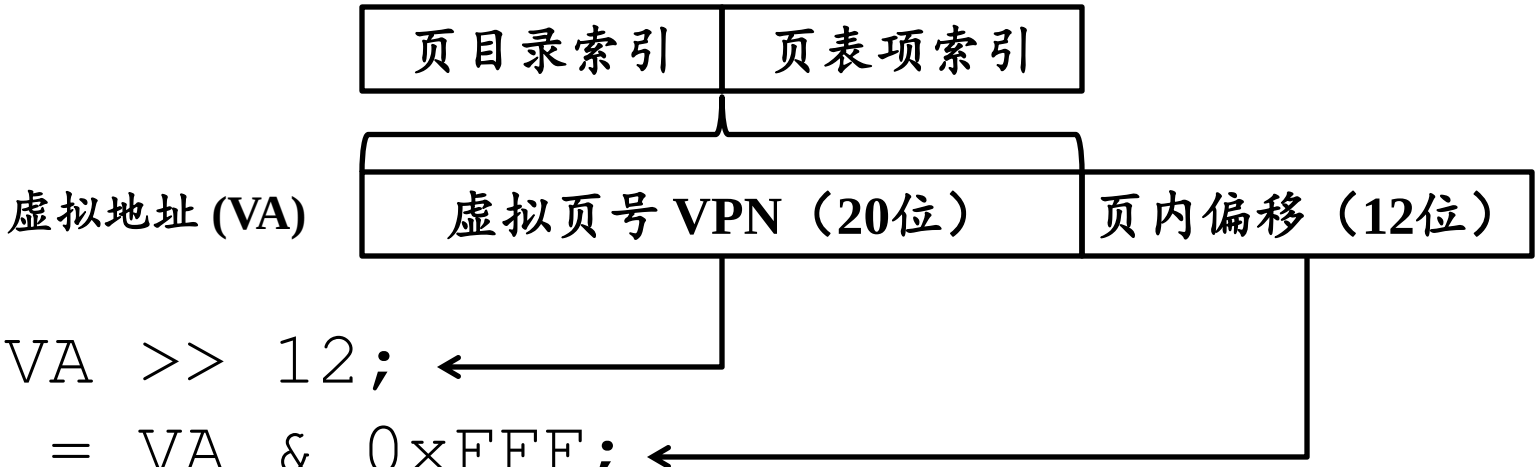

命中
查TLB
虚拟页号
虚拟地址
页目录索引
页表项索引
页内偏移
未命中
查页目录
需要访问两次（虚拟机中的）内存，慢！
怎么优化？
查页表
物理地址
物理页号
页内偏移

用TLB

- TLB本质上是一种缓存
- 把每次地址翻译的结果存储起来
- 下次遇到时直接取之前翻译好的结果

- 缓存开多大比较好?
- 64项？
- No! 2^{20}=1M项，为什么？
- 因为虚页号只有20位
- 直接开一个一一对应的数组！

```
uint32_t myTLB[1<<20];
```




```
VPN = VA >> 12;
OFFSET = VA & 0xFFF;

if (TLB_VALID(myTLB[VPN])) {
    result = myTLB[VPN] | OFFSET;
} else {
    result = page_translate(VA);
    myTLB[VPN] = result & ~0xFFF;
    SET_TLB_VALID(VPN);
}
```

- 此时的TLB已经不像个“缓存”
- 而像一个“页表”了

```
static unsigned char myTLBvalid[1 << 20];
static unsigned myTLB[1 << 20];
hwaddr_t page_translate(lnaddr_t addr)
{
    unsigned vpn = addr >> 12;
    if (myTLBvalid[vpn]) {
        return myTLB[vpn] | (addr & 0xfff);
    } else {
        unsigned result = page_translate_nocache(addr);
        myTLB[vpn] = result & ~0xfff;
        myTLBvalid[vpn] = 1;
        return result;
    }
}
void flush_tlb()
{
    memset(myTLBvalid, 0, sizeof(myTLBvalid));
}
```

- **12.83 MIPS**
- **比较：**
- **　　刚才：6.162 MIPS (2.08x)**
- **　　最初：0.8952 MIPS (14.3x)**

```
[1471795140.105632974] speed: 12852610.37 instrs/sec, 0.00 hlts/sec, jitter = 1.
28 ms (12.82%), eip = PAL_RLEBlitToSurface+00BE
[1471795145.308090802] speed: 12912016.71 instrs/sec, 0.00 hlts/sec, jitter = 1.
29 ms (12.87%), eip = PAL_RLEBlitToSurface+00AE
[1471795150.548505949] speed: 12819901.50 instrs/sec, 0.00 hlts/sec, jitter = 1.
25 ms (12.54%), eip = PAL_RLEBlitToSurface+00B6
[1471795155.781381571] speed: 12837674.04 instrs/sec, 0.00 hlts/sec, jitter = 1.
28 ms (12.82%), eip = PAL_RLEBlitToSurface+009F
[1471795161.043609463] speed: 12766666.62 instrs/sec, 0.00 hlts/sec, jitter = 1.
31 ms (13.14%), eip = PAL RLEBlitToSurface+00B6
```

