体系结构 Lab

杨健邦 515030910223

实验 1 Cache

【步骤一】

计算两个程序的 cache miss 次数。这里只考虑数组是 cacheline 对齐的情况。由已知条件可得，L1 cache 中一共有 32 个 set，每个 set 中，有 1 条 cache line，每条 cache line 能容纳 8 个 int 类型的数据。

a. 左图程序，只会出现 cold miss， 每访问 8 次，有 1 次 cache miss，因而 cache miss 的次数应该是 4096 * 8 / 8 = 4096 次。
b. 右图程序，出现了 conflict miss。因此，对数组的每次访存操作都会出现 miss。cache miss 的次数应该是 4096 * 8 = 32768 次。

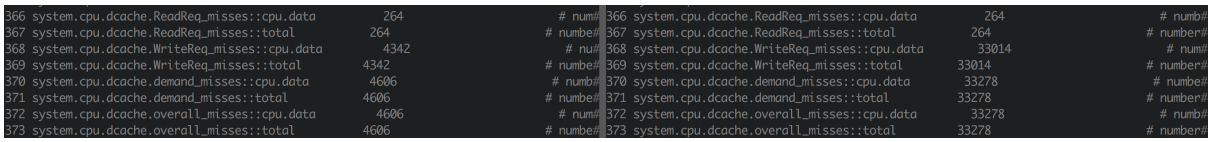
【步骤二】

根据实验要求，使用 gem5 自带的 se.py 程序来进行仿真，编译、链接和仿真命令为：

gcc -S -o proc.s proc.c
gcc -o proc -static proc.s
gem5/build/X86/gem5.opt gem5/configs/example/se.py -n 1 --caches --cacheline_size=32 --l1d_size=1kB --l1d_assoc=1 -c proc

【步骤三】

分别对比两个程序 gem5 仿真的 cache miss 次数和计算的 cache miss 次数的差异：



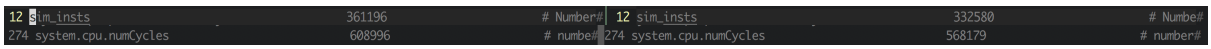
注：理论上程序只会出现 write miss，下面列出的是 write miss 对比。

程序	计算的 cache miss 次数	仿真的 cache miss 次数
左图程序	4096	4342
右图程序	32768	33014

两个程序计算得出的 cache miss 次数和仿真的 cache miss 次数大致相符，只有少量偏差，符合预期。

【步骤四】

对比两个程序的 **cache miss 次数**、**总周期数**和 **IPC**：



程序	Cache miss 次数	总周期数	IPC
左图程序	4342	608996	0.593100776
右图程序	33014	569179	0.584315303

左图程序 cache miss 次数少而总周期数多的原因是因为两个程序内层循环的总次数都是一样的，而外层循环左图程序循环了 4096 次而右图程序只循环了 8 次，因此左图程序执行的指令数比右图指令执行的指令数要多。

【步骤五】

实验结果与计算结果不严格相等的原因可能有:

a. 程序除了执行 main 函数里面的指令，还会在 main 函数之前和之后进行许多初始化的指令和退出指令，这些多出来的代码是在链接的时候被链接上去的，在这些代码中可能会有不少访存操作，所以也会造成不少 cache miss。

b. 仿真器可能还模拟了操作系统的调度，程序在执行的过程中可能会进行多次 context swtich，context switch 也会造成不少 cache miss。尤其这是个单核 CPU，调度的影响会更明显。

c. 在链接的时候，数组并不是按照 cacheline 对齐的。

实验 2 分支预测

【步骤一】

编写源程序，这里必须要将所有变量都标注 register，提示编译器将变量用寄存器存放，因为编译器不会做任何优化，如果变量在内存中存放（比如 i），会使得每次循环中访存变成了关键路径，je 和 cmovnz 的区别就体现不出来了。

```c
#include<stdio.h>

int main()
{
  register char x = 0, y = 64;
  for (register int i = 0; i < 1000000; ++i)
  {
    if (x != y)
      x = y;
  }
  return 0;
}
```

【步骤二】

gcc 不开优化编译产生 branch.s，然后修改汇编代码生成 cmovnz.s，再用 gcc 链接，这里由于要使用的是 cmpb 而不是 test，因此选择用 cmovnz 而不是 cmovz，两者的效果是相同的。

```
.L4:
    cmpb    %r12b, %r13b
    je .L3
    movl    %r12d, %r13d
.L3:
    addl    $1, %ebx
.L2:
    cmpl    $999999, %ebx
    jle .L4
```

```
.L4:
    cmpb    %r12b, %r13b
    cmovnz  %r12d, %r13d
    addl    $1, %ebx
.L2:
    cmpl    $999999, %ebx
    jle .L4
    movl    $0, %eax
    popq    %rbx
```

gcc -S -o branch.s source.c
gcc -S -o cmovnz.s source.c
gcc -o branch -static branch.s
gcc -o cmovnz -static cmovnz.s

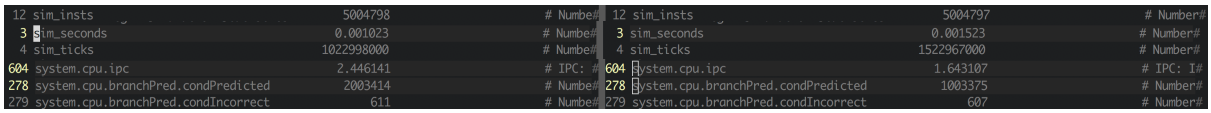
【步骤三】

使用 gem5 自带的 se.py 程序来进行仿真，选择 Deriv03CPU 为仿真的 CPU 模型，Deriv03CPU 为乱序流水线模型，符合实验要求，而且使用该模型的时候必须要使用 cache，具体参数配置为：

gem5/build/X86/gem5.opt gem5/configs/example/se.py --cpu-type=DerivO3CPU --caches -c branch
gem5/build/X86/gem5.opt gem5/configs/example/se.py --cpu-type=DerivO3CPU --caches -c cmovnz

