Lab3:性能分析

——181220010 丁豪

一、实现perf工具

1.解析命令行参数

经过查阅资料得知，argv代表调用可执行文件时的参数数量，args[]为各参数的字符串。这里我们需要分两种情况讨论，如果有-r参数，则分别解析次数和函数名，否则只解析函数名。

```
int main(int argc, char **argv) {
  int rounds = 1;
  void (*func)() = lookup("dummy");
  if (argc==4 && strcmp(argv[1],"-r")==0){
    sscanf(argv[2],"%d",&rounds);
    func = lookup(argv[3]);
  }
  else if (argc==2){
    func = lookup(argv[1]);
  }
  run(func, rounds);
}
```

2.统计时间

为了更加精确地统计时间，我们使用sys/time.h中的gettimeofday()函数，来获取精确到微妙的当前时间。具体实现过程经过STFW学习后如下，函数通过一个timeval结构体的引用来返回当前时间，fv_sec表示秒数,fv_usec表示微秒数，两者合一得到总微秒数作为返回值。

```
static uint64_t gettime() {
  struct timeval timenow;
  gettimeofday(&timenow,NULL);
  return (uint64_t)1000000*timenow.tv_sec+timenow.tv_usec;
}
```

3.数据处理与可视化

在此对统计量进行一些简单的数据分析与处理，最终在终端上直接显示如下统计量。

通过观察发现，多次运行同样的测试，最大值的差异性十分明显，推测与系统性行为有关，比如进程切换、程序局部性导致的性能差异（是否在cash中）等有关。设置总运行次数较大，可以有效使平均耗时符合真实情况，而相应地最大最小耗时却会变得更加极端一些，并在总次数不够大时对方差有较大影响（主要出现在最大值有峰值情况）。因此平均耗时将是我们主要考察的统计量，它具有最好的性质。

二、对multimod进行性能评估

1.multimod项目包含

直接复制multimod_p123三个函数到impl.c中，为了符合`void(*)()`格式，又给他们套了一层调用m123。通过全局变量aa,bb,mm分别给调用中的abm赋值，为了防止编译器报错与优化掉没有使用的un变量，又在每一个m函数中加了一个+1，其消耗的时间与multimod相比是微不足道的。

2.性能分析

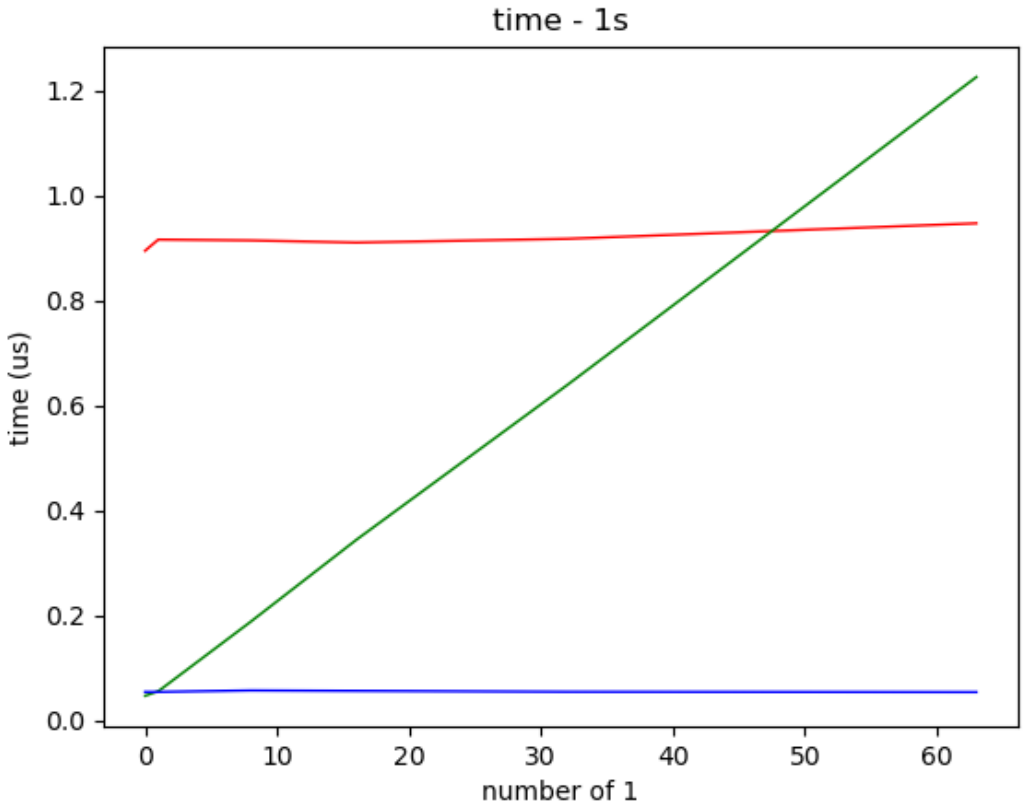
上文已经提到，均值将是我们分析的主要数据，这里分析次数较多，为了避免乱花渐欲迷人眼，我们干脆只看均值。修改main.c中的数据展示，使其只显示均值。