能不能更快？

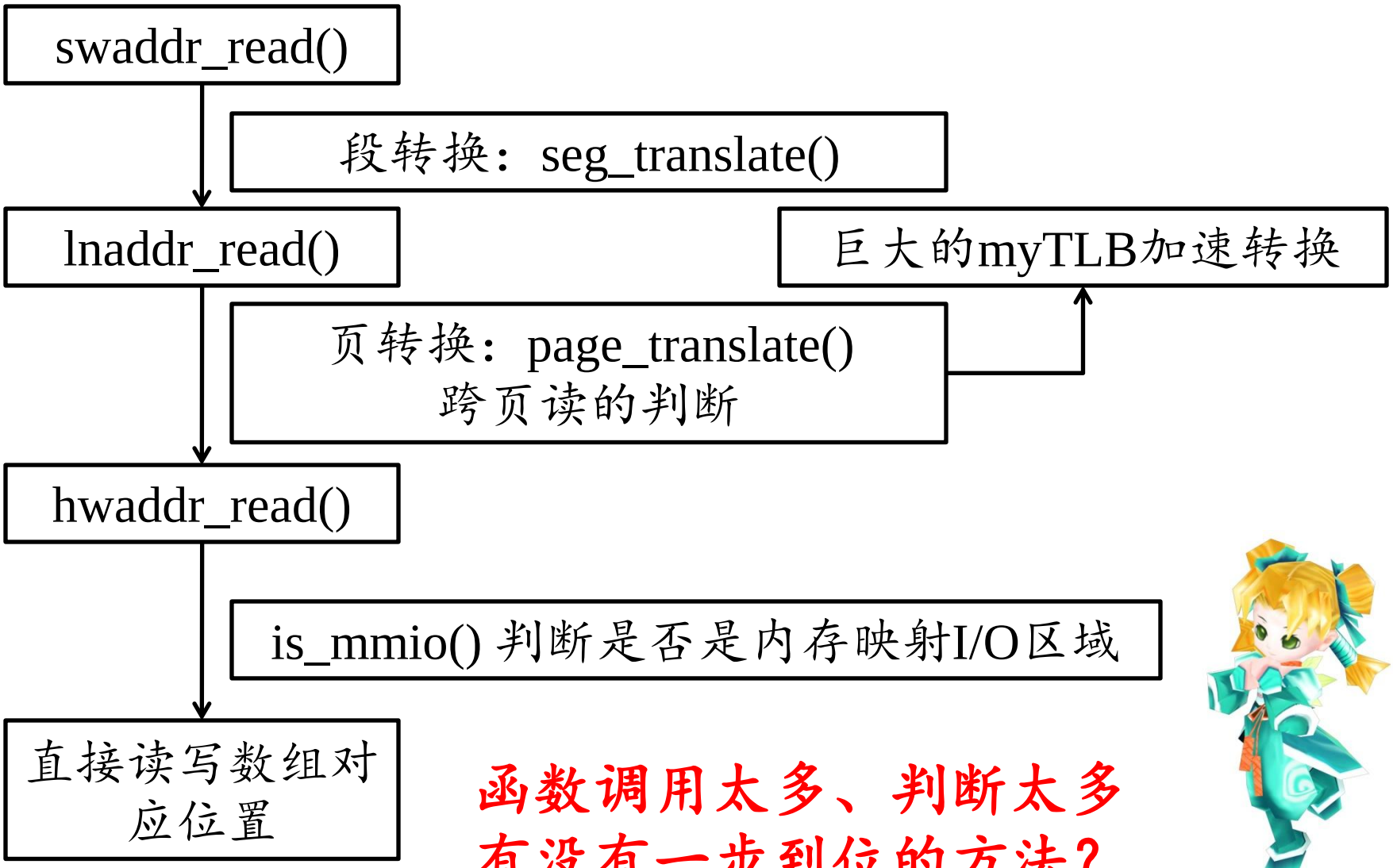
- swaddr_read() 等内存相关的函数还是很慢
- 主要是因为读内存的步骤很繁琐（各种转换、判断）
- 有没有办法优化呢？

Terminal 终端 - zby@macbookair: ~/src/fdu-ics/programming-assignment

文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)

```
Samples: 2M of event 'cycles', Event count (approx.): 1003499309600
 21.52%  nemu  nemu                [.] swaddr_read_slow
 10.40%  nemu  nemu                [.] page_translate
  9.21%  nemu  nemu                [.] dram_read
  8.57%  nemu  nemu                [.] is_mmio
  6.65%  nemu  nemu                [.] seg_translate
  6.31%  nemu  nemu                [.] load_addr
  5.33%  nemu  nemu                [.] read_ModR_M_l
  3.45%  nemu  nemu                [.] cpu_exec
  2.96%  nemu  nemu                [.] exec
  2.60%  nemu  nemu                [.] swaddr_read
  1.78%  nemu  nemu                [.] add_rm2r_v
  1.71%  nemu  nemu                [.] mov_rm2r_v
  1.43%  nemu  nemu                [.] decode_rm2r_l
  1.14%  nemu  nemu                [.] decode_r2rm_l
  1.13%  nemu  nemu                [.] read_ModR_M_b
  1.09%  nemu  nemu                [.] cmp_r2rm_v
  1.08%  nemu  nemu                [.] cmp_rm2r_v
  1.06%  nemu  nemu                [.] _2byte_esc
  0.98%  nemu  nemu                [.] inc_r_v
  0.90%  nemu  nemu                [.] decode_i_b
  0.81%  nemu  nemu                [.] mov_r2rm_v
  0.77%  nemu  nemu                [.] mov_rm2r_b
Press '?' for help on key bindings
```

