实验一 使用 gem5 对 benchmark 做性能分析

1. 实验目的

1. 学习通过程序运行数据对程序和执行程序的硬件做性能分析
2. 加深对不同硬件配置对程序性能影响的理解
3. 熟悉gem5模拟运行程序的流程

2. 实验内容

2.1 配置并测试

本实验需要对 **7 种配置** 各运行 **5 个 benchmark**。配置为（添加一行作为“配置号”）：

配置号	CPU_type	Issue width	CPU_clock	L2 cache
0	DerivO3CPU	8	1GHz	No
1	MinorCPU	-	1GHz	No
2	DerivO3CPU	2	1GHz	No
3	DerivO3CPU	8	4GHz	No
4	DerivO3CPU	8	1GHz	256kB
5	DerivO3CPU	8	1GHz	2MB
6	DerivO3CPU	8	1GHz	16MB

使用 gem5 自带的配置文件 `se.py`，针对上述配置，编写测试脚本，完成测试。

运行如下命令打印所有可能的配置选项：

```
build/X86/gem5.opt configs/example/se.py --help
```

其中比较重要的选项如下：

选项	作用
`-cpu-type=CPU_TYPE`	要运行的 cpu 类型
`-cpu-clock=CPU_CLOCK`	以 CPU 速度运行的块的时钟
`-mem-type=MEM_TYPE`	要使用的内存类型。选项包括不同的 DDR 内存和 ruby 内存控制器。
`-caches`	使用经典缓存执行模拟。
`-l2cache`	如果使用经典缓存，请使用 L2 缓存执行仿真。
`-c CMD, --cmd=CMD`	在系统调用仿真模式下运行的二进制文件。

根据需要设置相应的配置，如：

```
build/X86/gem5.opt \
        configs/example/se1.py \
        --cpu-type=DerivO3CPU \
        --mem-type=DDR3_1600_8x8 \
        --caches \
        --l1d_size=64kB \
        --l1i_size=64kB \
        --cpu-clock=1GHz \
        --cmd=/root/Arch_Labs/Lab2/cs251a-microbench-master/lfsr
```

全部运行指令请见测试脚本 `runbenckmark.sh`

设置 Issue width:

`se.py`：没有设置 Issue Width，用于运行 Minor

`se1.py`：设置为 8，具体为在 se1.py 中添加：`CPUClass.issueWidth = 8`

`se1.py`：设置为 2

上述三个文件均位于 example 中

Makefile:

改为 python3 运行，修改 `rand_spmv_arrs.py` 中的 `xrange` 为 `range`

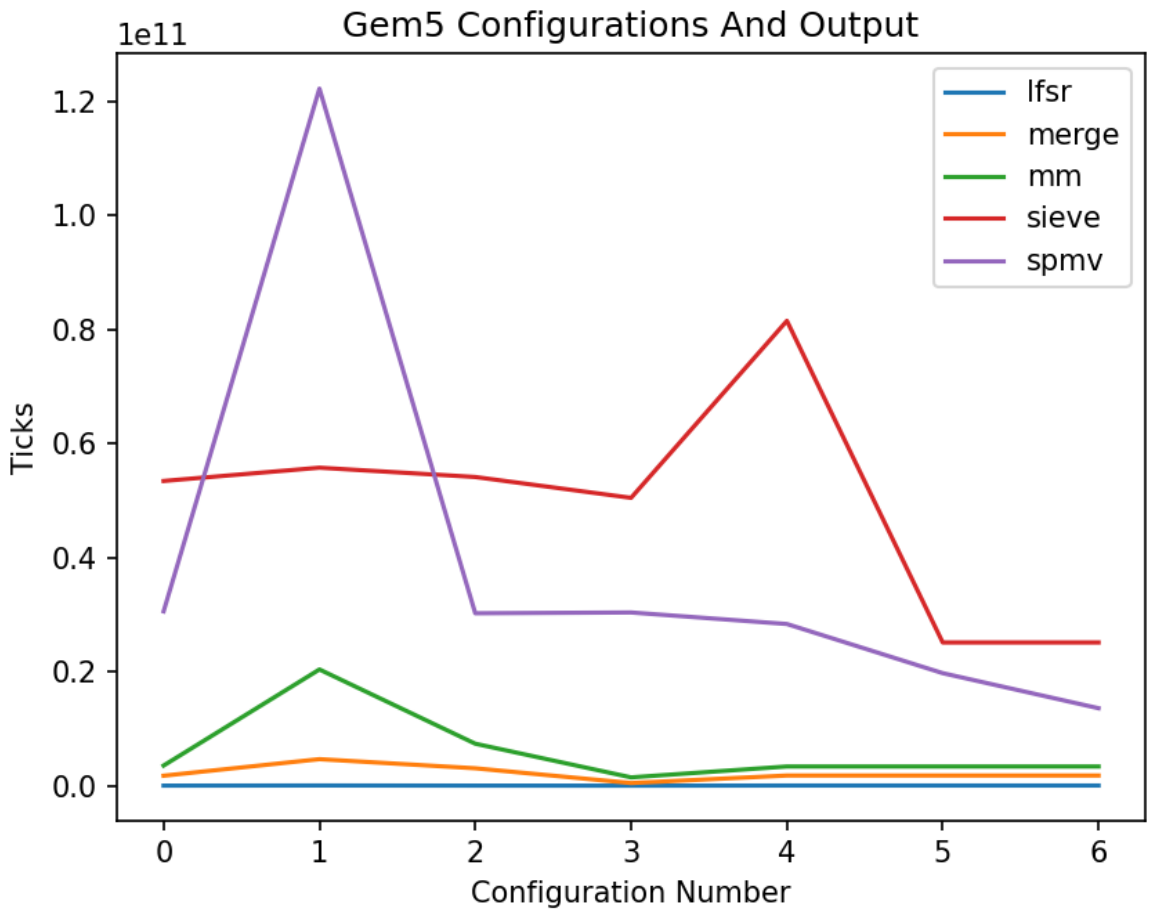
输出:

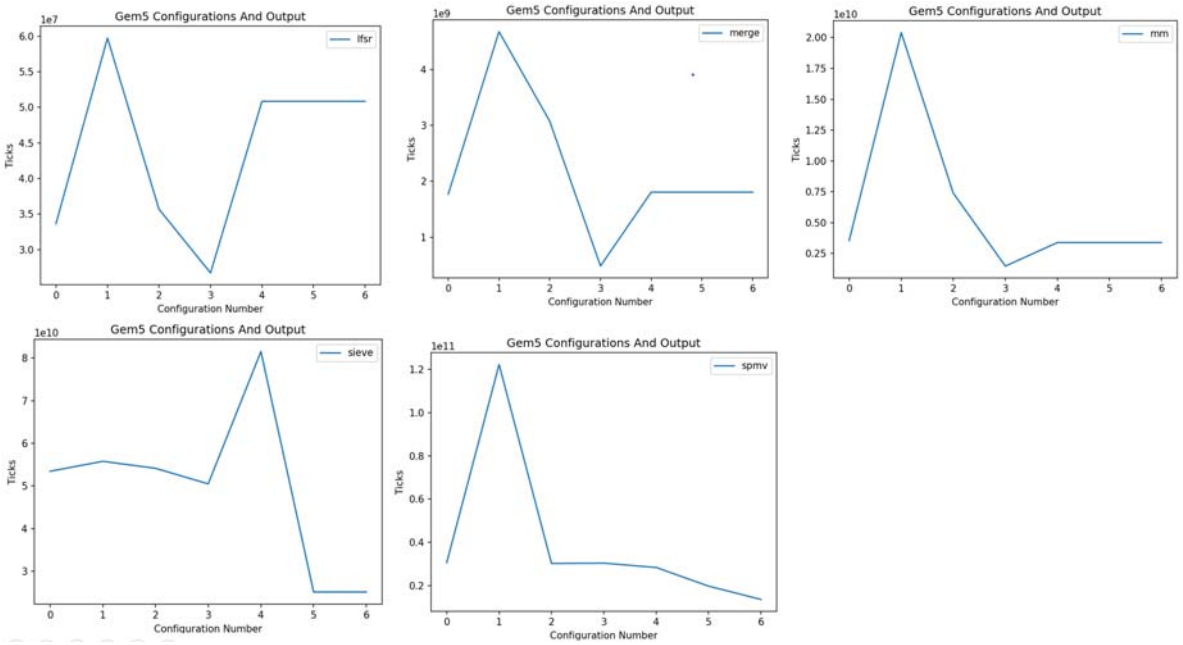
将每个输出的 m5out 文件夹复制，分别命名为：benchmark 名 + 配置号 ，其中配置号见本文档第一个表格的第一列。

