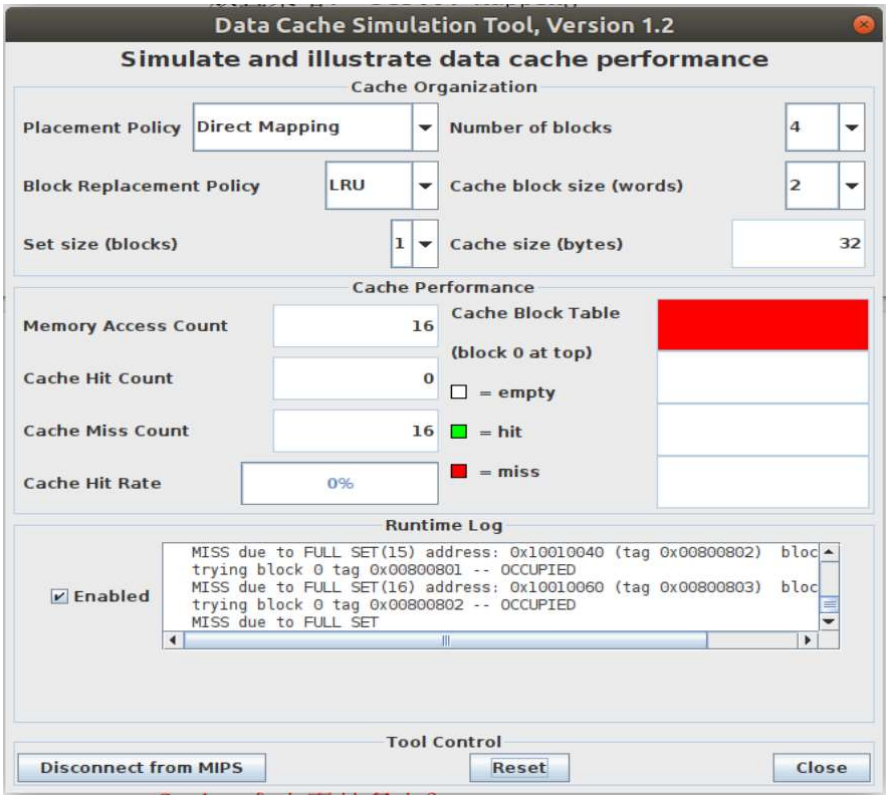
Lab6 - Cache Lab

UNIkeEN

本次实验的测试环境为 VMware 17 下的 Ubuntu18.04，双核，4+40GB

Exercise 1: Cache Visualization

场景1



- **Cache 命中率是多少?**

  如图所示，命中率为 0%

- **为什么会出现这个 cache 命中率?**

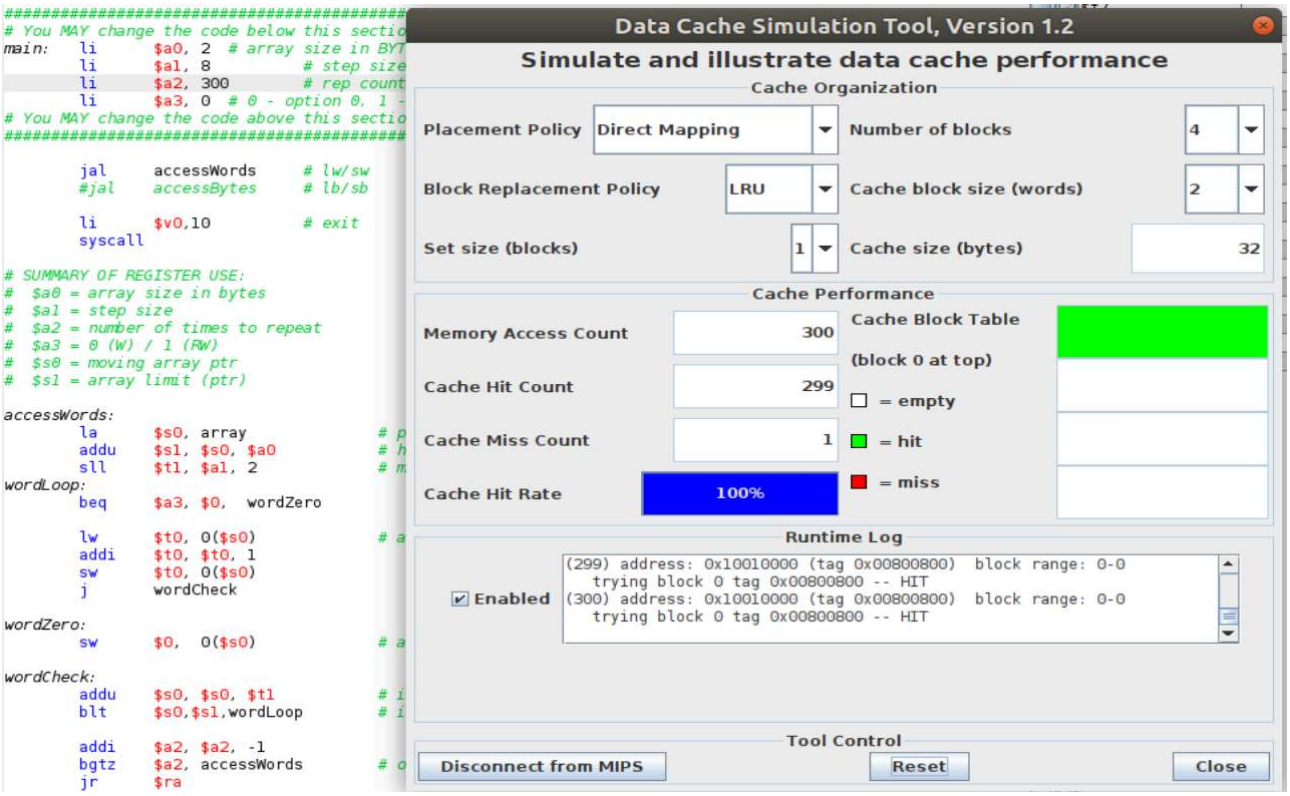
