南開大學
Nankai University

计算机学院

计算机系统设计实验报告

从一到无穷大：程序与性能

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2024 年 6 月 12 日

目录

1 概述

1.1 实验目的

1. 了解 nemu 中的浮点数存储方式。
2. 分析 numu 的性能瓶颈
3. 在能力范围内做优化

1.2 实验内容

1. 实现两程序的画面切换。
2. 实现 nemu 中的浮点数 FLOAT。
3. 使用 perf 性能分析工具分析程序运行过程中的性能瓶颈并思考优化思路。

2 编写不朽的传奇

2.1 展示你的计算机系统

加载第 3 个用户程序/bin/videotest, 在 nanos-lite/src/main.c 中添加：

```
load_prog("/bin/pal");
load_prog("/bin/hello");
load_prog("/bin/videotest");
```

加载 videotest 程序。

运行 nemu, 由于当前仅前只允许最多一个需要更新画面的进程参与调度，且 pal 程序先被加载，因此当前画面是仙剑奇侠传程序的画面。

想要在按下 F12 的时候, 让游戏在仙剑奇侠传和 videotest 之间切换，首先需要在 Nanos-lite 的 events_read() 函数添加对按键 F12 的识别，再识别到按下 F12 后进行程序切换。这个识别应该在识别到按键按下时进行识别，如果确认按下的是 F12，调用游戏切换函数。

```
if(key & 0x8000) {kd_ku = 'd'; key = key ^ 0x8000;}
else {
  kd_ku = 'u';
  if(key == _KEY_F12) {
    switch_game();
    Log("F12--switch game!");
  }
}
```

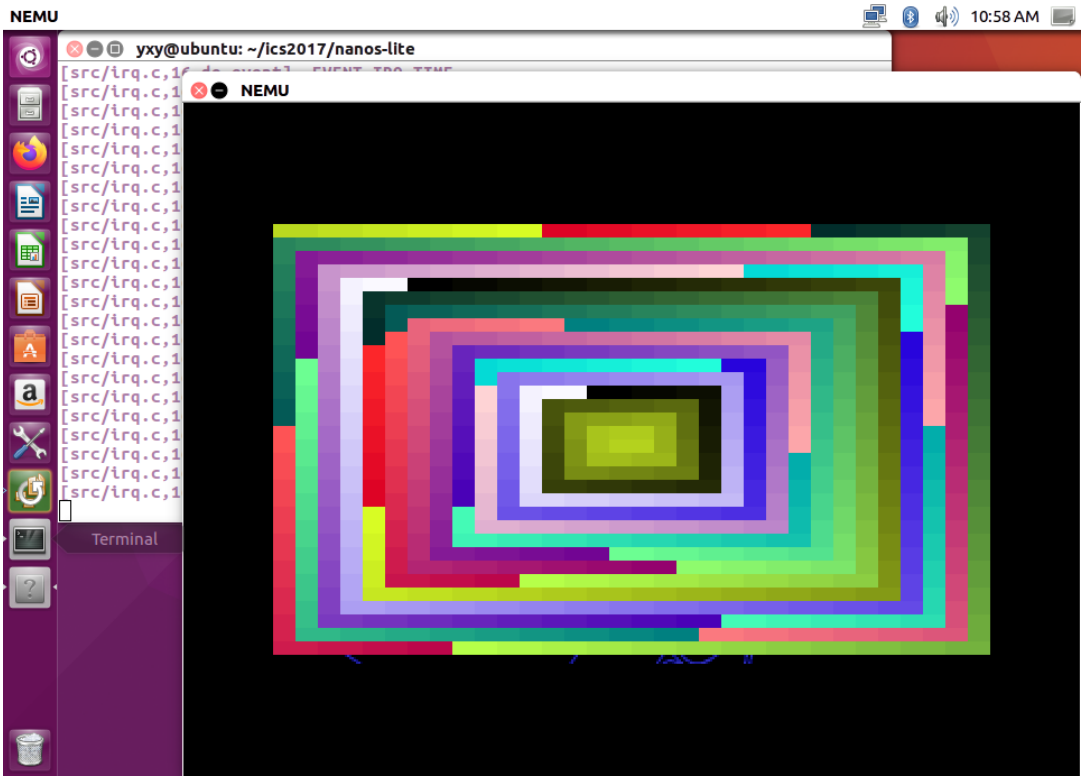
游戏切换函数定义在 nanos-lite/src/prog.c 中：首先在 prog.c 中设置变量 current_game 维护当前的游戏。根据 main.c 中的调用，需要切换画面的两程序进程号分别为 0 和 2，因此每次调用 switch_game() 函数都是让当前游戏在进程 0 与 2 间做切换：

```
int current_game = 0;
void switch_game(){
  current_game = 2 - current_game;
  Log("current game : %d", current_game);
}
```

