PA5 - 从一到无穷大: 程序与性能

世界诞生的故事 - 外传

先驱已经创造了一个功能齐全现代计算机, 终于可以来思考计算机系统领域中扩日持久的终极问题了: 如何让程序跑得更快?

浮点数的支持

我们已经在 PA3 中把仙剑奇侠传运行起来了, 但却不能战斗, 这是因为还有一些浮点数相关的工作需要处理. 现在到了处理的时候了. 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). 例如实数 `1.2` 和 `5.6` 用 `FLOAT` 类型来近似表示, 就是

```
1.2 * 2^16 = 78643 = 0x13333

+----+------------------+-------------------+

| 0  |        1         |       3333        |

+----+------------------+-------------------+
```

```
5.6 * 2^16 = 367001 = 0x59999

+----+------------------+--------------------+

| 0  |        5         |       9999         |

+----+------------------+--------------------+
```

而实际上，这两个 `FLOAT` 类型数据表示的数是:

```
0x13333 / 2^16 = 1.19999695

0x59999 / 2^16 = 5.59999084
```

对于负实数，我们用相应正数的相反数来表示，例如`-1.2`的 `FLOAT` 类型表示为:

```
-(1.2 * 2^16) = -0x13333 = 0xfffecccd
```

比较 FLOAT 和 float

`FLOAT` 和 `float` 类型的数据都是 32 位，它们都可以表示 2^32 个不同的数. 但由于表示方法不一样, `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?

接下来我们来考虑 `FLOAT` 类型的常见运算，假设实数 `a`, `b` 的 `FLOAT` 类型表示分别为 `A`, `B`.

- 由于我们使用整数来表示`FLOAT`类型, `FLOAT`类型的加法可以直接用整数加法来进行:

```
A + B = a * 2^16 + b * 2^16 = (a + b) * 2^16
```

-
- 由于我们使用补码的方式来表示 `FLOAT` 类型数据，因此 `FLOAT` 类型的减法用整数减法来进行.

```
A - B = a * 2^16 - b * 2^16 = (a - b) * 2^16
```

-
- `FLOAT` 类型的乘除法和加减法就不一样了:

```
A * B = a * 2^16 * b * 2^16 = (a * b) * 2^32 != (a * b) * 2^16
```

- 也就是说，直接把两个 `FLOAT` 数据相乘得到的结果并不等于相应的两个浮点数乘积的 `FLOAT` 表示. 为了得到正确的结果，我们需要对相乘的结果进行调整: 只要将结果除以 `2^16`，就能得出正确的结果了. 除法也需要对结果进行调整，至于如何调整，当然难不倒聪明的你啦.

- 如果把 `A = a * 2^16` 看成一个映射，那么在这个映射的作用下，关系运算是保序的，即 `a <= b` 当且仅当 `A <= B`，故 `FLOAT` 类型的关系运算可以用整数的关系运算来进行.

  有了这些结论，要用 `FLOAT` 类型来模拟实数运算就很方便了. 除了乘除法需要额外实现之外，其余运算都可以直接使用相应的整数运算来进行. 例如

  ```
  float a = 1.2;float b = 10;int c = 0;if (b > 7.9) {

    c = (a + 1) * b / 2.3;

  }
  ```

  用 `FLOAT` 类型来模拟就是

  ```
  FLOAT a = f2F(1.2);

  FLOAT b = int2F(10);int c = 0;if (b > f2F(7.9)) {

    c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));

  }
  ```

  其中还引入了一些类型转换函数来实现和 `FLOAT` 相关的类型转换.

  仙剑奇侠传的框架代码已经用 `FLOAT` 类型对浮点数进行了相应的处理. 你还需要实现一些和 `FLOAT` 类型相关的函数:

  ```
  /* navy-apps/apps/pal/include/FLOAT.h */int32_t F2int(FLOAT a);FLOAT int2F(int a);FLOAT F_mul_int(FLOAT a, int b);FLOAT F_div_int(FLOAT a, int b);/* navy-apps/apps/pal/src/FLOAT/FLOAT.c */FLOAT f2F(float a);FLOAT F_mul_F(FLOAT a, FLOAT b);FLOAT F_div_F(FLOAT a, FLOAT b);FLOAT Fabs(FLOAT a);
  ```

  其中 `F_mul_int()`和 `F_div_int()`用于计算一个 `FLOAT` 类型数据和一个整型数据的积/商，这两种特殊情况可以快速计算出结果，不需要将整型数据先转化成 `FLOAT` 类型再进行运算.

  事实上，我们并没有考虑计算结果溢出的情况，不过仙剑奇侠传中的浮点数结果都可以在 `FLOAT` 类型中表示，所以你可以不关心溢出的问题. 如果你不放心，你可以在上述函数的实现中插入 assertion 来捕捉溢出错误.