  阅读 Pseudo Code 可知，当访问下标步长为 8 words 时，相邻两次访问的地址相差 32 bytes，不在同一个 block 内，同时下一次访问会替换掉上一次访问的 cache，所以每次访问都 miss。

- **增加 Rep Count 参数的值，可以提高命中率吗？为什么?**

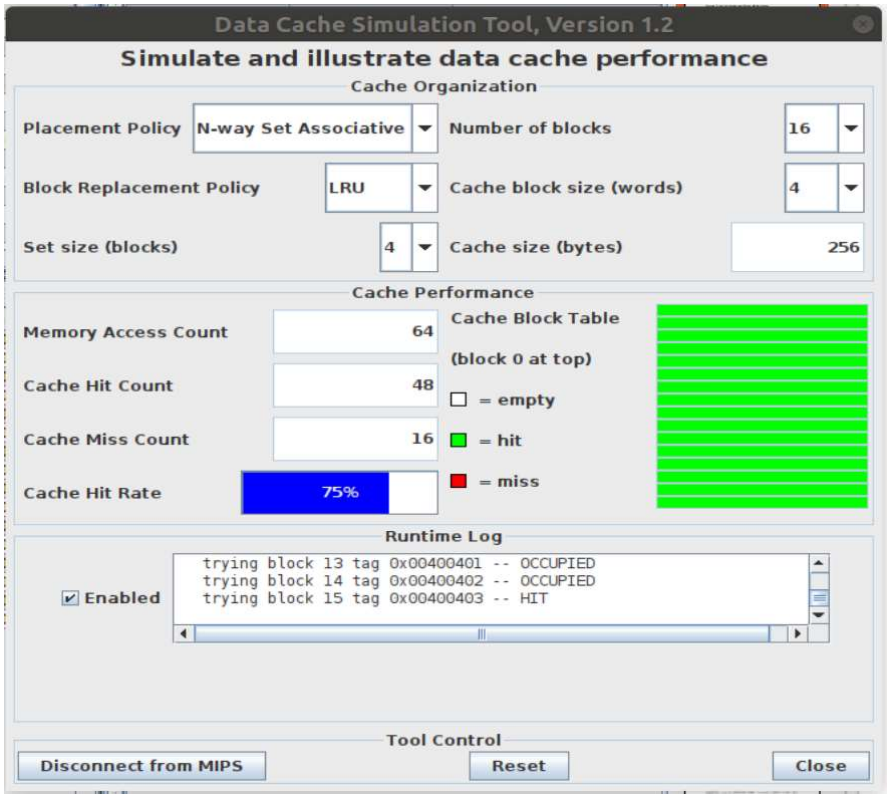
  不会。相邻两次访问的地址相差 32 bytes，刚好为 cache 大小。直接映射下，每次访问都会映射到 cache 中的同一个 block（本例中为第一个），每次访问都会替换掉上一次访问的 cache，重复访问并不能提高命中率。