回顾一下优化后的函数调用流程



函数调用太多、判断太多
有没有一步到位的方法？

简化流程

- 段转换——反正是平坦模式，直接忽略

- 跨页读的判断——如果跨越的两个虚页，在物理上是连续的话，则无需判断。而大多数情况下，跨页读都发生在连续的物理页上。有没有办法利用这样一个特点？
- 内存映射I/O的判断——只有极少数页面对应的是内存映射I/O区域。有没有办法利用这样一个特点？
- 有没有办法把页转换与这些判断整合起来？

一步到位

- 注意到myTLB[VPN]存储的是PPN<<12

myTLB[VPN]	物理页号 PPN（20位）	12位0

- 低12位全部是0，可以利用起来
- 因此规定:
- 若低12位为0，则表示本虚页和后一个虚页
- (1)在物理上连续，无需跨页读判断
- (2)都不是内存映射I/O区域，无需is_mmio()
- 若低12位为非0，则表示需要进一步处理。

伪代码

```
uint32_t myTLB2[1<<20];
uint32_t swaddr_read(swaddr_t addr, size_t len, sreg)
{
    uint32_t result = myTLB2[addr>>12];
    if ((result & 0xFFF) == 0) {
        return dram_read(result | (addr & 0xFFF), len);
    } else {
        return swaddr_read_miss(addr, len, sreg);
    }
}
```

大多数情况下，都会从红色处返回。
少数情况下，会调用swaddr_read_miss()进行进一步判断。

swaddr_write()也做类似的操作。

伪代码

```
uint32_t swaddr_read_miss(swaddr_t addr, size_t len, sreg)
{
    uint32_t pa1 = safe_page_translate(addr);
    uint32_t pa2 = safe_page_translate(addr + 0x1000);
    int pa1_mmio_id = is_mmio(pa1);
    if (pa1_mmio_id < 0) {
        if (is_mmio(pa2) < 0 && (pa2>>12) == (pa1>>12)+1) {
            myTLB2[addr >> 12] = pa1 & ~0xFFF;
        }
        return hwaddr_read(pa1, len);
    } else {
        return mmio_read(pa1, len, pa1_mmio_id);
    }
}
void flush_myTLB2()
{
    memset(myTLB2, -1, sizeof(myTLB2));
}
```

何时需要flush_TLB2()?

(1)最初初始化时

(2) CR3变化时

- **24.09 MIPS**
- **比较：**
- **　　刚才：12.83 MIPS (1.88x)**
- **　　最初：0.8952 MIPS (26.9x)**

- **此时游戏基本可以流畅运行了。**

```
[1471855162.559108405] speed: 25791133.25 instrs/sec, 0.00 hlts/sec, jitter = 0.
81 ms (8.06%), eip = PAL_RLEBlitToSurface+00D0
[1471855165.155083903] speed: 25992529.39 instrs/sec, 0.00 hlts/sec, jitter = 0.
77 ms (7.66%), eip = PAL_RLEBlitToSurface+00D9
[1471855167.730772361] speed: 26087610.14 instrs/sec, 0.00 hlts/sec, jitter = 0.
66 ms (6.65%), eip = PAL_RLEBlitToSurface+00D0
[1471855170.299594583] speed: 26157446.25 instrs/sec, 0.00 hlts/sec, jitter = 0.
62 ms (6.22%), eip = PAL_DrawCharOnSurface+00B3
[1471855172.842510004] speed: 26422924.35 instrs/sec, 0.00 hlts/sec, jitter = 0.
65 ms (6.55%), eip = PAL_RLEBlitToSurface+00BB
```

能不能更快？

一些优化技巧

- likely() 和 unlikely() 宏

- 用来告诉编译器，分支条件更偏向于成立/不成立
- #define likely(x)　　__builtin_expect((x),1)
- #define unlikely(x)　　__builtin_expect((x),0)

- if (unlikely(发生错误的条件)) {
- 　　// 处理错误的代码
- }
- 将条件套入unlikely()中
- 告诉编译器错误很少发生
- 这样编译器能够利用此信息产生更高效的代码

用在myTLB2[]
的低12位判断上！

```
int test(int v) {
    a(); b(); c();
    if (unlikely(v == 0)) { x(); y(); z(); }
    d(); e(); f();
}
```

举例

加了unlikely()

```
push   %ebx
sub    $0x8,%esp
mov    0x10(%esp),%ebx
call   80483f0 <a>
call   8048400 <b>
call   8048410 <c>
test   %ebx,%ebx
je     80484b3 <test+0x33>
nop
lea    0x0(%esi,%eiz,1),%esi
call   8048420 <d>
call   8048430 <e>
add    $0x8,%esp
pop    %ebx
jmp    8048440 <f>
call   8048450 <x>
call   8048460 <y>
lea    0x0(%esi),%esi
call   8048470 <z>
jmp    804849b <test+0x1b>
```

x,y,z被放到函数最后

没加unlikely()

```
push   %ebx
sub    $0x8,%esp
mov    0x10(%esp),%ebx
call   80483f0 <a>
call   8048400 <b>
call   8048410 <c>
test   %ebx,%ebx
jne    80484af <test+0x2f>
nop
lea    0x0(%esi,%eiz,1),%esi
call   8048450 <x>
call   8048460 <y>
call   8048470 <z>
nop
call   8048420 <d>
call   8048430 <e>
add    $0x8,%esp
pop    %ebx
jmp    8048440 <f>
```

