PA5实验报告

1711333 蒋璋

浮点数的支持

要在NEMU中实现浮点指令也不是不可能的事情. 但实现浮点指令需要涉及x87架构的很多细节, 根据KISS法则, 我们选择了一种更简单的方式: 我们通过整数来模拟实数的运算, 这样的方法叫binary scaling.

我们先来说明如何用一个32位整数来表示一个实数. 为了方便叙述, 我们称用binary scaling方法表示的实数的类型为`FLOAT`. 我们约定最高位为符号位, 接下来的15位表示整数部分, 低16位表示小数部分, 即约定小数点在第15和第16位之间(从第0位开始). 从这个约定可以看到, `FLOAT`类型其实是实数的一种定点表示.

```
31  30                  16                   0
+----+-------------------+--------------------+
|sign|      integer      |      fraction      |
+----+-------------------+--------------------+
```

这样, 对于一个实数`a`, 它的`FLOAT`类型表示`A = a * 2^16`(截断结果的小数部分)

浮点数和整数的运算指令

实现这部分指令其实是为了效率，不然通过int -> float -> FLOAT的转换也能达到想要的目的，只是那样的效率太低。而直接使用整数进行运算省去了将整数转化为浮点数的过程，大大减少了运算的次数。int型和FLOAT型之间的转化通过移位运算即可，因为`FLOAT = int * 2^16`, `int = FLOAT/2^16`，那么FLOAT型为int型左移16位；int型由FLOAT型右移16位。

浮点数除整数: `a * 2^16 / b` = `(a / b) * 2^16`

浮点数乘整数: `a * 2^16 *b` = `(a * b) * 2^16`

```
static inline int F2int(FLOAT a)
{
  return (a >> 16);
```

```
}

static inline FLOAT int2F(int a)
{
  return (a << 16);
}

static inline FLOAT F_mul_int(FLOAT a, int b)
{
  return a * b;
}

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

浮点数指令

浮点数乘除

(a * b) * 2^16 = (a * 2^16) * (b * 2^16) / 2^16

(a / b) * 2^16 = ((a * 2^16) / (b * 2^16)) * 2^16 + ((a * 2^16) % (b * 2^16) * 2 ^16) / (b*2^16)

除法可以先计算整数部分，再计算小数部分。小数部分可以继续拆分，移位多次。这里小数部分移位16次是为了避免a出现溢出。

```
FLOAT F_mul_F(FLOAT a, FLOAT b)
{
  return ((uint64_t)a * b) >> 16;
}

FLOAT F_div_F(FLOAT a, FLOAT b)
{
  assert(b != 0);
  int sign = 1;
  a = a < 0 ? -a : a;
  b = b < 0 ? -b : b;
  if ((a ^ b) & (0x1 << 31))
    sign = -1;

  int res = a / b;
  a %= b;
  for (int i = 0; i < 16; i++)
  {
    a <<= 1;
    res <<= 1;
    if (a >= b)
    {
      a -= b;
      res++;
    }
  }
  return res * sign;
```

```
}
```

float到FLOAT的转换

由于我们不能调用x87的指令（nemu中没有实现），所以我们不能通过强制类型转换把float变成int。那么我们只能手动地把float转化为FLOAT.

首先我们要清楚float在内存中的表示形式。在标准下, float使用ieee754标准来表示。由1位符号位，8位指数位和23位的尾数部分组成。指数位的偏移量为127，尾数省略小数点之前的1。那么将一个ieee754格式的浮点数表示成实数应该是((1<<23) + frac) * 2^(exp - 127 - 23)

因此将其转化为FLOAT型只需要再乘上 2 ^ 16即可，那么转化的结果就是

((1 << 23) + frac) * 2^(exp - 134)

但是指数部分还有例外

IEEE754规定，如果指数部分为0，为非规格浮点数，指数要多加1即1-127; 当指数为0，尾数为0的时候表示0; 当指数全1的时候表示无穷或者不定量，对这些特殊情况我们应该分开讨论。那么float到FLOAT的转换就可以实现出来了。