实现 binary scaling

实现上述函数来在仙剑奇侠传中对浮点数操作进行模拟. 实现正确后，你就可以在仙剑奇侠传中成功进行战斗了.

通往高速的次元

恭喜你! 你亲手一砖一瓦搭建的计算机世界可以运行真实的程序, 确实是一个了不起的成就! 不过通常来说, 仙剑奇侠传会运行得比较慢, 现在是时候对 NEMU 进行优化了.

说起优化, 不知道你有没有类似的经历: 辛辛苦苦优化了一段代码, 结果发现程序的性能并没有得到明显的提升. 事实上, Amdahl's law 早就看穿了这一切: 如果优化之前的这段代码只占程序运行总时间的很小比例, 即使这段代码的性能被优化了成千上万倍, 程序的总体性能也不会有明显的提升. 如果把上述情况反过来, Amdahl's law 就会告诉我们并行技术的理论极限: 如果一个任务有 5%的时间只能串行完成(例如初始化), 那么即使使用成千上万个核来进行并行处理, 完成这个任务所需要的时间最多快 20 倍.

跑题了... 总之, 盲目对代码进行优化并不是一种合理的做法. 好钢要用在刀刃上, Amdahl's law 给你最直接的启示, 就是要优化 hot code, 也就是那些占程序运行时间最多的代码. KISS 法则告诉你, 不要在一开始追求绝对的完美, 一个原因就是, 在整个系统完成之前, 你根本就不知道系统的性能瓶颈会出现在哪一个模块中. 你一开始辛辛苦苦追求的完美, 对整个系统的性能提升也许只是九牛一毛, 根本不值得你花费这么多时间. 从这方面来说, 我们不得不承认 KISS 法则还是很有先见之明的.

那么怎样才能找到 hot code? 一边盯着代码, 一边想"我认为...", "我觉得...", 这可不是什么靠谱的做法. 最可靠的方法当然是把程序运行一遍, 对代码运行时间进行统计. Profiler(性能剖析工具)就是专门做这些事情的.

GNU/Linux 内核提供了性能剖析工具 perf, 可以方便地收集程序运行的信息. 通过运行 `perf record` 命令进行信息收集:

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

如果运行时发现类似如下错误:

```
/usr/bin/perf: line 24: exec: perf_4.9: not found

E: linux-tools-4.9 is not installed.
```

请安装 `linux-tools`:

```
apt-get install linux-tools
```

通过 `perf record` 命令运行 NEMU 后, `perf` 会在 NEMU 的运行过程中收集性能数据. 当 NEMU 运行结束后, `perf` 会生成一个名为 `perf.data` 的文件, 这个文件记录了收集的性能数据. 运行命令 `perf report` 可以查看性能数据, 从而得知 NEMU 的性能瓶颈.

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?

天下武功唯快不破

相信你也已经在 NEMU 中运行过 microbench，发现 NEMU 的性能连真机的 1%都不到. 使用 `perf` 也没有发现能突破性能瓶颈的地方. 那 NEMU 究竟慢在哪里呢?

回想一下，执行程序，其实就是不断地执行程序的每一条指令: 取指，译码，执行，更新 `eip`... 在真机中，这个过程是通过高速的门电路来实现的. 但 NEMU 毕竟是个模拟器，只能用软件来实现"执行指令"的过程: 执行一次 `exec_wrapper()`, 客户程序才执行一条指令, 但运行 NEMU 的真机却需要执行上百条 native 指令. 这也是 NEMU 性能不高的根本原因. 为了方便叙述，我们将"客户程序的指令"称为"客户指令". 因此，作为软件模拟器的 NEMU 注定无法摆脱"用 n 条 native 指令模拟一条客户指令"的命运. 要提高 NEMU 的性能，我们就只能想办法减小 `n` 了.

事实上，模拟器的这种工作方式称为解释执行: 每一条客户指令的执行都需要经历完整的指令生命周期. 但仔细回顾一下计算机的本质，执行指令的最终结果就是改变计算机的状态(寄存器和内存)，而真正改变状态的动作，只有指令生命周期中的"执行"阶段，其它阶段都是为状态的改变作铺垫: 取指是为了取到指令本身，译码是为了看指令究竟要怎么执行，更新 `eip` 只是为了执行下一条指令. 而且，每次解释执行的时候，这些辅助阶段做的事情都是一样的. 例如

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
```

每次执行到这条指令的时候，取指都是取到相同的比特串，译码总是发现"要将 0x1234 移动到 `eax` 中"，更新 `eip` 后其值也总是 `0x100005`. 反正执行客户指令的结果就是改变计算机的状态，这不就和执行