一些优化技巧

- 内联函数 (inline)

- 优点:
- (1)函数内联可以减少函数调用的开销
- (2)还可以做进一步的优化

- 缺点:
- (1)一般会导致代码大小增大
- (2)过度使用内联可能反而使得性能降低

假设我们有这样一个函数：

```
inline void test(int x)
{
    if (x == 1) printf("a");
    if (x == 2) printf("b");
    if (x == 3) printf("c");
}
```

当这样调用时：

```
test(3);
```

swaddr_read()的len参数同理！

若不进行内联：

会有一次函数调用开销

和三次判断的开销

若进行内联+编译优化：

编译器会知道test()中x==3

这样就省去if判断语句了

阻碍内联优化的情况

- 一般情况下，若函数的实现在另一个文件中，则无法进行内联

- 设源码 a.c 含有函数a()，源码 b.c 含有函数b()
- b()函数调用了a()函数，现在想把a()内联进入b()

- 回忆一下编译连接的过程：
- gcc -O2 -c a.c => 生成a.o
- gcc -O2 -c b.c => 生成b.o
- gcc -o test a.o b.o
  => 将a.o b.o连接为可执行文件test

阻碍内联优化的情况

- 原因在于，不同.c源码是分开编译的，各自独立
- gcc -O2 -c a.c
- gcc -O2 -c b.c

- 编译b.c的时候，编译器并不知道a()函数的实现在哪里，更无法对其进行内联优化。

- 一般的做法是：把实现放在.h文件中
- 这样编译每个.c时，编译器都能看到实现

- 但是不想改代码了，有没有简单的办法？

LTO 链接时优化 (link time optimization)

- 解决办法就是改进最后一步——链接！
- 在链接时，链接器能看到所有的代码，可以进行优化。
  - gcc 叫这种优化“链接时优化(LTO)”
    - MSVC叫这种优化“全程序优化”
- 但是编译的步骤需要一些改动：
- gcc -flto -O2 -c a.c
- gcc -flto -O2 -c b.c
- gcc -flto -O2 -o test a.o b.o

- LTO会显著增加链接器的运行时间（最后一步）

修改Makefile以启用LTO

- **nemu/Makefile.part**

```
nemu_CFLAGS_EXTRA := -flto -O2 -ggdb3 -Ilib-common

nemu_LDFLAGS := -flto -O2 -lreadline -lSDL


$(nemu_BIN): $(nemu_OBJS)
	$(call make_command, $(CC), $(nemu_LDFLAGS), ld $@, $^)
```

- **40.45 MIPS**
- **比较：**
- **　刚才：24.09 MIPS (1.68x)**
- **　最初：0.8952 MIPS (45.2x)**

```
[1471860431.611093807] speed: 40606576.07 instrs/sec, 0.00 hlts/sec, jitter = 0
39 ms (3.88%), eip = PAL_DrawCharOnSurface+00A3
[1471860433.278317351] speed: 40304728.36 instrs/sec, 0.00 hlts/sec, jitter = 0
41 ms (4.07%), eip = PAL_DrawCharOnSurface+00A1
[1471860434.926922201] speed: 40757679.32 instrs/sec, 0.00 hlts/sec, jitter = 0
40 ms (4.04%), eip = PAL_RLEBlitToSurface+00D0
[1471860436.594185051] speed: 40303614.89 instrs/sec, 0.00 hlts/sec, jitter = 0
36 ms (3.61%), eip = PAL_RLEBlitToSurface+00C4
[1471860438.261302178] speed: 40308117.56 instrs/sec, 0.00 hlts/sec, jitter = 0
41 ms (4.13%), eip = PAL_RLEBlitToSurface+00D3
```

但是

能不能再快一点？

- 随着速度的提高，已经很难再现有代码的基础上提速了，需要另辟蹊径。

- 以exec()函数为例，从汇编上看，其代码已经很精简。

```
                                make_helper(exec) {
                                  ops_decoded.opcode = instr_fetch(eip, 1);
                                  return opcode_table[ops_decoded.opcode](eip);
mov     %edi,%eax               }
push    %rbx
mov     %edi,%ebx
shr     $0xc,%eax  //取EIP的虚页号
mov     0xb3d2480(,%rax,4),%eax // 查myTLB2表
test    $0xfff,%eax  // 判断低12位是否为0
jne     4077cc <exec+0x3c> // 若非零则跳转(很少跳转)
mov     %edi,%edx
and     $0xfff,%edx //取EIP的页内偏移
or      %edx,%eax //或起来
movzbl  0x33807a0(%rax),%eax //读内存
mov     %eax,0xb3cacc3(%rip) //存入ops_decoded.opcode
mov     %ebx,%edi
mov     %eax,%eax
pop     %rbx
mov     0x4195a0(,%rax,8),%rax // 查函数指针表
jmpq    *%rax // 直接跳转!
mov     $0x1,%edx // 调用swaddr_read_miss读取内存
mov     $0x1,%esi
callq   4016e0 <swaddr_read_miss>
jmp     4077b7 <exec+0x27>
nopl    (%rax)
```

很难再按照原来的办法优化!