为了方便统计不同情况下3个函数的时间开销，我们继续修改main.c函数，把上图中的aa,bb,cc参数的定义改到main.c中，远处改为申明，这样方便在main.c种对他们进行赋初值与修改。

1）性能与a,b中1的数量的关系

控制a为0x00000000ffffffff，m为0x1000000000000000，循环总论述rounds=100000。将b的值从0x1开始逐渐增加1的个数直到0x7fffffffffffffff，统计3个函数在每一种情况中的时间消耗。为了方便观察，我们把他们放在一张表格中。

b中1的数量	m1平均耗时	m2平均耗时	m3平均耗时
0	0.894320	0.046400	0.053830
1	0.915650	0.055030	0.053850
8	0.914100	0.187490	0.056630
16	0.909940	0.343440	0.055790
32	0.917150	0.638720	0.054110
63	0.946630	1.225380	0.053510

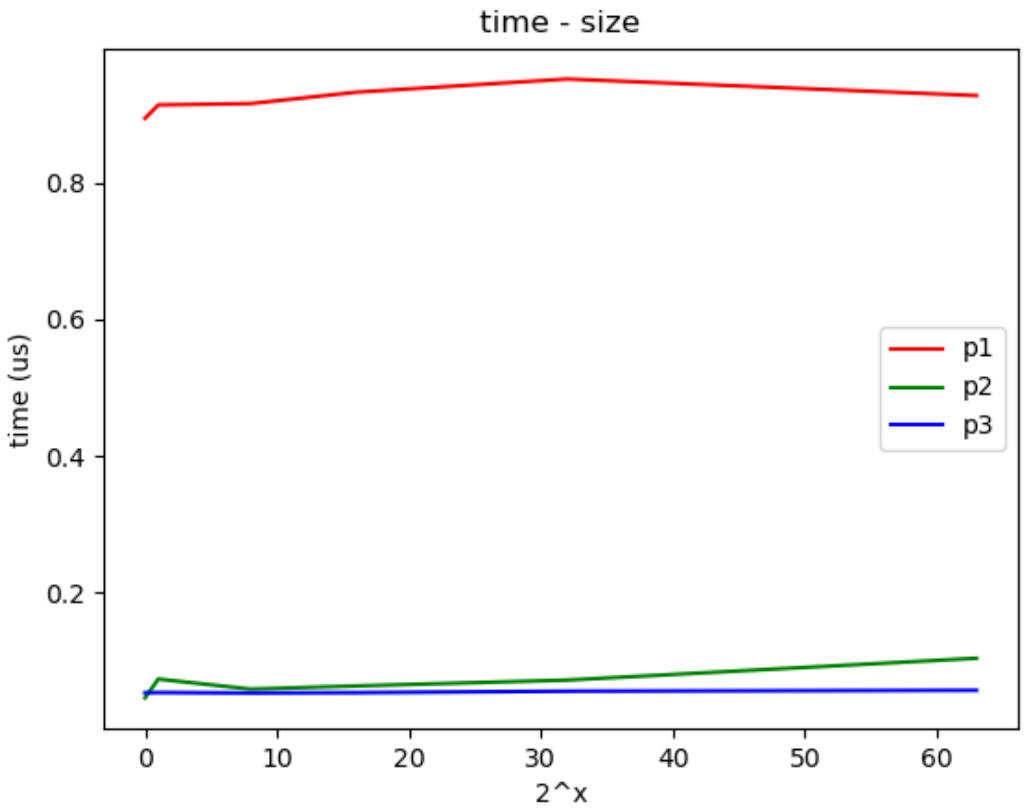


可以发现，m1与m3的时间与1的数量无明显关系，而m2与1的数量有明显的正相关性，甚至是线性相关。

2）性能与a,b的大小关系

在上一部分讨论中其实已经隐含了大小关系对时间的影响，这里为了避免1的数量导致的时间影响，仍控制a为0x00000000ffffffff，m为0x1000000000000000。b从0x1开始，逐渐添加尾数中0的数量，最终达到0x10000000000000000000。

