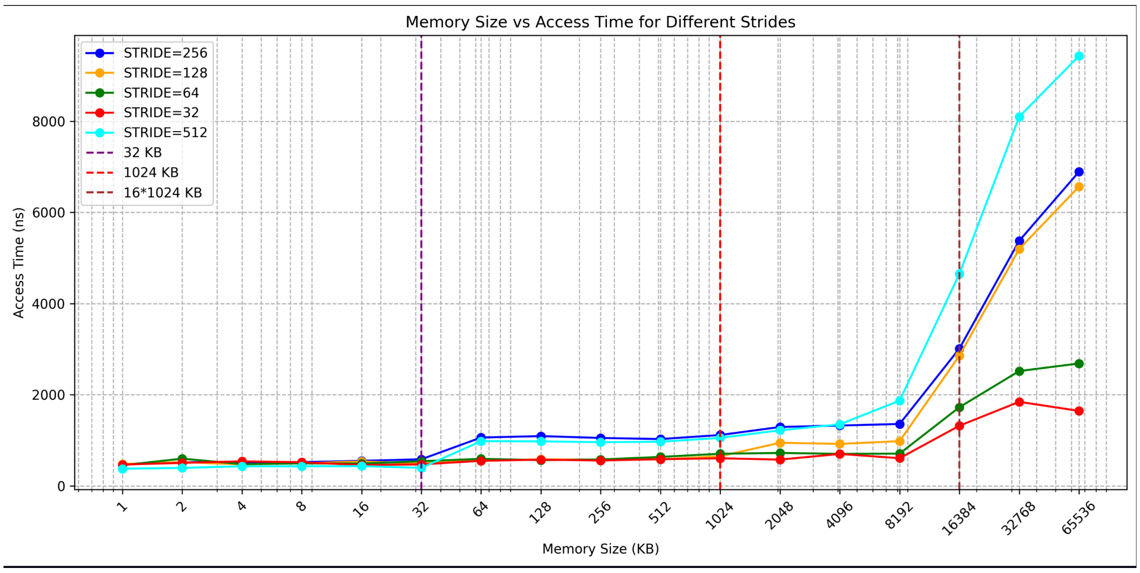
缓存测量实验报告

宋建昊 2022010853

步骤 1（Cache Size）

- 访存序列：分配内存数组的索引序列为a[(n*stride) mod size]，n为循环访问轮次取值为0-ITERATIONS-1
- 测试不同步长不同访存大小的访问时间。
- 执行结果