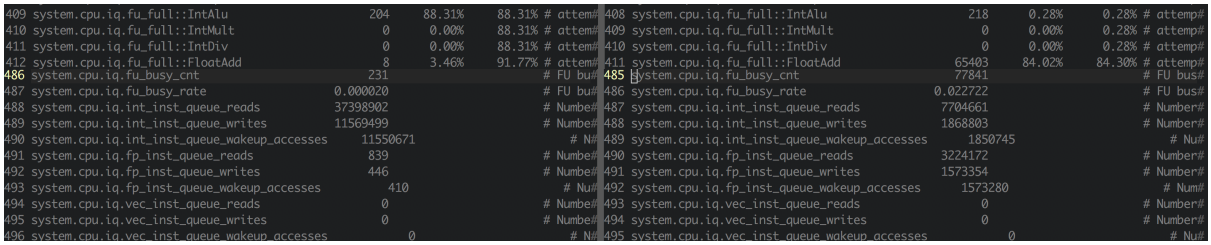
【步骤四】

罗列对比两者的**总指令数、运行时间、IPC、分支预测和流水线**信息。



程序	总指令数	运行时间/s	IPC	分支预测(错误/预测)
branch	5004798	0.001023	2.446141	611/2003414
cmovnz	5004797	0.001523	1.643107	607/1003375

流水线信息如下：



【步骤五】

结合汇编文件解释上面所列的数值差异性的原因。

a. 两个程序的总指令数几乎是一样的，说明执行时间、IPC 的差异并不是由指令数的差异产生的。

b. 由于 CMOV 指令的特性是“cmov will stall on eflags in your test program”，因此 cmovnz 并不能和之前 cmpb 流水线执行，它会一直 stall，直到 cmpb 指令结束才会执行。而 je 指令却可以通过分值预测的方式，和 cmpb 流水线地执行。从这个方面可以预测，如果分支预测足够准确的话，je 程序循环的关键路径会比 cmovnz 的关键路径要短。

c. 从分支预测信息来看，je 的分支预测情况是绝大部分的分支预测都正确了，因此几乎没有什么惩罚，可以验证我们上面 je 程序循环的关键路径会比 cmovnz 的关键路径要短的猜想。

d. 最后再结合流水线信息，可以发现 cmovnz 程序 function unit busy 的情况比 je 的要严重，而且 function unit 的 queue 比 je 的要长。这验证了上面 cmov 指令“cmov will stall on eflags in your test program”的观点。

实验 3 单指令多操作

【步骤一】

结合 SIMD.s，阐述矢量化程序的运算流程。

a. xmm0-7 系列寄存器能够存放 128 个 bit 的数据。
b. 首先使用 movdqa 指令先将数据段的 4 个 4Byte 的 1 加载到%xmm1 寄存器中，作为被加数。
c. 每次循环中一次性从数组中读入 16 个 Byte 的数据（4 个数组元素），使用 movdqa 指令加载到%xmm0 寄存器中。
d. 使用 paddd 指令将%xmm0 和%xmm1 的各自的 4 个元素同时分别进行加法操作。
e. 然后使用 movdqa 指令将%xmm0 中的 4 个元素再写入内存中。
f. 最后循环变量 i 加上 16 而不是 4，因为每次循环都处理了 4 个元素。

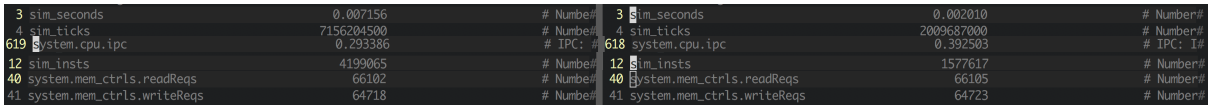
【步骤二】

使用 gem5 自带的 se.py 程序来进行仿真，选择 Deriv03CPU 为仿真的 CPU 模型，编译、链接和仿真命令为：

gcc -o Ori -static Ori.s
gcc -o SIMD -static SIMD.s
gem5/build/X86/gem5.opt gem5/configs/example/se.py --cpu-type=DerivO3CPU --caches -c SIMD
gem5/build/X86/gem5.opt gem5/configs/example/se.py --cpu-type=DerivO3CPU --caches -c SIMD

【步骤三】

对比两种实现方法的**运行时间、IPC、总指令数、访存请求个数：**



程序	运行时间/s	IPC	总指令数	访存请求个数
Ori	0.007156	0.293386	4199065	130820
SIMD	0.002010	0.392503	1577617	132828

【步骤四】

在 SMID.s 中的访存指令个数与仿真中的访存请求个数是否近似相等？如果相差很多，原因是什么？

在仿真结果 stats.txt 文件中找到了访存指令个数相关数据，访存指令个数为 1051761，而访存请求数有 132828，访存指令数比访存请求书多 7 倍左右，这可能是因为 cache 的原因，cold miss 后会把数据其它相邻的数据一起读进 cache 之中，而程序正好是顺序访问一个连续的数组，因此 cache 起了作用。

【步骤五】

结合实验数据，从体系结构的角度阐述 SIMD 性能更优的原因。

1. 由于 1 条指令可以操作多条数据，SIMD 的总指令数比 Ori 要少得多，这减少了取指的带宽。
2. 减少硬件的 data hazard 检测。
3. 内存访问的延迟被均摊了。