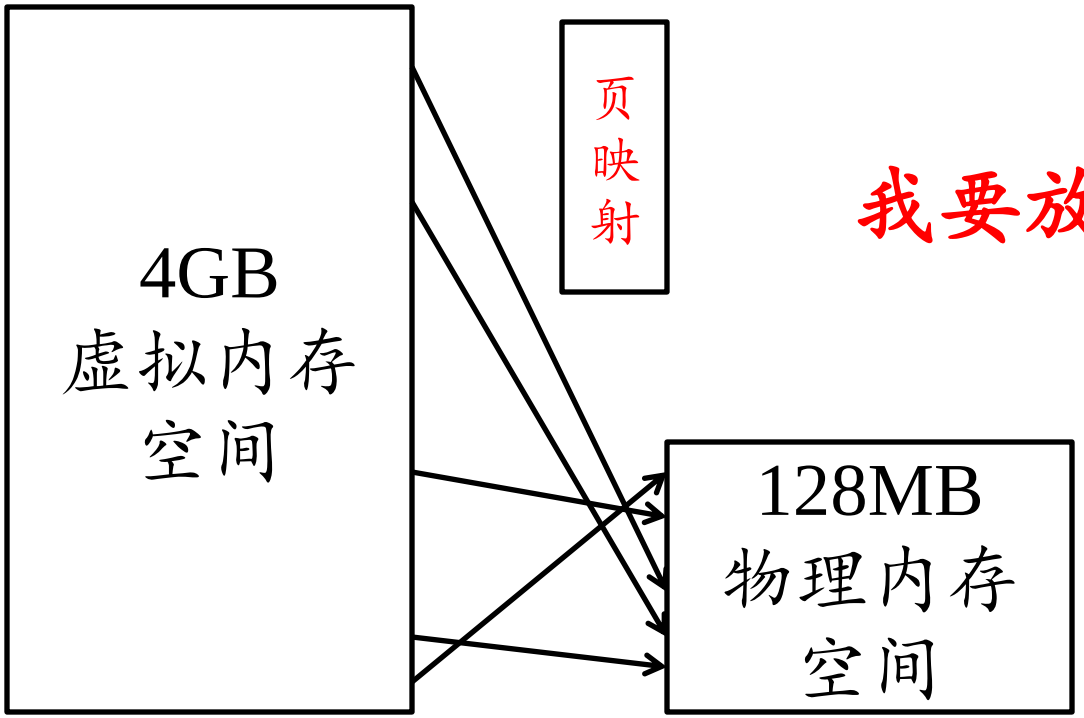
另辟蹊径

- 期末考试前，zrz问我，能不能用一个简单公式直接做这些映射，我说，应该没有吧。。。

- 但是，期末考试之后，有一天我突发奇想
- 想到了一个办法。。。。

NEMU的内存模块在做些什么

- 用软件代码模拟硬件MMU的功能
- 将虚拟地址转换为物理地址
- 为虚拟机中的程序提供一块4GB的虚拟内存空间



我要放大招了！

充分利用硬件MMU

- 我们的NEMU也运行在Linux提供的分页机制下
- NEMU每一次内存访问，都要经过硬件MMU转换
- 与其用软件代码模拟MMU的功能
- 为何不直接利用硬件MMU这一“免费”资源呢？

- x86-64为我们提供了至少TB级的虚拟内存空间
- 从中拿出区区4GB根本不是问题

- 注意，这里4GB是指虚拟内存空间！
- 并不是真的开了4GB大小的数组。

思路

- 开一块128MB的“物理内存”
- 开一块4GB地址空间，初始为空，设起始地址为base
- 若虚拟机想访问addr处，我们直接访问base+addr处
- 若发生缺页异常，我们则对出现异常的位置进行判断
  - 若发生在4GB地址空间外，则NEMU自身出了问题，终止
  - 若发生在4GB地址空间内，说明虚拟机第一次想访问该地址
    - 对出现缺页的地址addr调用page_translate()进行页转换
    - 在“物理内存”和4GB地址空间之间建立映射
    - 恢复程序的执行
- 当CR3变化时，直接清空整个4GB地址空间

- 整个过程有点像“把磁盘上的页面换入内存”的过程

一些问题

- 跨页读判断？
  - 硬件自动帮我们解决了！
- 内存映射I/O判断？
  - 没办法支持is_mmio()判断
  - 还好，只有显存是内存映射I/O的方式，特殊处理一下
- 缺点？
  - 要求整块的4GB地址空间
  - 因此必须要64位CPU、操作系统
- 怎样实现？
  - 需要通过操作系统接口来完成。

怎样实现

- 由于安全方面的原因，现代的操作系统是不会允许直接操作物理内存、页表的！所以一切都要使用操作系统的接口/库来完成。

- 如何让一块内存同时映射到两个位置（物理、虚拟）？
  - **POSIX Shared Memory**
- 如何建立映射？
  - **mmap()**
- 如何处理缺页（访问非法地址）？
  - **libsigsegv**