```
mov $0x1234,(cpu.eax 的地址)
```

这条 native 指令的效果一样吗?

这正是即时编译(JIT) 的思想: 通过生成并执行 native 指令来直接改变(被模拟)计算机的状态. 这里的编译不再是"生成机器指令"的含义了，而是更广义的"语言转换": 把客户程序中执行的机器语言转换成与之行为等价的 native 机器语言. "即时"是指"这一编译的动作并非事先进行"，而是"在客户程序执行的过程中进行"，这样的好处是，不会被执行到的客户指令，就不需要进行编译.

为了生成 native 指令，我们至少也要知道相应的客户指令要做什么. 因此，我们至少也要对客户指令进行一次取指和译码. 通常情况下，客户指令不会发生变化，因此编译成的 native 指令也不会发生变化. 这意味着，我们只需要对客户指令进行一次编译，生成相应的 native 指令，然后存放起来，将来碰到相同的客户指令，就不必重新编译，而是可以找到之前编译的结果直接执行了. 这恰恰就是 cache 的思想: 我们将 native 指令序列组织成一个 TB(translation block), 用客户程序的 `eip` 来索引; 这个 cache 由一系列的 TB 组成; 执行客户程序的时候，先用 `eip` 索引 cache，若命中，说明相应的客户指令已经被编译过了，此时可以不必重新编译，直接取出相应的 TB 并执行; 若缺失，说明相应的客户指令还没有被编译过，此时才需要对客户指令进行取指和译码，并编译成相应的 TB，更新 cache，以便于将来多次执行.

一个值得考虑的问题是，每次编译多少条客户指令比较合适? 若一次编译一条客户指令，则会导致每执行完一条客户指令就需要重新对 cache 进行索引，查看下一条客户执行是否被编译过. 细心的你会发现，在即时编译模式中，一条客户指令被编译过，当且仅当它被执行过. 也就是说, NEMU 在执行完一条客户执行之后，都会去检查下一条指令有没有被执行过. 我们知道，顺序执行是程序最基本的执行流之一. 我们很容易想到，在一个顺序执行的模块中，如果其中的一条指令被执行过，那就意味着整个模块的每一条指令都已经被执行过. 这说明，若一次编译一条客户指令, "检查下一条指令有没有被执行过" 大多数时候是一个冗余的动作. 为了避免这些冗余的动作，我们可以一次编译一个顺序执行的模块，这样的模块称为基本块.

要如何进行编译呢? 我们知道, x86 的指令集非常复杂，如果要考虑每一条 x86 客户指令如何编译到 native 指令，就太麻烦了. 嘿，我们在 PA2 中引入的 RTL 就是用来解决这个问题的: RTL 只有少数的基本指令，我们只需要考虑如何将少数的 RTL 基本指令编译到 native 指令就可以了! 引入 RTL 还有另一个好处，就是方便 NEMU 的移植:

```
          +-----+
 x86 ---> |     | ---> mips
mips ---> | RTL | ---> arm
riscv ---> |     | ---> x86
          +-----+
|          |   |         |
+ front-end +   + back-end +
```

以 RTL 为分界，我们可以把即时编译模式的 NEMU 分为两部分: 前端用于将客户程序的机器语言编译成 RTL，后端负责将 RTL 编译成 native 机器语言. 这样以后，要在 NEMU 中支持一种新的客户程序架构 x，只需要增加相应的前端

模块来将 x 编译成 RTL 即可；要让 NEMU 运行在一种新的架构 y，只需要增加相应的后端模块来将 RTL 编译成 y 即可.

于是，即时编译模式的 NEMU 的工作方式如下:

```
while (1) {

  tb = query_cache(cpu.eip);

  if ( cache miss ) {

    tb = jit_translate(cpu.eip); // translate a basic block

    update_cache(cpu.eip, tb);

  }



  jump_to(tb); // cpu.eip will be updated while executing tb

}
```

关于`jit_translate()`如何工作，可以参考《QEMU, a Fast and Portable Dynamic Translator》
https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.html 它讲述 QEMU 中的 JIT 如何实现的文章.

什么？这就没有了?

事实上，实现 JIT 的坑非常多，不过如果你看到这里，相信你也有一定能力来面对这些坑了. 踩坑其实是非常非常宝贵的经验，也是做这些项目的意义所在: 通过做项目，知道了以前永远也不可能知道的东西. 上述文章提到, QEMU 的早期版本可以做到只比真机性能慢 4 倍. 看着 microbench 的分数越来越高，了解每一项技术带来的性能提升及其背后揭示的原理，这些都是最好的回报，也是系统方向科研人员所追求的奥义.

不妨尝试一下阅读 QEMU 的源代码，虽然现在的 QEMU 已经不是上述那篇十几年前的文章所说的那个样子。