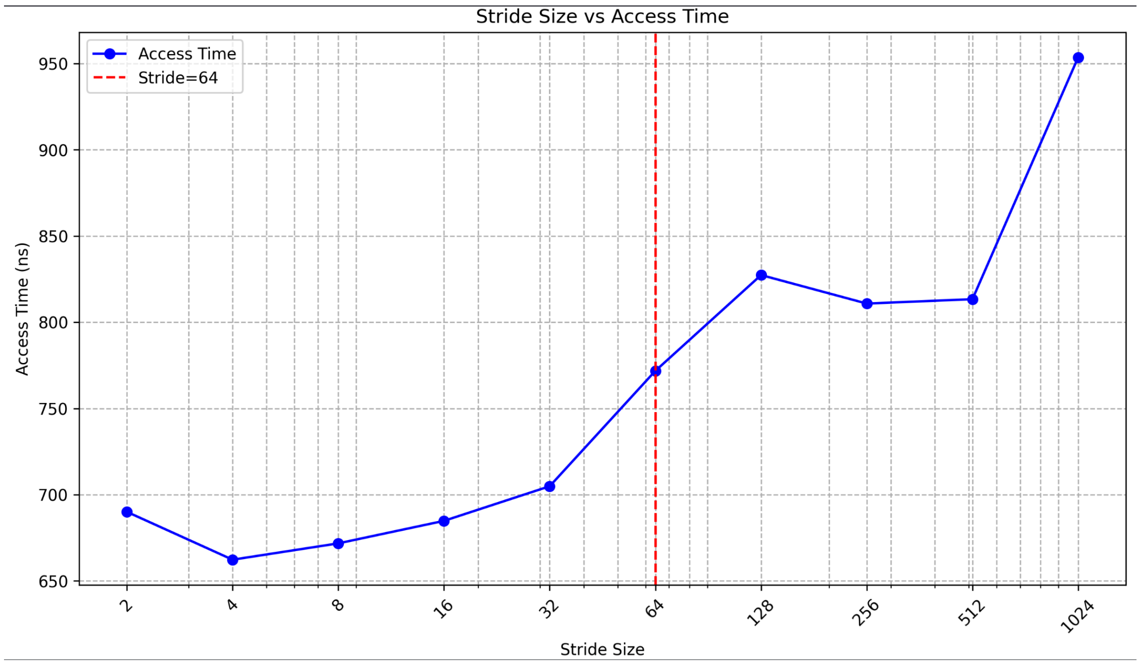


- 结果分析：可以看到在三条虚线（对应三级Cache，通过命令行查看的准确的cpu参数，32KB/1MB/16MB）处访问时间由于Cache Miss发生阶跃，符合预期的实验现象。
- 思考题：理论上 L2 Cache 的测量与 L1 DCache 没有显著区别。但为什么 L1 DCache 结果匹配但是 L2 Cache 不匹配呢？你的实验有出现这个现象吗？请给出一个合理的解释。
- 确实在L1 DCache处发生的阶跃非常明显但是L2 Cache处的阶跃程度明显更低。原因可能是L2 Cache为数据和指令共用的cache，有部分空间缓存了指令，导致了这个问题，而 L1 DCache只保存数据，从而和实验预期的阶跃符合得更好。

步骤 2（Cache Line Size）

- 访存序列：分配内存数组的索引序列为a[(n*stride) mod size]，n为循环访问轮次取值为0-ITERATIONS-1
- 步长 stride_sizes = { 2,4,8,16,32,64,128,256,512,1024 }，测试不同步长的访存时间