2.2 运行结果（Ticks）

配置号	lfsr	merge	mm	sieve	spmv
0	32409000	1769870000	3551471000	53448207000	30593681000
1	57691000	4674926000	20402614000	55761680000	122232723000
2	34386000	3074216000	7382404000	54132727000	30254954000
3	25568500	485231500	1484017000	50489781500	30393596500
4	48947000	1805059000	3386216000	81516642000	28372334000
5	48947000	1805059000	3386216000	25130090000	19752948000
6	48947000	1805059000	3386216000	25130090000	13600672000

绘制图像如下：





2.3 回答问题

1、应该使用什么指标来比较不同系统配置之间的性能？为什么？

- 运行同一个二进制文件的**时钟周期数**和**时间**：最直观体现系统性能的指标，运行时间越短，反映出系统的性能越好。在 gem5 的统计输出文件中对应为 `simTicks` 和 `simSeconds` 。
- CPU 提交的指令数：总指令数越少，效率越高。在统计输出文件中对应为 `simInsts`.
- `hostInstRate`：表示在模拟过程中，模拟器每秒钟提交的指令数量，一个系统的 `hostInstRate` 更高，说明该系统能够更快地执行指令，具有更高的性能。
- `hostMemory`：指gem5模拟器中用于模拟的主机内存的总大小，该指标可以用来评估模拟器对于内存访问的需求，以及对于主机系统资源的占用情况。
- Cache miss rate: 若 Cache miss，那么需要更多的时间从更低一级缓存取数据，影响系统执行时间。
- Cache 大小和相联度：会影响 Cache 命中率。
- CPI: Cycle Per Instruction，表示完成单个指令所需的时钟周期数，较低的 CPI 意味着每个指令完成所需的时钟周期更少，也就意味着系统的速度更快。
- IPC: Instructions Per Cycle，每个时钟周期内所执行的指令条数，IPC 能反映出 CPU 指令集并行度，受流水线深度、cache 大小等因素的影响，可以反映出系统的性能。
- Branch 指令预测成功率：命中率越高，系统执行的效率越高。

2、是否有任何基准测试受益于删除 L2 缓存？请说明理由。

有，`lfsr` 和 `merge`，前者的效果更加明显。

当 L2 Cache 存在时，如果 L1 Cache 没有命中，CPU 会先从 L2 Cache 中查找数据，如果找不到，才会访问主存。当程序访问的数据并**不具有比较好的局部性**时，也即程序连续访问的数据位于存储空间的相隔较远的位置，导致频繁的 Cache Miss 和 Cache 换入换出，反而比不使用 L2 Cache 直接访问内存要多出很多处理操作，最终导致删除 L2 Cache 反而测试效果更好。

`lfsr` 实现了一个伪随机数生成器，并根据生成的随机数作为下标，访问结构体数组的第一个元素。首先，随机生成的下标很大概率不会相邻，其次即使相邻，因为每次访问的都是含八个整形数据的结构体元素的第一个数据，相邻结构体元素的第一个数据也不相邻，所以该程序访问的数据不具有良好的程序局部性，所以删除 L2 Cache 反而测试效果更好

3、在讨论程序的运行行为时，我们会遇到 a) memory regularity，b) control regularity，和 c) memory locality，请谈一谈你对他们的理解。

a) memory regularity：

访问内存的模式是否规律，如连续的地址空间。当 memory regularity 比较高时，可以通过预测和优化技术提高程序性能。比如访问数组时，程序大概率会按照固定的顺序进行访问，所以处理器通常采取将数组数据加载到 Cache 中，减小访问内存的次数。

b) control regularity：

描述的是程序在控制流方面的规律性。程序的执行过程中，通常会出现循环、条件分支等控制结构。当这些结构的执行具有较高的规律性时，处理器可以利用分支预测技术来提高程序性能。如果预测正确率较高，则可以大幅度提高处理器的利用率和程序性能。

c) memory locality：

内存局部性即数据访问的局部性，分为时间局部性和空间局部性。时间局部性是指程序在一段较短的时间内，可能会多次访问同一内存地址；空间局部性是指程序在访问内存时，有可能接着访问其相邻的内存。当程序具有较高的内存局部性时，可以通过缓存技术来提高程序性能。