代码省略

- 由于代码涉及到很多与NEMU无关的细节
- 所以在PPT上就不贴出代码了

- 请参考shm-test.c

每次访问内存只需一次加法！

- **63.71 MIPS**
- **比较：**
- **刚才：40.45 MIPS (1.58x)**
- **最初：0.8952 MIPS (71.2x)**

```
[1471874220.649953753] speed: 63673367.46 instrs/sec, 0.00 hlts/sec, jitter = 0.
27 ms (2.74%), eip = PAL_DrawCharOnSurface+0042
[1471874221.705680425] speed: 63665575.04 instrs/sec, 0.00 hlts/sec, jitter = 0.
26 ms (2.58%), eip = PAL_RLEBlitToSurface+00C8
[1471874222.757837835] speed: 63867280.19 instrs/sec, 0.00 hlts/sec, jitter = 0.
26 ms (2.61%), eip = process_keys+0088
[1471874223.812556380] speed: 63714475.33 instrs/sec, 0.00 hlts/sec, jitter = 0.
24 ms (2.42%), eip = PAL_GetCurrentMap+000F
[1471874224.868265850] speed: 63658750.69 instrs/sec, 0.00 hlts/sec, jitter = 0.
24 ms (2.40%), eip = process_keys+0018
```

能不能再快一点？

再快一点？

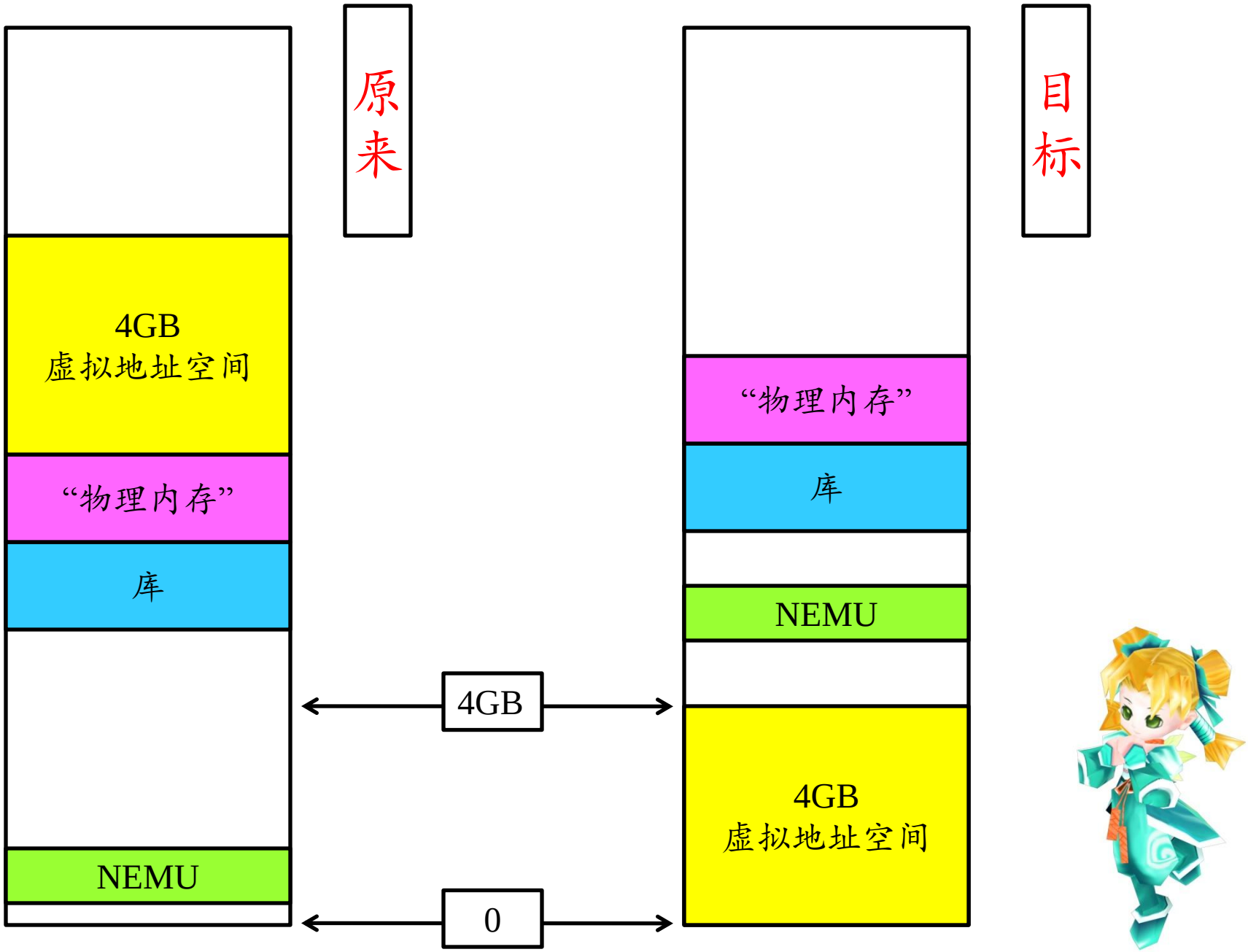
- “每次访问内存只需一次加法”

以exec()为例，读内存只剩三条指令！

```
mov     0x2a7e9c1(%rip),%rdx          # 72a8bfe8 <virt_mem>
mov     %edi,%eax
movzbl  (%rdx,%rax,1),%eax
mov     %eax,0x2ae0e4d(%rip)          # 72aee480 <ops_decoded>
mov     0x70016460(,%rax,8),%rax
jmpq    *%rax
nopl    (%rax)
```