b尾数中0的数量	m1平均耗时	m2平均耗时	m3平均耗时
0	0.894320	0.046400	0.053830
1	0.913890	0.073800	0.054120
8	0.915870	0.059120	0.053450
16	0.932530	0.063790	0.053960
32	0.951830	0.072330	0.056110
63	0.927630	0.104270	0.057400

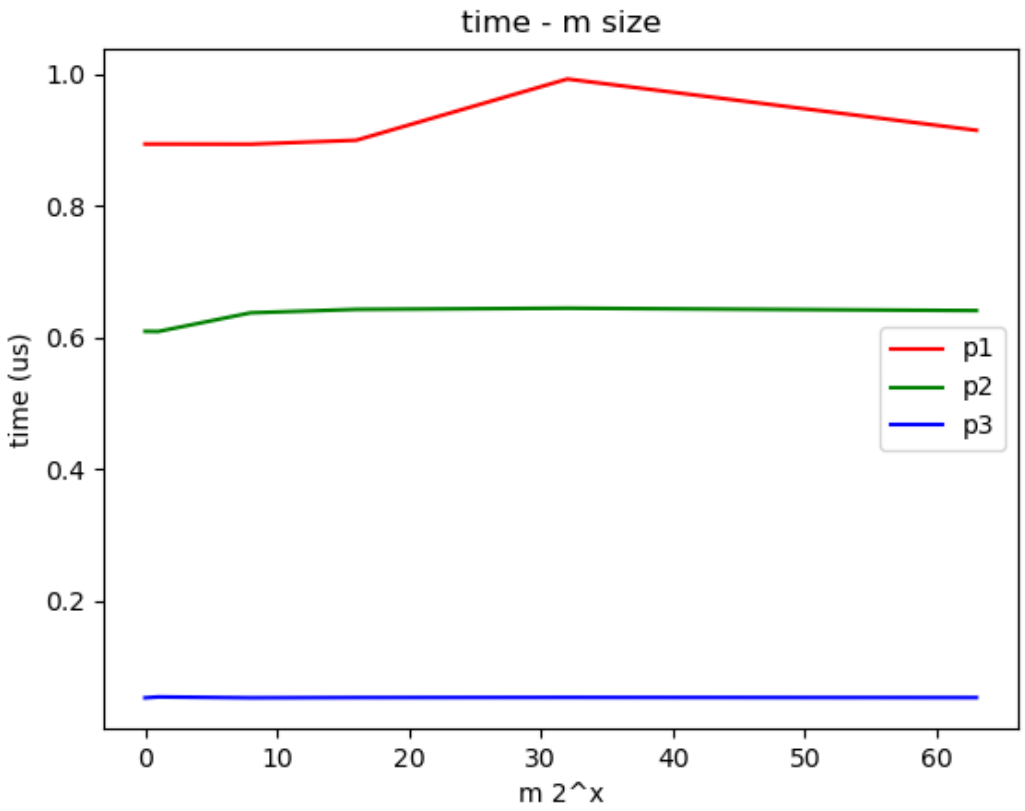


结论与上一测试有些许相似，m1与m3的耗时几乎不随b的大小而变，m2的时间正相关增加，但相关性不如上一实验中那么强。

3)性能与m大小的关系

此调查延续2)中的大小讨论，现在固定a,b都为0x00000000ffffffff，把m的大小按上述b的方式变化

m尾数中0的数量	m1平均耗时	m2平均耗时	m3平均耗时
0	0.893740	0.609530	0.053290
1	0.893740	0.609290	0.054670
8	0.893540	0.637610	0.053300
16	0.899520	0.642810	0.053690
32	0.992460	0.644620	0.053910
63	0.914630	0.640990	0.053630