- **为了最大化 hit rate，在不修改 cache 参数的情况下，如何修改程序中的参数最大化 hit rate?**

  将 array size 修改为 2，4，8，16 或 32 bytes；则每次访问的都是同一个地址，仅第一次访问时缓存失效，之后全部命中。此时增加重复次数即可使命中率趋于 100%，如下图所示。



场景2



- **Cache 命中率是多少?**

  如图所示，命中率为 75%

- **为什么会出现这个 cache 命中率?**

  相邻两次访问地址相差 8 bytes，前 4 次访问对 set 0 block 0 连续访问 4 次，仅第一次 miss；5-8次访问对 set 1 block 0 连续访问 4 次，仅第一次 miss... 以此类推，计算可得最后 4 次访问刚好填充 cache 的最后一块（set 3 block 3）；则每个 block 都经历 1 次 miss 和 3 次 hit，总命中率 75%。

- **增加 Rep Count 参数的值，例如重复无限次，命中率是多少？为什么?**

  命中率趋于 100%。Cache 的空间刚好足够容纳整个数组，则访问过程钟不会发生 cache 替换，第二次及以后重复访问时，每次访问都将 hit。

Exercise 2: Loop Ordering and Matrix Multiplication

```
(base) unikeen@ubuntu:~/Code/CS2305/
gcc -o matrixMultiply -ggdb -Wall -p
./matrixMultiply
ijk:    n = 1000, 2.548 Gflop/s
ikj:    n = 1000, 21.443 Gflop/s
jik:    n = 1000, 2.483 Gflop/s
jki:    n = 1000, 0.371 Gflop/s
kij:    n = 1000, 18.766 Gflop/s
kji:    n = 1000, 0.369 Gflop/s
```

- 1000-1000 **的矩阵相乘,哪种嵌套顺序性能最好？哪种嵌套顺序性能最差?**

  如上图所示，ikj 的性能最好，jki 和 kji 的性能最差

- **教材《深入理解计算机系统》**(CSAPP 3e **中文版** P449) **分析了 6 个版本的矩阵乘法最内层循环中的** cache miss **次数。和你观察到的结果一致吗？最内层循环中数据访问的步长是怎么影响性能的?**

  和我观察到的一致，最内层相邻数据访问的步长越短，性能越高。最内层数据访问步长的缩短提升了程序的空间局部性，进而提高缓存命中率，提升性能。

- **修改** `matrixMultiply.c` **，再次观察程序的性能是否有改善（浮点运算吞吐率** Gflops/s**），从中你得到哪些经验?**

```
gcc -o matrixMultiply -ggdb -Wall
./matrixMultiply
ijk:    n = 1000, 0.527 Gflop/s
ikj:    n = 1000, 13.961 Gflop/s
jik:    n = 1000, 0.534 Gflop/s
jki:    n = 1000, 0.394 Gflop/s
kij:    n = 1000, 18.938 Gflop/s
kji:    n = 1000, 0.394 Gflop/s
```

  部分有改善，得到经验：在部分情况下将最常用变量保存在寄存器中，可以提高程序性能。

- **为什么实际运算性能差距如此大?**

  硬件级的预取机制可以识别出各类访问模式，提前将内存中的内容缓存以提高命中率。

Exercise 3: Cache Blocking and Matrix Transposition

分块代码如下：

```
void transpose_blocking(int n, int blocksize, int *dst, int *src) {
    int num_blk = (n-1)/blocksize + 1;
    for(int i=0; i<num_blk; i++)
        for(int j=0; j<num_blk; j++)
            for(int x = i*blocksize; x<(i+1)*blocksize && x<n; x++)
                for(int y = j*blocksize; y<(j+1)*blocksize && y<n; y++)
                    dst[y + x * n] = src[x + y * n];
}
```