```
struct ieee754
{
  uint32_t frac : 23;
  uint32_t exp : 8;
  uint32_t sign : 1;
};
FLOAT f2F(float a)
{
  /* You should figure out how to convert `a' into FLOAT without
   * introducing x87 floating point instructions. Else you can
   * not run this code in NEMU before implementing x87 floating
   * point instructions, which is contrary to our expectation.
   *
   * Hint: The bit representation of `a' is already on the
   * stack. How do you retrieve it to another variable without
   * performing arithmetic operations on it directly?
   */

  struct ieee754 *f = (struct ieee754 *)&a;
  uint32_t res;
  uint32_t frac;
  int exp;

  if ((f->exp & 0xff) == 0xff)
  {
    // NaN or Inf
    assert(0);
  }
  else if (f->exp == 0)
  {
    exp = 1 - 127;
    frac = (f->frac & 0x7fffff);
  }
  else
  {
```

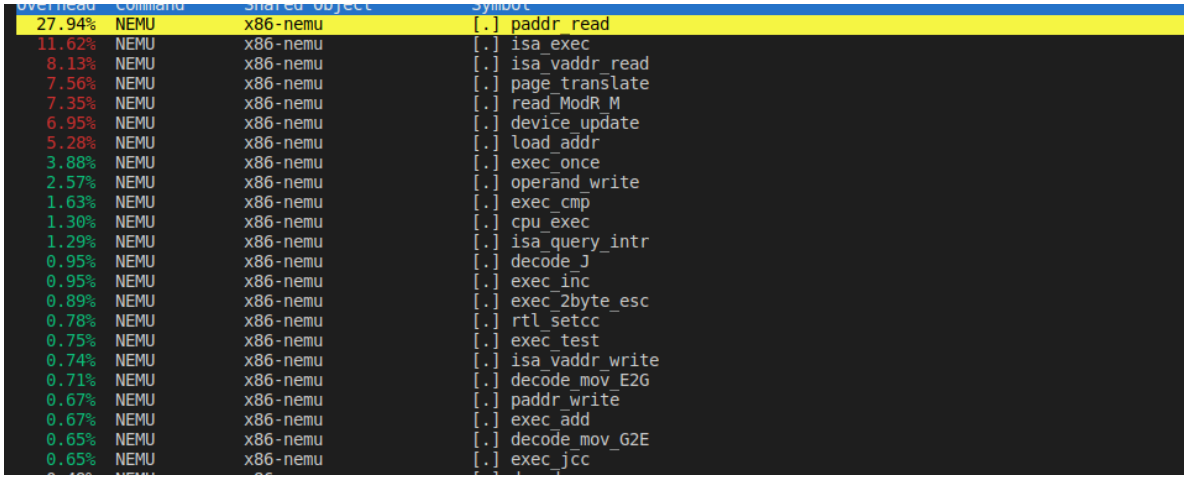
```
    exp = f->exp - 127;
    frac = (f->frac & 0x7fffff) | (1 << 23);
  }