在 schedule 函数中做进程切换时，就不应该是从进程 1 切换到进程 0 了，而应该是从进程 1 切换到当前游戏：

```
  if(count%1000 == 0) current = &pcb[1];
  else current = &pcb[current_game];
```

此时运行 NEMU，最初画面为仙剑奇侠传程序，按下 F12 按键后，切换到 videotest 程序：



3 浮点数的支持

在 NEMU 中，使用 binary scaling 方法，用整数来模拟实数的运算，使用 32 位整数来表示一个实数。

将 32 位整数表示的实数类型称为 FLOAT，最高位为符号位，后面的 15 位为整数部分，低 16 位为小数部分。

对于一个实数 a, 它的 FLOAT 类型表示 $A = a * 2^{16}$, 负实数使用相应正数的相反数表示。

假设实数 a, b 的 FLOAT 类型表示分别为 A, B 对于 FLOAT 的运算：

$$A + B = a * 2^{16} + b * 2^{16} = (a + b) * 2^{16}$$

$$A - B = a * 2^{16} - b * 2^{16} = (a - b) * 2^{16}$$

$$A * B = a * 2^{16} * b * 2^{16} = (a * b) * 2^{32}$$

$$A/B = (a * 2^{16})/(b * 2^{16}) = a/b$$

FLOAT 类型的加减运算可以用整数的加减运算来进行，乘除需要另作处理。

FLOAT 类型的关系运算可以用整数的关系运算来进行。

3.1 比较 FLOAT 和 float

Q: FLOAT 和 float 能表示的数集是不一样的. 思考一下, 我们用 FLOAT 来模拟表示 float, 这其中隐含着哪些取舍

A:

- 表示范围不同:float 的表示范围是-3.402e38~3.402e38;FLOAT 的表示范围是-32767.9999 ~32767.9999。FLOAT 的表示范围远小于 float.
- 精度不同：实数存为 FLOAT 时损失的精度更多。

3.2 FLOAT 与 float 转换

在了解了 NEMU 中浮点数如何存储后，需要去实现它的存储转换。

3.2.1 FLOAT 模拟浮点数

浮点数 float 的存储方式遵从 IEEE 的规范。

将符号位、指数、尾数分别记为 S、E、M，数值为：$(-1)^S * M * 2^E$。根据前面对于 FLOAT 和 float 存储方式的了解，具体转换步骤如下:

- 根据 float 获取指数、尾数、符号位:

```
unsigned uf = *(unsigned *)(&a);
unsigned sign = uf >> 31;
int exp = (uf >>23) & 0xFF;
unsigned frac = uf & 0x7FFFFF;
```

- 通过指数获取移码:

```
exp -= 127
```

- 判断是否溢出，指数大于 7 时小数位不足，需要向左移动；指数位小于 7 则小数位溢出，需要右移:

```
if (exp >= 7 && exp < 22)
  res = frac << (exp - 7);
else if (exp < 7 && exp > -32)
  res = frac >> 7 >> -exp;
else
  assert(0);
```