Part1: 改变矩阵大小

Matrix Size	100	500	1000	2000	5000
Naive	0.007	0.175	1.604	18.43	215.29
Blocking	0.004	0.120	3.727	10.78	75.217

- **矩阵分块实现矩阵转置是否比不用矩阵分块的方法快?**

  在矩阵较小时，分块并没有明显变快（甚至在矩阵大小为1000时变得更慢）；矩阵较大时，分块才能明显提高性能，且矩阵越大性能提升越明显。

- **为什么矩阵大小要达到一定程度，矩阵分块算法才有效果?**

  矩阵较小时，原始矩阵可以较完整的放入缓存中，或大部分放入缓存，缺失占比较少，此时分块性能提升不大。

Part2: 改变分块大小（Blocksize）

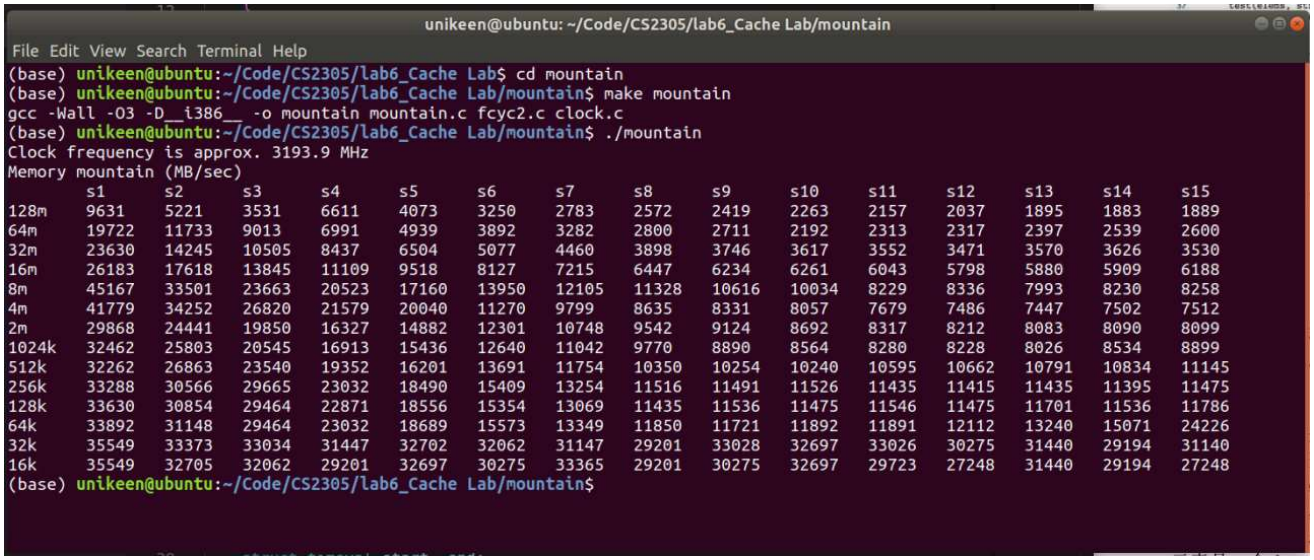
Block Size	50	100	200	500	1000	5000
Naive	1005.73	1015.83	1005.56	1007.91	1005.95	982.944
Blocking	134.898	124.476	132.116	130.595	164.26	956.225