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

  return (f->sign) ? -res : res;
}
```

运行结果

可以看到进入了对战画面

程序与性能

使用perf分析性能可以得到overhead结果如下图所示。在排名靠前的几个函数中，实际已经很难继续进行优化。这是nemu在设计之时为了简化考虑，很多地方并没有考虑性能。从图中可以看到，大部分的开销在于访存开销。为了对比引入虚拟存储造成的开销，我们可以对比一下PA3中的overhead



Overhead	Command	Shared Object	Symbol
27.94%	NEMU	x86-nemu	[.] paddr_read
11.62%	NEMU	x86-nemu	[.] isa_exec
8.13%	NEMU	x86-nemu	[.] isa_vaddr_read
7.56%	NEMU	x86-nemu	[.] page_translate
7.35%	NEMU	x86-nemu	[.] read_ModR_M
6.95%	NEMU	x86-nemu	[.] device_update
5.28%	NEMU	x86-nemu	[.] load_addr
3.88%	NEMU	x86-nemu	[.] exec_once
2.57%	NEMU	x86-nemu	[.] operand_write
1.63%	NEMU	x86-nemu	[.] exec_cmp
1.30%	NEMU	x86-nemu	[.] cpu_exec
1.29%	NEMU	x86-nemu	[.] isa_query_intr
0.95%	NEMU	x86-nemu	[.] decode_J
0.95%	NEMU	x86-nemu	[.] exec_inc
0.89%	NEMU	x86-nemu	[.] exec_2byte_esc
0.78%	NEMU	x86-nemu	[.] rtl_setcc
0.75%	NEMU	x86-nemu	[.] exec_test
0.74%	NEMU	x86-nemu	[.] isa_vaddr_write
0.71%	NEMU	x86-nemu	[.] decode_mov_E2G
0.67%	NEMU	x86-nemu	[.] paddr_write
0.67%	NEMU	x86-nemu	[.] exec_add
0.65%	NEMU	x86-nemu	[.] decode_mov_G2E
0.65%	NEMU	x86-nemu	[.] exec_jcc

在没有了虚拟存储以后，paddr_read在overhead的占比明显减少。由于执行的指令数是相同的，我们可以以isa_exec作为baseline, 计算出虚拟存储引入的overhead约为56%。这说明了mmu作为硬件存在的必要性，软件模拟带来的开销实在是太大了。

Overhead	Command	Shared Object	Symbol
18.22%	NEMU	x86-nemu	[.] isa_exec
12.10%	NEMU	x86-nemu	[.] read_ModR_M
11.26%	NEMU	x86-nemu	[.] paddr_read
10.19%	NEMU	x86-nemu	[.] device_update
9.95%	NEMU	x86-nemu	[.] load_addr
4.69%	NEMU	x86-nemu	[.] operand_write
3.65%	NEMU	x86-nemu	[.] exec_once
3.04%	NEMU	x86-nemu	[.] exec_cmp
2.25%	NEMU	x86-nemu	[.] paddr_write
2.14%	NEMU	x86-nemu	[.] decode_J
1.98%	NEMU	x86-nemu	[.] cpu_exec
1.92%	NEMU	x86-nemu	[.] exec_inc
1.33%	NEMU	x86-nemu	[.] rtl_setcc
1.27%	NEMU	x86-nemu	[.] exec_add
1.19%	NEMU	x86-nemu	[.] exec_jcc
1.15%	NEMU	x86-nemu	[.] exec_2byte_esc
1.07%	NEMU	x86-nemu	[.] exec_test
1.06%	NEMU	x86-nemu	[.] isa_vaddr_read
0.89%	NEMU	x86-nemu	[.] decode_r

问题 性能瓶颈的来源

Profiler可以找出实现过程中引入的性能问题, 但却几乎无法找出由设计引入的性能问题. NEMU毕竟是一个教学模拟器, 当设计和性能有冲突时, 为了达到教学目的, 通常会偏向选择易于教学的设计. 这意味着, 如果不从设计上作改动, NEMU的性能就无法突破上述取舍造成的障壁. 纵观NEMU的设计, 你能发现有哪些可能的性能瓶颈吗?

1. 访存

上面可以看到主要的性能瓶颈在于访存，一方面为了简易，我们没有实现TLB和cache的模拟，但是在实际的机器中，这两者是非常重要的东西，cache和TLB都能达到90%以上的命中率，能显著的减少访存的次数。我想原因大概有首先cache和TLB的实现比较复杂，如果出现缺失不好处理；另外也是很重要的一点就是由于cache和TLB也是用软件模拟，所以实际的性能可能不增反降，所以干脆采用鸵鸟算法，不去管他。

2. 执行单元

在主流的模拟器如qemu中，指令的执行单元是基本块，在没有遇到分支跳转或者是系统调用的时候，一次是可以执行很多条指令的，这样省去了很多检查。而且，我们应当将完成特定功能的若干指令组成的集合放在一起模拟，将若干指令一起翻译，而不是一条指令一条指令的翻译，这样既能减少翻译次数，还能提高执行效率，但是这样就太过复杂，不便于实现，因此nemu在这里页进行了取舍。