4、对于这三个程序属性——a) memory regularity，b) control regularity，和 c) memory locality——从 stats.txt 中举出一个统计指标（或统计指标的组合），通过该指标你可以区分一个workload是否具有上述的某一个属性。（例如，对于control regularity，它与分支指令的数量成反比。但你一定可以想到一个更好的）。

a) memory regularity：

可以用 Cache 预取的有效性来衡量。如上一题中分析的，如果程序的 memory regularity 较高，那么可以通过预取减少访问内存的次数。在 gem5 的输出统计文件中，有几个指标可以体现 memory regularity：

- `system.cpu.dcache.prefetcher.pfIssued`：number of hwpf issued (Count)
- `system.cpu.dcache.prefetcher.pfUseful`：number of useful prefetch (Count)
- `system.cpu.dcache.prefetcher.accuracy`：accuracy of the prefetcher (Count)

分别为 dcache 的硬件预取指令数、有用预取指令数和预取的准确性，准确性越高，说明预取的效果越好，memory regularity 越高。

注意到，在本次实验要求的配置中，并没有对硬件预取做相应配置，查看 config.ini 可知，default 情况下硬件预取是 `false`，所以需要通过添加 `--l1d-hwp-type` 开启这一配置。

此外，Cache 的命中率也可以在一定程度上反映 memory regularity.

b) control regularity：

可以使用分支预测命中率来衡量，如前一个问题中分析的，如果一个程序的 control regularity 比较高，那么处理器的分支预测正确率往往较高。在 gem5 的输出统计文件中，有几个指标可以体现 control regularity：

- `system.cpu.branchPred.condPredicted`：Number of conditional branches predicted (Count)
- `system.cpu.branchPred.condIncorrect`：Number of conditional branches incorrect (Count)

分别为条件分支预测次数和预测不准确率，后者与前者的比例越低，说明不准确率越低，即准确率越高，说明 control regularity 越高。

c) memory locality：

可以用 Cache 命中率来衡量，如果一个程序的局部性比较好，那么其 Cache 的命中率就会比较高。

- `system.cpu.dcache.overallAccesses::total`：number of overall (read+write) accesses (Count)
- `system.cpu.dcache.overallHits::total`：number of overall hits (Count)

为 dcache 的访问次数和命中次数，后者比前者的比例越高，说明 cache 命中率越高，说明 memory locality 越高。

5、对于每一个实验中用到的 benchmark，描述它的a) memory regularity, b) control regularity, c) memory locality；解释该 benchmark 对哪个微架构参数最敏感（换句话说，你认为"瓶颈"是什么），并使用推理或统计数据来证明其合理性。

首先，我的思路是，根据第四题选取的统计指标，结合代码分析。

但是在具体分析过程中，多次出现了**指标结果和源码分析不一致**的情况，对于这一点，我认为是由于在对 C 代码编译时，开启了 `-O3` 优化，导致生成的二进制文件的结构与原本的 C 代码的**结构有很大的差别**，影响了 memory regularity、control regularity 和 memory locality，所以出现了上述不一致的情况。

为了验证这个猜测，我将 Makefile 的 `-O3` 优化取消，并用 gem5 进行测试。结果如下。

上述提到的指标列举如下（仅列举了配置号为5，即带 L2 Cache 且其大小为 2MB 的系统配置，实际是参考了各输出文件，配置号为 5 的指标输出最具有代表性）：

benchmark	prefetcher.accuracy	branchPred.condPredicted	branchPred.condIncorrect	overallAccesses::total	overallHits::total
`lsfr`	0.049753	502941	696	4504272	4012078
`merge`	0.440071	763092	74909	3278409	3277445
`mm`	0.069994	2691678	980	31003126	7184812
`sieve`	0.560310	3207254	1059	18000846	17486457
`spmv`	0.091531	1703509	25452	21155693	5128206

根据上述结果，结合代码分析如下(**以下分析是没有开 `-O3` 的二进制文件，非本次实验要求的 benchmark**):

benchmark	代码内容	memory regularity	control regularity	memory locality
`lsfr`	`lfsr` 实现了一个伪随机数生成器，并根据生成的随机数作为下标，访问结构体数组的第一个元素	**低：**硬件预取准确率低。访问的数组下表是伪随机生成的，不具有连续性。	**很高**：分支预测的准确率约为 99%，没有复杂的跳转结构，主要包含一个 while 循环	**低：**访问的数组下标是伪随机的，所以刚访问的数据大概率不会被重复访问，且其相邻位置也大概率不会被访问，所以局部性不好。（*）
`merge`	归并排序	**高：** 硬件预取的准确率很高。访问的数据基本上是连续的数组元素。	**较高：** 分支预测的命中率比较高，但是在五个程序里是最低的。分支主要为 while 循环和函数调用，而 while 循环除了退出时，其他时间都是 taken	**高：** 有非常高的 Cache 命中率，约为 99.97%，分析代码可知，访问的数据通常是连续的数组元素，有比较好的局部性。