- 最后判断符号位正负，根据正负决定是返回本身还是返回相反数：

```
return (sign) ? -res : res;
```

3.2.2 FLOAT 模拟运算

根据前面对于 FLOAT 的了解，FLOAT 类型的加减运算以及关系运算可以用整数的加减运算以及关系运算模拟，但乘除运算以及绝对值运算需要做额外的处理：

乘法： 根据

$$A * B = a * 2^{16} * b * 2^{16} = (a * b) * 2^{32}$$

为了得到正确的结果，我们需要将相乘结果除以 2^{16}（即右移 16 位）即可得到正确结果。该运算实现在 navy-apps/apps/pal/src/FLOAT/FLOAT.c 中的 F_mul_F 函数中：

```
FLOAT F_mul_F(FLOAT a, FLOAT b)
{
  // assert(0);
  // return 0;
  return ((int64_t)a * (int64_t)b) >> 16;
}
```

除法： 根据

$$A + B = (a * 2^{16})/(b * 2^{16}) = a/b$$

为了得到正确的结果，我们需要需要进行逐次左移进行列式计算得到正确结果。该运算实现在 navy-apps/apps/pal/src/FLOAT/FLOAT.c 中的 F_div_F 函数中：

```
FLOAT F_div_F(FLOAT a, FLOAT b)
{
  // assert(0);
  // return 0;
  assert(b != 0);
  FLOAT x = Fabs(a);
  FLOAT y = Fabs(b);
  FLOAT z = x / y;
  x = x % y;

  for (int i = 0; i < 16; i++)
  {
    x <<= 1;
    z <<= 1;
    if (x >= y)
    {
      x -= y;
      z++;
    }
  }
  if (((a ^ b) & 0x80000000) == 0x80000000)
  {
    z = -z;
  }
  return z;
}
```

绝对值:

```
FLOAT Fabs(FLOAT a)
{
  return (a > 0) ? a : -a;
}
```

3.3 FLOAT 与 int 转换

3.3.1 类型转换

根据 FLOAT 的存储结构，对于正数而言，FLOAT 右移 16 位即为 int，int 左移 16 位即为 FLOAT；对于负数而言，需要先通过取相反数获取数值部分，然后左移或右移获取数值部分的 FLOAT 或 int 值，最后再取相反数得到最终结果：

```
static inline int F2int(FLOAT a) {
 // assert(0);
  //return 0;
 if ((a & 0x80000000) == 0)
    return a >> 16;
  else
    return -((-a) >> 16);
}
```

```
static inline FLOAT int2F(int a) {
  //assert(0);
  //return 0;
  if ((a & 0x80000000) == 0)
    return a << 16;
  else
    return -((-a) << 16);
}
```

3.3.2 模拟运算

FLOAT 与 int 的乘除运算可以通过类型转换变为 FLOAT 的乘除运算：

```
static inline FLOAT F_mul_int(FLOAT a, int b) {
  //assert(0);
  //return 0;
  return F_mul_F(a, int2F(b));
}
```

```
static inline FLOAT F_div_int(FLOAT a, int b) {
  //assert(0);
  //return 0;
  return F_div_int(a, int2F(b));
}
```

3.4 指令实现

运行程序，进入打怪界面，但是出现指令未实现的报错：

```
[src/irq.c,16,do_event] _EVENT_IRQ_TIME
invalid opcode(eip = 0x0804a066): 0f ac d0 10 c1 fa 10 50 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0804a066 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0804a066) in the disassembling result to distinguish which case
 it is.

If it is the first case, see
 _ _____   ___     __    __ __                          _
(_)___ \ / _ \   / /    |  \/  |                        | |
  _ __) | (_) |/ /_    | \  / | __ _ _ __   _   _  __ _| |
 | |__ < > _ <| '_ \   | |\/| |/ _` | '_ \| | | |/ _` | |
 | |___) | (_) | (_) |  | |  | | (_| | | | | |_| | (_| | |
 |_|____/ \___/ \___/   |_|  |_|\__,_|_| |_|\__,_|\__,_|_|

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!
```