- 能不能连“一次加法”都省掉？

- 需要把低4GB地址空间空出来
- 把NEMU代码和数据放到高于4G的地址空间去


原来
目标
4GB
虚拟地址空间
“物理内存”
库
NEMU
4GB
0
“物理内存”
库
NEMU
4GB
虚拟地址空间

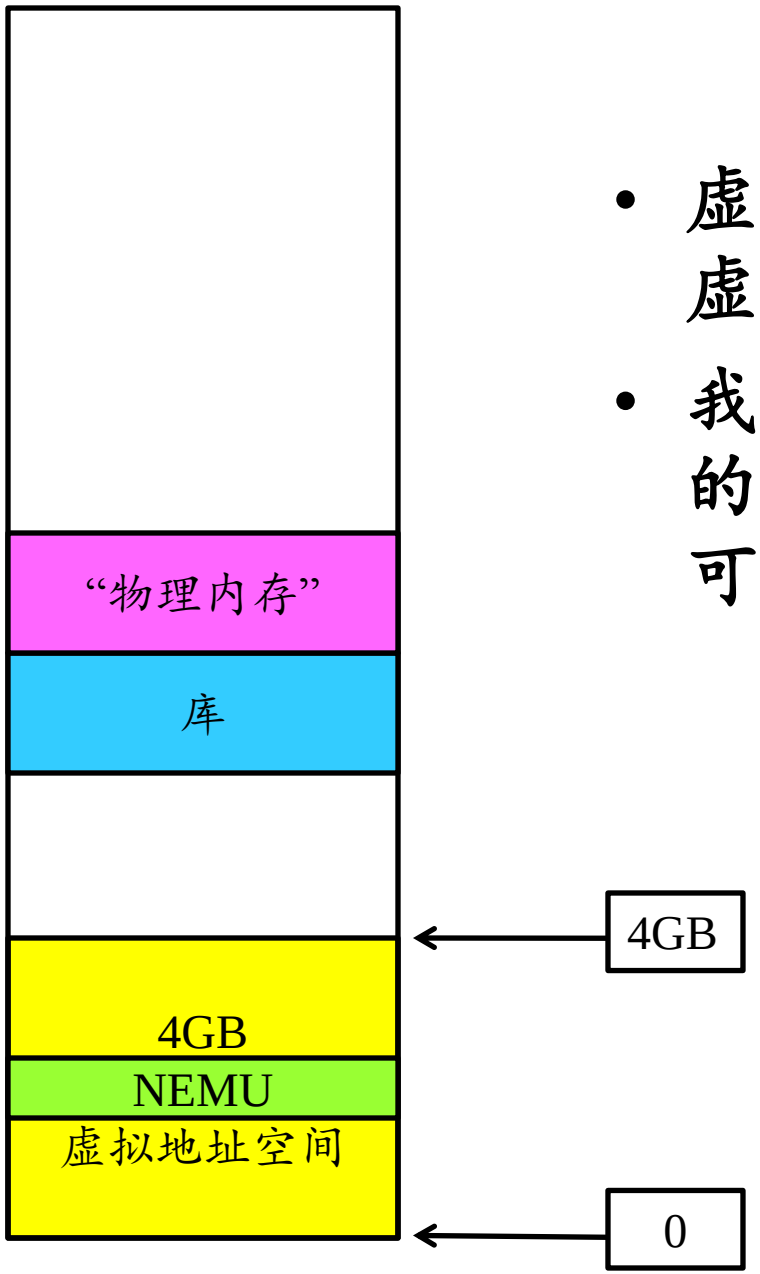
理想与现实

- x86_64对于代码放在较低位置有特殊优化
- 若把NEMU代码放在高于4GB位置
- 必须调整编译选项
- 调整后的代码比原来慢不少
- 虽然可能能省掉“一次加法”，但得不偿失

- 真的不行了吗？

曲线救国

- 虚拟机并没有使用全部4GB虚拟地址空间！
- 我们只要找个虚拟机没用到的地方，把NEMU塞进去即可！



每次访问内存只需一条指令！

```
movzbl (%edi),%eax
mov    %eax,0x2ac6df6(%rip)          # 72ac
mov    0x70014c00(,%rax,8),%rax
jmpq   *%rax
data32 data32 nopw %cs:0x0(%rax,%rax,1)
```

- **66.87 MIPS**
- **比较：**
- **刚才：63.71 MIPS (1.05x)**
- **最初：0.8952 MIPS (74.7x)**

- **只快了一点点(5%)**
- **此时瓶颈已不在内存**

```
[1471918553.736515941] speed: 67056559.88 instrs/sec, 0.00 hlts/sec, jitter = 0.25 ms (2.
45%), eip = PAL_RLEBlitToSurface+0097
[1471918554.752360412] speed: 66160845.25 instrs/sec, 0.00 hlts/sec, jitter = 0.26 ms (2.
59%), eip = PAL_RLEBlitToSurface+00AE
[1471918555.754865147] speed: 67029367.17 instrs/sec, 0.00 hlts/sec, jitter = 0.25 ms (2.
46%), eip = PAL_RLEBlitToSurface+009A
[1471918556.759453718] speed: 66893333.68 instrs/sec, 0.00 hlts/sec, jitter = 0.26 ms (2.
63%), eip = PAL_RLEBlitToSurface+00CD
[1471918557.758910154] speed: 67236748.30 instrs/sec, 0.00 hlts/sec, jitter = 0.23 ms (2.
33%), eip = PAL_RLEBlitToSurface+00CA
```

能不能再快一点？

NEMU运行缓慢
的终极解决方案！

换电脑！

换电脑

- 当然我并不是真的买了台新的电脑

- 我之前的速度都是在我的笔记本上跑出来的。
- **Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz**
  - 主频只有1.70GHz, 没有睿频