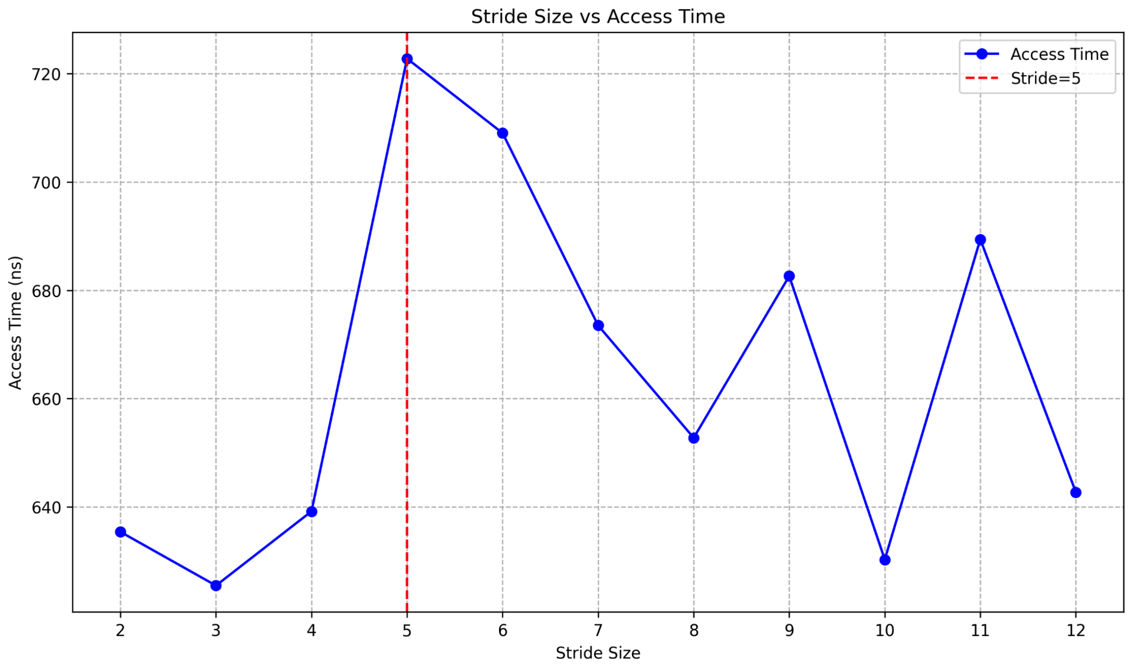
- 执行结果



- 结果分析：在访问步长64开始有明显访问时间的延长，原因是相邻的两次访存不在同一个cache line中，和cpu参数中查找得到的缓存行大小为64B对应，实验结果符合预期。
- 思考题：Prefetch 对于该实验的结果有影响吗？给出你的分析和结论。
- 我认为Prefetch 对实验结果确实有影响，因为现代处理器的预取机制会根据访问模式提前加载 Cache Line，从而减少部分 Cache Miss 的开销，尤其是在步长规律且适中时效果显著，比如步长略大于 Cache Line 大小的情况下，预取器可能预测并加载后续数据，使运行时间低于理论上的“每次都 Miss”预期，而在步长很小或非常大时，影响则会减弱，因为小步长已命中较多，大步长可能超出预取范围，加之循环访问的回绕可能打断预测连续性，导致效果不一，为此可以禁用 Prefetch 从而更准确地分析缓存行为。

步骤 3（Cache Associativity）

- 访存序列：设 `L1DCache` 大小为 size，区间划分数num=2的幂次， 访存序列为 a[(4sizen*num) mod 2size]
- 采取实验思路一，num = { 2,4,8,16,32,64,128,256,512,1024 }，测试不同划分块数的访问时间。
- 执行结果



- 结果分析：我们采取实验思路一，从 I=2的五次方开始，访存时间显著增加，根据实验一方法得到的缓存相连度为2的3次方=8，和cpu参数中查找得到的缓存行相连度为8对应，实验结果符合预期。
- 相联度算法分析：实验思路一将数组分为2的n次方块并只访问其中的奇数块，随着n逐渐增大，访问的块数（n个奇数块）会逐渐超过缓存的组数与相联度的乘积。当n增加到某个临界值时，缓存无法容纳所有访问的块，导致冲突未命中，访问时间会显著变慢。此时，2的n-2次方（即前一个n值对应的块数减去未使用的偶数块影响）可以看作缓存的相联度。

步骤 4（MatMul Optimization）

- 进行了如下的代码修改。

-
  ```
      // TODO: Your optimized code:
      //==========================================================
      constexpr size_t BLOCK_DIM = 128;

      // 计算总块数
      constexpr size_t NUM_BLOCKS = MATRIX_SIZE / BLOCK_DIM;

      // 主计算循环
      for (register size_t block_idx = 0; block_idx < NUM_BLOCKS; ++block_idx)
  {
          register size_t start_col = block_idx * BLOCK_DIM;
          register size_t end_col = start_col + BLOCK_DIM;

          // 按行处理
          for (register size_t row = 0; row < MATRIX_SIZE; row++) {
              // 缓存行首地址
              const int* a_row = a[row];
              register int* d_row = d[row];

              // 块内循环展开（每次处理4个元素）
              register size_t col = start_col;
              for (; col + 3 < end_col; col += 4) {
                  register int temp0 = a_row[col];
                  register int temp1 = a_row[col + 1];
                  register int temp2 = a_row[col + 2];
                  register int temp3 = a_row[col + 3];

                  // 内部矩阵乘法
                  for (register size_t k = 0; k < MATRIX_SIZE; k++) {
                      d_row[k] += temp0 * b[col][k] +
                          temp1 * b[col + 1][k] +
                          temp2 * b[col + 2][k] +
                          temp3 * b[col + 3][k];
                  }
              }

              // 处理剩余元素
              for (; col < end_col; col++) {
                  register int temp = a_row[col];
                  for (register size_t k = 0; k < MATRIX_SIZE; k++) {
                      d_row[k] += temp * b[col][k];
                  }
              }
  ```

```
            }
        }
        // Stop here.
        //=======================================================
```