benchmark	代码内容	memory regularity	control regularity	memory locality
`mm`	矩阵乘法，使用了分块技术	**低：** 分别分块访问每个矩阵的元素，三个数组轮流访问且访问的下标也不相同。	**高：** 分支预测命中率非常高。代码中的控制流程非常规律。有三层外部循环以及三层内部循环。这种多层嵌套循环结构导致代码执行的控制流程清晰而且规律。	**低：** Cache 的命中率很低，由于最终的计算指令需要同时访问三个数组。
`sieve`	埃氏筛	**高：** 硬件预取的命中率较高。访问数组的主要有两个 for 循环，访问的下标均为每次加 1，规律性强。	**高：** 分支预测的准确率非常高。这段带代码的控制流程主要是两个 for 循环，控制规律性都比较强。	**高：** Cache 的命中率很高。由于两个 for 循环基本都是顺序访问 notprime 数组，所以数据局部性较好。
`spmv`	稀疏矩阵与向量相乘	**低：**硬件预取的命中率较低。由于访问每一行是第一个非零元，所以对于内层循环的 j 不确定。所以访问 val[] 和 vec[] 数组都会频繁出现跨步的情况，规律性差。	**高：** 分支预测的正确率高。这段代码的控制流主要体现在一个两重 for 循环上，循环变量每次都增加 1，所以控制的规律性很强。	**低：** Cache 的命中率较低。问 val[] 和 vec[] 数组都会频繁出现跨步的情况，所以局部性差。

说明： **(*)** 由于 L1 dCache 比较大，而且该代码的数组比较小，所以导致 Cache 的命中率比较高。经过测试，当 L1 dCache 的大小减小时，`lsfr` 的 Cache 命中率显著减小：当取消 L2 Cache，将 L1 dCache 由 64 KB 减为 256 B 时，`lsfr` 的 Cache 命中率下降为约原来的一半，相比之下，`merge` 的 Cache 命中率基本不变。所以此处命中率高并不能说明该代码的局部性好。

此时发现**不开 `-O3` 优化时，指标结果和源码分析基本一致。**

之后，对于**开启了 `-O3` 优化**的二进制文件，也即本次实验原本设计的 benchmark，再次进行 gem5 测试，结果如下：（仅列举了配置号为5，即带 L2 Cache 且其大小为 2MB 的系统配置，实际是参考了各输出文件，配置号为 5 的指标输出最具有代表性）

benchmark	prefetcher.accuracy	branchPred.condPredicted	branchPred.condIncorrect	overallAccesses::total	overallHits::total
`lsfr`	0.318777	3009	693	4031	3789
`merge`	0.435092	787973	59035	725573	724682
`mm`	0.065869	297933	988	2888162	2743203
`sieve`	0.491525	2278126	1081	2344071	1821393
`spmv`	0.285135	1691008	25469	4941278	3331127

因为开启了优化， 所以以下分析根据主要上述指标结果：

benchmark	代码内容	memory regularity	control regularity	memory locality
`lsfr`	`lfsr` 实现了一个伪随机数生成器，并根据生成的随机数作为下标，访问结构体数组的第一个元素	**高：**硬件预取的准确路较高。	**较高**：分支预测的准确率约为 76%。	**低：**由上面的（*）可知。
`merge`	归并排序	**高：** 硬件预取的准确路较高。	**较高：** 分支预测的命中率比较高。	**高：** 有非常高的 Cache 命中率，约为 99.88%。
`mm`	矩阵乘法，使用了分块技术	**低：** 硬件预取的准确率较低。	**高：** 分支预测命中率非常高。	**高：** Cache 的命中率很高。
`sieve`	埃氏筛	**高：**硬件预取的准确率很高。	**高：**分支预测命中率非常高。	**较高：** Cache 的命中率较高
`spmv`	稀疏矩阵与向量相乘	**高：** 硬件预取的准确路较高。	**高：**分支预测命中率高。	**较高：** Cache 的命中率较高。

通过与不开 `-O3` 的结果相比较，发现开启 `-O3` 优化后，以下几处的运行指标发生了显著改变：

- `lsfr` 的硬件预取准确性由低变高。
- `mm` 的 Cache 命中率由低变高。
- `spmv` 的硬件预取准确性由低变高。
- `spmv` 的 Cache 命中率由低变高。