通过查看 i386 手册，0f 处是 2-byte escape，在 2 byte_opcode_table 的 ac 处的指令是 shrd。

shrd 指令与 shld 指令相对应，对两个指令都进行实现：

3.4.1 shrd

根据 opcode table 在 nemu/src/cpu/exec 的 exec.c 的 2 byte_opcode_table 中 ac 位置补上该指令。

```
/* 0xac */    IDEX(Ib_G2E, shrd), EMPTY, EMPTY, IDEX(E2G,imul2),
```

之后在 all-instr.h 中增加对 shrd 指令的 Helper 函数的声明：

```
make_EHelper(shrd);
```

然后在 logic.c 中对该函数进行定义。在 i386 手册 P394 找到了对 shrd 指令的说明：

```
Operation

(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
   IF ShiftAmt ≥ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[r/m, ShiftAmt - 1]; (* last bit shifted out on exit *)
      FOR i ← 0 TO OperandSize - 1 - ShiftAmt
      DO
         BIT[r/m, i] ← BIT[r/m, i - ShiftAmt];
      OD;
      FOR i ← OperandSize - ShiftAmt TO OperandSize - 1
      DO
         BIT[r/m,i] ← BIT[inBits,i+ShiftAmt - OperandSize];
      OD;
      Set SF, ZF, PF (r/m);
         (* SF, ZF, PF are set according to the value of the result *)
      Set SF, ZF, PF (r/m);
      AF ← UNDEFINED;
   FI;
FI;
```

根据说明实现 shrd 指令:

```
make_EHelper(shrd)
{
  rtl_shr(&t0, &id_dest->val, &id_src->val);
  rtl_li(&t2, id_src2->width);
  rtl_shli(&t2, &t2, 3);
  rtl_subi(&t2, &t2, id_src->val);
  rtl_shl(&t2, &id_src2->val, &t2);
  rtl_or(&t0, &t0, &t2);
  operand_write(id_dest, &t0);
  rtl_update_ZFSF(&t0, id_dest->width);

  print_asm_template3(shrd);
}
```

3.4.2 shld

根据 opcode table 在 nemu/src/cpu/exec 的 exec.c 的 2 byte_opcode_table 中 ac 位置补上该指令。

```
  /* 0xa4 */    IDEX(Ib_G2E, shld), EMPTY, EMPTY, EMPTY,
```

之后在 all-instr.h 中增加对 shld 指令的 Helper 函数的声明:

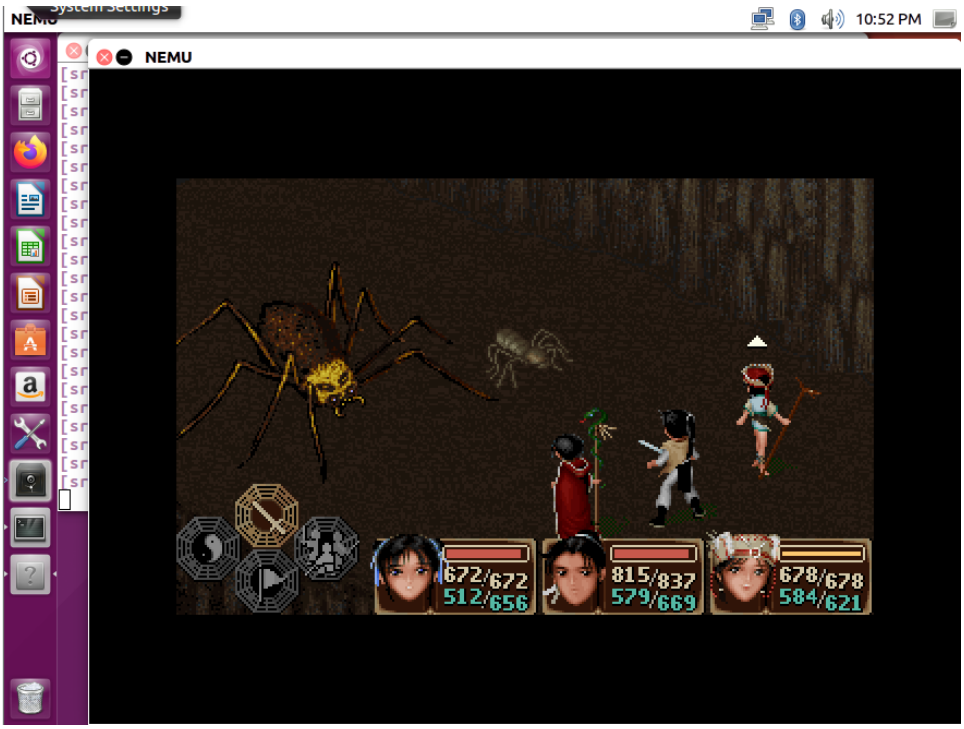
```
make_EHelper(shld);
```

根据 i386 手册对 shld 指令的说明, 在 logic.c 中实现 shld 指令:

```
make_EHelper(shld)
{
  rtl_shl(&t0, &id_dest->val, &id_src->val);
  rtl_li(&t2, id_src2->width);
  rtl_shli(&t2, &t2, 3);
  rtl_subi(&t2, &t2, id_src->val);
  rtl_shr(&t2, &id_src2->val, &t2);
  rtl_or(&t0, &t0, &t2);
  operand_write(id_dest, &t0);
  rtl_update_ZFSF(&t0, id_dest->width);

  print_asm_template3(shld);
}
```

重新运行程序，打怪可正常进行：

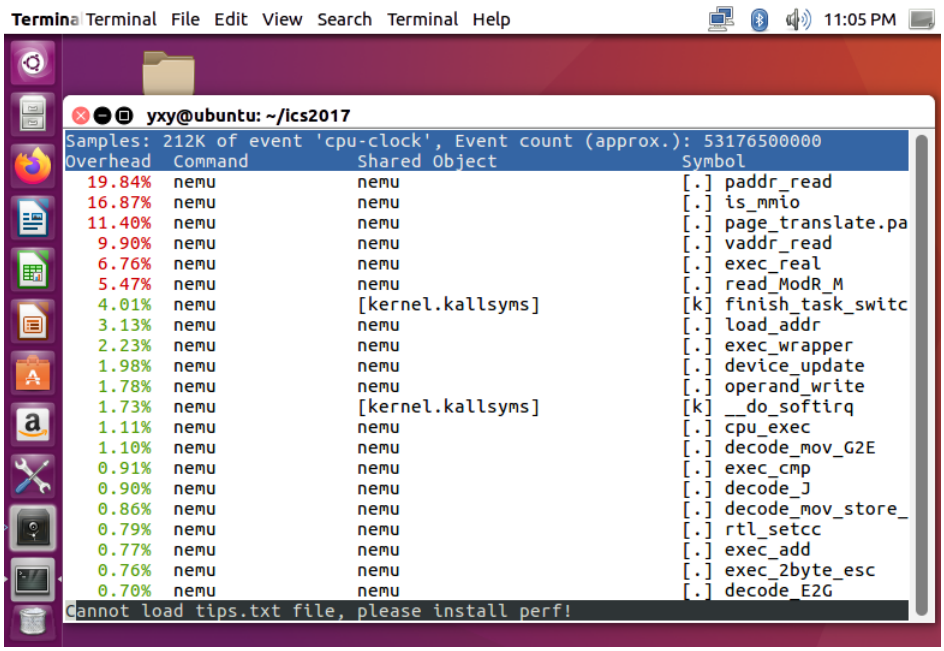


4 通往高速的次元

使用 perf 进行性能分析：

```
perf record nemu/build/nemu nanos-lite/build/nanos-lite-x86-nemu.bin
perf report
```

得到如下的性能分析结果：



通过性能分析可以发现，占用主要是在内存读取的时候。占用最高的函数 paddr_read 和物理地址读写函数 paddr_write 调用了占用第二的函数 is_mmio，而 paddr_read 是在占用率第四的 vaddr_read 函数和占用率第三的 page_translate 中被调用的。

提升占用率高的操作的性能可以更多的提升整体性能，因此可以将内存读取作为性能瓶颈去做优化。如果读取效率得到大规模提升，那么整体效率也会有比较明显的提升。