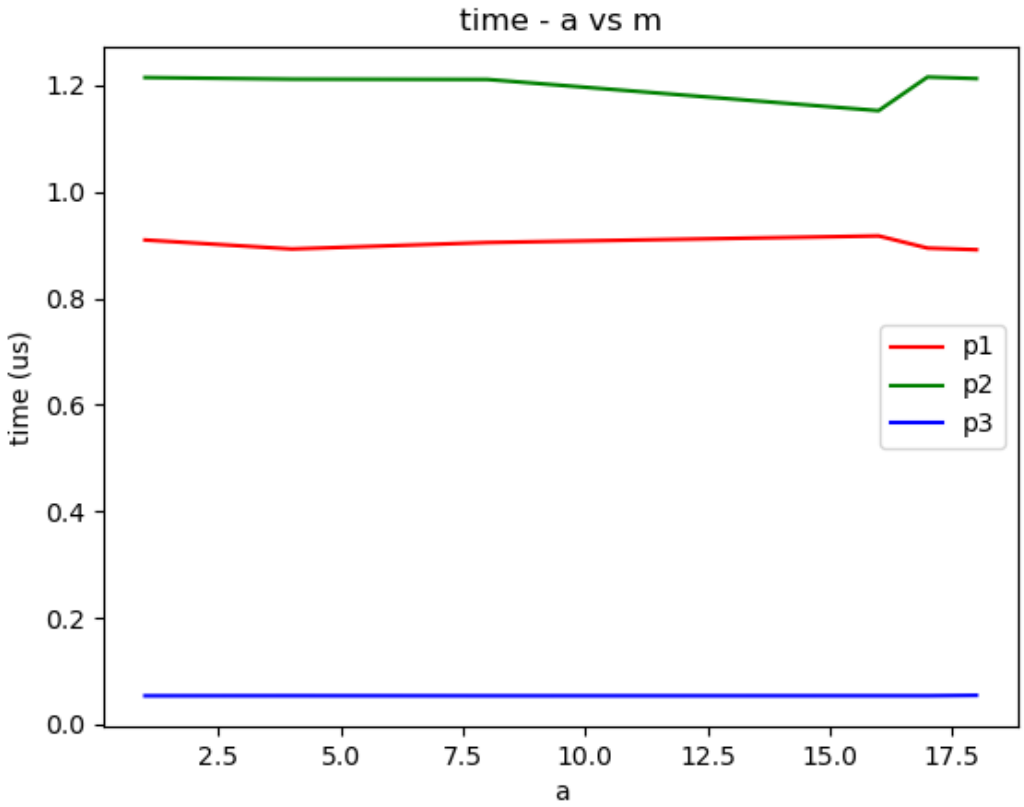


大体上，三者的时间开销与m都没有很大关系。

4)m与a的大小的关系

在m2的实现中，每一步都分别加上了一次a的值，并进行了mod m操作，因此觉得a能否被m整除，以及不整除的时候余数是多少，很可能对时间开销有较大影响（对m2）。于是进行如下测试，控制b=0x7fffffffffffffff，m=16，改变a的值，分别测试三个函数的时间开销。

a的值	m1平均耗时	m2平均耗时	m3平均耗时
1	0.909650	1.214770	0.053560
4	0.892730	1.211710	0.053710
8	0.904840	1.211110	0.053580
16	0.917250	1.152500	0.053660
17	0.894420	1.215690	0.053650
18	0.891330	1.212760	0.054430



可以发现当a mod m等于0时时间明显更小，而其他时候基本大差不差。因为我们的3种实现中并没有对取模后的余数情况进行优化处理，这样也是十分合理的。

三、思考题

还有哪些有趣的结论?

1.不论何种芯片的何种实现，其复制吞吐量总体来看都随着复制的字节数呈现递减趋势，并且递减速度逐渐放缓，最后吞吐量趋于收敛。

2.unroll在-O1优化等级下的表现普遍比-O2 -O3要好。

3.AMD EPYC芯片虽然对movs指令的硬件优化远不及Intel i5,但在字节数较多时其性能仍旧比unroll实现要好。

四、小结

lab3到此就结束了，受时间限制，没有进行绘图。如果要绘图的话我可能会把结果暂时存在文件中，然后用Python读取文件，并利用第三方库画图。实验中用到的中间结果已经删去或者注释掉，希望不会对自动批阅系统与助教老师的批阅造成困扰。