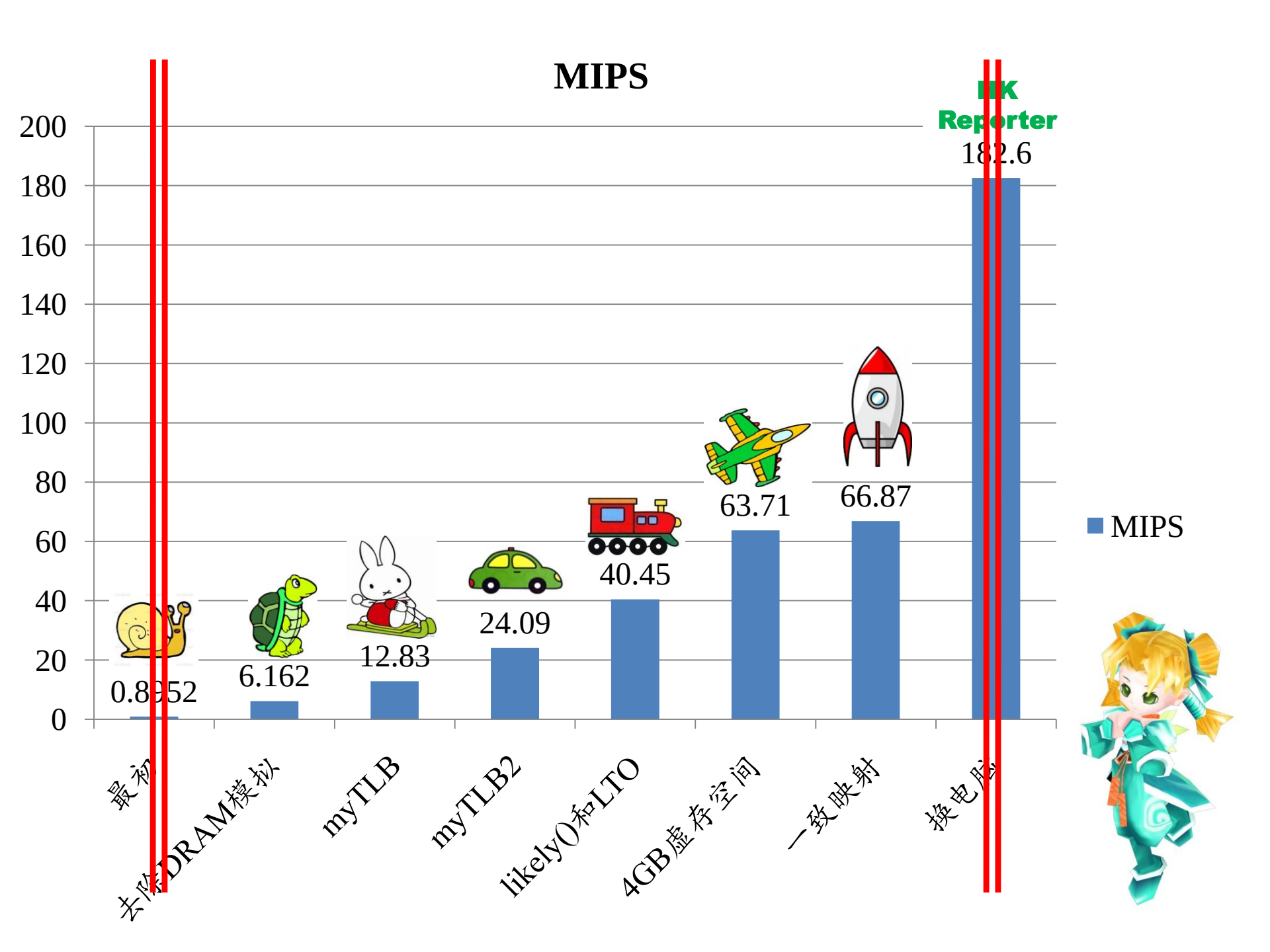
- 完全相同的代码，放到台式机上：
- **Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz**

4.00/1.70=2.35

- **182.6 MIPS**
- **比较：**
- **刚才：63.71 MIPS (2.87x)**
- **最初：0.8952 MIPS (204x)**

```
[1472823627.571172145] speed: 182008895.83 instrs/sec, 0.00 hlts/sec, jitter = 0
.08 ms (0.77%), eip = process_keys+002D
[1472823627.937542434] speed: 183216885.40 instrs/sec, 0.00 hlts/sec, jitter = 0
.08 ms (0.83%), eip = PAL_RLEBlitToSurface+00A5
[1472823628.305038448] speed: 182682723.93 instrs/sec, 0.00 hlts/sec, jitter = 0
.08 ms (0.81%), eip = process_keys+0034
[1472823628.672236508] speed: 182806136.68 instrs/sec, 0.00 hlts/sec, jitter = 0
.10 ms (0.97%), eip = SDL_GetTicks+0057
[1472823629.040192348] speed: 182414578.19 instrs/sec, 0.00 hlts/sec, jitter = 0
```

总结


MIPS
200
180
160
140
120
100
80
60
40
20
0
0.8952
6.162
12.83
24.09
40.45
63.71
66.87
182.6
MIPS
最初
去除DRAM模拟
myTLB
myTLB2
likely()和LTO
4GB虚存空间
一致映射
换电脑

Q&A

谢谢！