- **分块（Blocking）优化**

```
constexpr size_t BLOCK_DIM = 128;
constexpr size_t NUM_BLOCKS = MATRIX_SIZE / BLOCK_DIM;
for (size_t block_idx = 0; block_idx < NUM_BLOCKS; ++block_idx) {
    size_t start_col = block_idx * BLOCK_DIM;
    size_t end_col = start_col + BLOCK_DIM;
    // 块内计算
}
```

  - **缓存局部性**：现代CPU有层次化的缓存（如L1、L2、L3），分块将矩阵划分成较小的子块（这里每块宽度为128列），使得每个子块的数据更可能在访问时驻留在高速缓存中。
    - 未分块时，矩阵乘法可能导致频繁的Cache Miss（尤其是当MATRIX_SIZE很大时），因为每次迭代访问的内存跨度较大。
    - 分块后，每次只处理一个子块（BLOCK_DIM列），减少了跨行访问的范围，提高了空间局部性（Spatial Locality）和时间局部性（Temporal Locality）。
  - **减少内存访问延迟**：通过限制计算范围，数据可以被重复利用。例如，a[row][col]和b[col][k]在块内多次使用时，可能仍留在Cache中，减少从主存加载的开销。
  - **参数选择**：BLOCK_DIM = 128通常是根据Cache行大小（常见为64字节）或L1/L2 Cache容量经验性选择的，确保块数据适合缓存。

- **循环展开（Loop Unrolling)**

```
for (; col + 3 < end_col; col += 4) {
    register int temp0 = a_row[col];
    register int temp1 = a_row[col + 1];
    register int temp2 = a_row[col + 2];
    register int temp3 = a_row[col + 3];
    for (size_t k = 0; k < MATRIX_SIZE; k++) {
        d_row[k] += temp0 * b[col][k] +
                    temp1 * b[col + 1][k] +
                    temp2 * b[col + 2][k] +
                    temp3 * b[col + 3][k];
    }
}
```

- **减少循环开销**：展开循环减少了条件判断和跳转的次数。未展开时，每迭代一次需要检查col < end_col并更新col，而展开后每次处理4个元素，循环次数减少到原来的1/4。
- **减少内存访问冗余**：将a_row[col]到a_row[col + 3]提前加载到寄存器（temp0到temp3），在k循环中重复使用，避免反复从内存读/取。

实验结果

```
//循环进行了二十次实验，每次加速比都大于3，平均加速比达到4.95，具体数据如下.
time spent for original method : 3.289 s
time spent for new method : 0.47 s
time ratio of performance optimization : 6.99787
test 1 passed
time spent for original method : 3.591 s
time spent for new method : 0.541 s
time ratio of performance optimization : 6.63771
test 2 passed
time spent for original method : 3.925 s
time spent for new method : 0.827 s
time ratio of performance optimization : 4.74607
test 3 passed
time spent for original method : 3.815 s
time spent for new method : 0.914 s
time ratio of performance optimization : 4.17396
test 4 passed
time spent for original method : 3.749 s
time spent for new method : 0.809 s
time ratio of performance optimization : 4.63412
test 5 passed
time spent for original method : 3.844 s
time spent for new method : 0.702 s
time ratio of performance optimization : 5.47578
test 6 passed
time spent for original method : 3.729 s
time spent for new method : 0.864 s
time ratio of performance optimization : 4.31597
test 7 passed
time spent for original method : 3.546 s
time spent for new method : 0.545 s
time ratio of performance optimization : 6.50642
test 8 passed
time spent for original method : 3.686 s
time spent for new method : 0.749 s
time ratio of performance optimization : 4.92123
test 9 passed
time spent for original method : 3.545 s
time spent for new method : 0.542 s
time ratio of performance optimization : 6.54059
test 10 passed
time spent for original method : 3.755 s
time spent for new method : 0.894 s
time ratio of performance optimization : 4.20022
test 11 passed
time spent for original method : 4.133 s
time spent for new method : 0.938 s
time ratio of performance optimization : 4.40618
test 12 passed
time spent for original method : 4.881 s
time spent for new method : 0.859 s
time ratio of performance optimization : 5.68219
test 13 passed
```

```
time spent for original method : 4.657 s
time spent for new method : 0.897 s
time ratio of performance optimization : 5.19175
test 14 passed
time spent for original method : 3.893 s
time spent for new method : 0.774 s
time ratio of performance optimization : 5.02972
test 15 passed
time spent for original method : 3.849 s
time spent for new method : 0.822 s
time ratio of performance optimization : 4.68248
test 16 passed
time spent for original method : 3.706 s
time spent for new method : 0.838 s
time ratio of performance optimization : 4.42243
test 17 passed
time spent for original method : 3.806 s
time spent for new method : 0.907 s
time ratio of performance optimization : 4.19625
test 18 passed
time spent for original method : 3.695 s
time spent for new method : 0.781 s
time ratio of performance optimization : 4.73111
test 19 passed
time spent for original method : 4.22 s
time spent for new method : 0.922 s
time ratio of performance optimization : 4.57701
test 20 passed
//20次实验全部通过
score: 20/20
//原始程序平均时间
average time spent for original method : 3.8657 s
//优化后平均时间
average time spent for new method : 0.77975 s
//平均加速比
average time ratio of performance optimization : 4.95761
```