对哪个微架构最敏感：

benchmark	微架构	原因
`lsfr`	L1 dCache 的大小	由（*）可知，dCache 的大小会显著影响该 benchmark 的 Cache 命中率
`merge`	分支预测	该 benchmark 的分支预测准确率为五个中的最低，若能通过选择更有效的分治策略提升预测准确率，则能显著提升其性能。
`mm`	L1 dCache 的大小	该程序使用了三个数组，且会频繁访问三个数组元素，规律性较低，如果能将数组的更多元素放入 dCache 中，则可以减少访存时间。
`sieve`	L1 dCache 的大小，以及 L2 Cache 的延迟	该程序创建的数组较大，但访存非常频繁，若可以提高 L1 dCache 的大小或者降低 L2 Cache 的延迟，则可以降低访存时间。
`spmv`	L1 dCache 的大小，以及 L2 Cache 的延迟	该程序的空间需求大，访存频繁。经测试 Cache 命中率与 L1 dCache 大小有关联但是并不强，这意味着会有较多的 L1 Cache 的换入换出，所以 L2 Cache 的延迟很关键。此外根据测试，L1 dCache 的大小对于最终时钟周期数影响较大。

6、选择一个 benchmark，提出一种你认为对该 benchmark 非常有效的应用程序增强、ISA 增强和微体系结构增强。

选择 `sieve`，这是一个埃氏筛，输出 1000000 以内的素数个数。

通过两个嵌套的 for 循环来实现埃氏筛。外部循环从 2 开始迭代，直到 n 的平方根。内部循环更新当前数字 (p) 的所有倍数，通过在 "notprime" 数组中将它们标记为非素数。

应用程序增强：

- 可以采用效率更高的欧拉筛，相对于朴素的埃氏筛，欧拉筛不会重复标记一个数是不是素数的倍数，时间可以从埃氏筛的 $O(nloglogn)$ 优化到欧拉筛的 $O(n)$

- 也可以考虑采用 MillerRabbin 算法，该算法本质上是一种随机化算法，能在 $O(log^3n)$ 的时间复杂度下快速判断出一个数是否是素数，但具有一定的错误概率。不过在一定数据范围内，通过一些技巧可以使出错概率非常低。
- 对于该欧拉筛代码本身的优化，可以考虑一下几个小点：
  - 外层循环 `for (int p=2; p*p<=n; p++)`，本质上是枚举 p 从 2 到根号 n，但是这样写会在每次循环计算一次乘法，所以可以在外层先计算根号 n，在循环条件上只需要 p 小于已经计算出的根号 n 即可。
  - 内层循环 `for (int i=p*2; i<=n; i += p)`，对于大于 2 的素数 p，其倍数 $2*p$ 已经在分析素数 2 时被标记过，甚至所有小于 p 的素数 q，$p*q$ 也已经在 p 之前标记过，所以这一层循环可以从 $p*p$ 开始。

ISA 增强：

- 可以使用 SIMD，对于 `for(int p = 2; p < n; ++p) { total+=!notprime[p]; }`，求和之间没有数据依赖，可以在一个时钟周期内计算多个整型数据的和。

微系体结构增强：

- 根据上面的图表可知，增加 L2 Cache 的容量可以有效缩减程序运行时间，提升效率，所以可以增加大容量的 L2 Cache，以及降低 L2 Cache 的延迟。

3. 附录

- `se.py`：没有设置 Issue Width，用于运行 Minor
- `se1.py`：设置为 Issue Width = 8
- `se2.py`：设置为 Issue Width = 2
- `runbenchmarkvm.sh`：测试运行的脚本
- `runbenchmarkvm-prefetcher.sh`：测试运行的脚本，加入了硬件预取
- `ouput`：测试输出的 m5out，该文件夹下有 35 个子文件夹，命名为 benchmark 名 + 系统配置编号，系统配置编号如下：

配置号	CPU_type	Issue width	CPU_clock	L2 cache
0	DerivO3CPU	8	1GHz	No
1	MinorCPU	-	1GHz	No
2	DerivO3CPU	2	1GHz	No
3	DerivO3CPU	8	4GHz	No
4	DerivO3CPU	8	1GHz	256kB
5	DerivO3CPU	8	1GHz	2MB
6	DerivO3CPU	8	1GHz	16MB