（选做）测量缓存的写策略和替换策略

写策略检测实验

实验原理与思路

构造访问序列，通过测量L1数据缓存（L1 DCache）在Write Hit和Write Miss两种情况下的写操作延迟，来推断其写策略是Write-through还是Write-back。

- **验证写策略**：
  - Write-through：写操作同时更新Cache和主存，延迟较高，Hit和Miss的延迟差异较小。
  - Write-back：写操作仅更新Cache，延迟较低，Hit和Miss的延迟差异较大。
- **测量方法**：通过高精度计时器（RDTSC）测量写操作的周期数，比较Write Hit和Write Miss的平均延迟。
- **先Miss后Hit的顺序**：先测试Miss避免数组初始化或预热影响结果，清空Cache后测试Hit，确保干净的初始状态。

实验结果分析

- 若hit_time远低于miss_time（如10 vs 50 cycles），表明Write-back策略。
- 若两者接近（如30 vs 35 cycles），表明Write-through策略。
- 二者差距的比例阈值选为0.7.
- 实验输出

  ```
  Using RDTSC:
  Average Write Hit latency: 89 cycles
  Average Write Miss latency: 90 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 88 cycles
  Average Write Miss latency: 89 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 88 cycles
  Average Write Miss latency: 93 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 92 cycles
  Average Write Miss latency: 92 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 90 cycles
  Average Write Miss latency: 95 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 91 cycles
  Average Write Miss latency: 91 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 88 cycles
  Average Write Miss latency: 94 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 88 cycles
  Average Write Miss latency: 90 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 89 cycles
  Average Write Miss latency: 93 cycles
  Likely Write-through: Hit and Miss latencies similar.
  Using RDTSC:
  Average Write Hit latency: 89 cycles
  Average Write Miss latency: 91 cycles
  Likely Write-through: Hit and Miss latencies similar.
  ```

- 在十次实验中，Miss和Hit的时间都相差细微，远小于判断阈值，所以我们推测CPU的写策略是Write-through。

替换策略检测实验

实验原理与思路

- 推断L1 DCache的替换策略（LRU、FIFO、OPT或其他）：
- LRU：替换最久未使用的行。
- FIFO：替换最早进入的行。
- OPT：替换未来最远使用的行
- Miss-like测试：构造冲突：访问array[0], array[SETS], ..., array[SETS * 8]（9次，超过8路），填满同一组并触发替换。
- 测量array[0]：若array[0]被替换（取决于策略），访问将触发Miss，延迟较高。
- 延迟对比：若Miss-like延迟远高于Hit（如2倍以上），说明array[0]被替换，策略符合假设。若延迟接近，可能为随机替换或假设不符。

实验结果分析

```
Testing L1 DCache Replacement Strategy (32KB, 8-way):
Testing LRU behavior (Miss-like test)...
Testing Cache Hit baseline...
Median Hit latency: 76 cycles
Median Miss-like latency (LRU): 77 cycles
Possibly Random or mismatch: No clear pattern.
Testing FIFO behavior (Miss-like test)...
Testing Cache Hit baseline...
Median Hit latency: 71 cycles
Median Miss-like latency (FIFO): 76 cycles
Possibly Random or mismatch: No clear pattern.
Testing OPT behavior (Miss-like test)...
Testing Cache Hit baseline...
Median Hit latency: 73 cycles
Median Miss-like latency (OPT): 76 cycles
Possibly Random or mismatch: No clear pattern.
```

最终实验结果和预期三种替换方式都不符合，所以推测CPU Cache替换策略大概率是某种其他的策略（比如LFU）或者某种混合策略。

实验机器参数

```
//lscpu --cache
songjh22@Vincent:/mnt/c/Users/24463/Desktop/HW1$ lscpu --cache
NAME ONE-SIZE ALL-SIZE WAYS TYPE        LEVEL  SETS PHY-LINE COHERENCY-SIZE
L1d       32K     256K    8 Data            1    64        1             64
L1i       32K     256K    8 Instruction     1    64        1             64
L2         1M       8M    8 Unified         2  2048        1             64
L3        16M      16M   16 Unified         3 16384        1             64
//lscpu
songjh22@Vincent:/mnt/c/Users/24463/Desktop/HW1$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits virtual
  Byte Order:             Little Endian
```

```
CPU(s):                   16
  On-line CPU(s) list:    0-15
Vendor ID:                AuthenticAMD
  Model name:             AMD Ryzen 7 7840HS w/ Radeon 780M Graphics
    CPU family:           25
    Model:                116
    Thread(s) per core:   2
    Core(s) per socket:   8
    Socket(s):            1
    Stepping:             1
    BogoMIPS:             7584.88
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt
pdpe1gb rdts
                          cp lm constant_tsc rep_good nopl tsc_reliable
nonstop_tsc cpuid extd_apicid tsc_known_freq pni pclmulqdq ssse3 fma cx16 sse4_1
sse4_2 movbe
                          popcnt aes xsave avx f16c rdrand hypervisor lahf_lm
cmp_legacy svm cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw topoext
perfctr_core
                          ssbd ibrs ibpb stibp vmmcall fsgsbase bmi1 avx2 smep
bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb
avx512cd
                          sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves
avx512_bf16 clzero xsaveerptr arat npt nrip_save tsc_scale vmcb_clean flushbyasid
de
                          codeassists pausefilter pfthreshold v_vmsave_vmload
avx512vbmi umip avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg
avx512_vpopc
                          ntdq rdpid
Virtualization features:
  Virtualization:         AMD-V
  Hypervisor vendor:      Microsoft
  Virtualization type:    full
Caches (sum of all):
  L1d:                    256 KiB (8 instances)
  L1i:                    256 KiB (8 instances)
  L2:                     8 MiB (8 instances)
  L3:                     16 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-15
Vulnerabilities:
  Gather data sampling:   Not affected
  Itlb multihit:          Not affected
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Not affected
  Reg file data sampling: Not affected
  Retbleed:               Not affected
  Spec rstack overflow:   Vulnerable: Safe RET, no microcode
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer
sanitization
```

```
  Spectre v2:             Mitigation; Retpolines; IBPB conditional; IBRS_FW;
STIBP conditional; RSB filling; PBRSB-eIBRS Not affected; BHI Not affected
  Srbds:                  Not affected
  Tsx async abort:        Not affected
```

实验建议

测量实验基于对Cache系统的很简单的建模和抽象，但感觉现在CPU有很多复杂的功能设计，通过同学们现有的知识和代码水平无法屏蔽掉这些设计带来的干扰，时常会导致实验结果不准确，可以多普及这方面的知识。