- **当 blocksize 增加时性能呈现什么变化趋势？为什么?**

  性能先提升后降低。blocksize过小时，缓存块中的数据无法被全部利用；blocksize过大时，缓存无法放下单个块，性能逐渐降低到不分块时的情况。

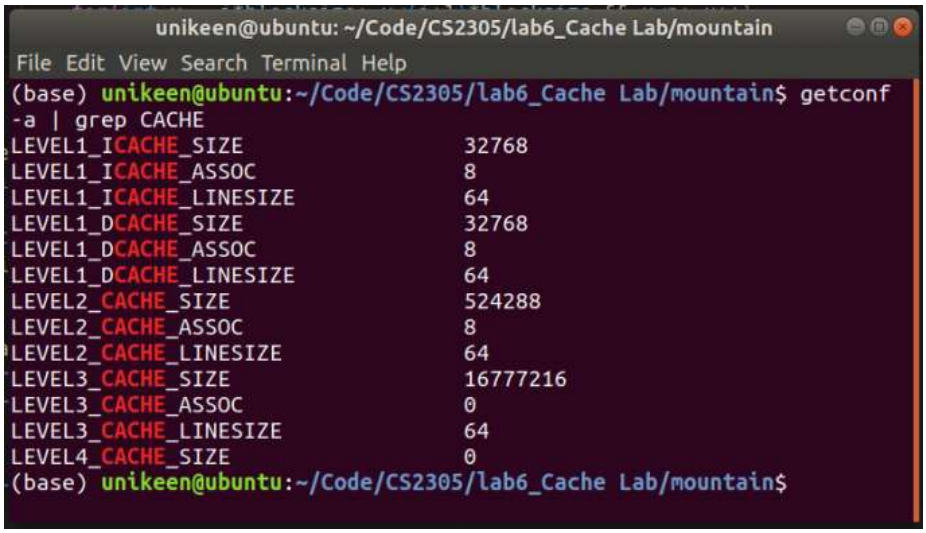
Exercise 4: Memory Mountain

- **请罗列出运行结果**

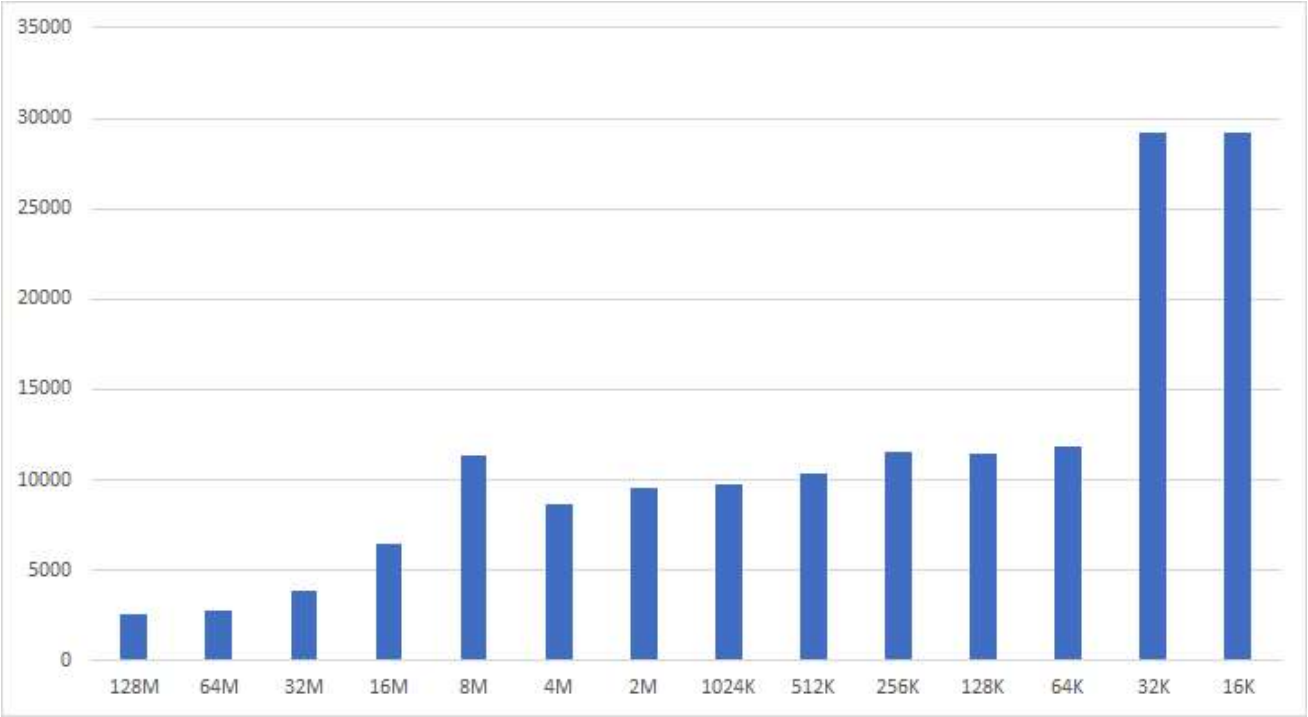


- **程序运行所在的系统，一级高速缓存、二级高速缓存的大小分别为多大？有三级高速缓存吗？如果有，容量为多少？**

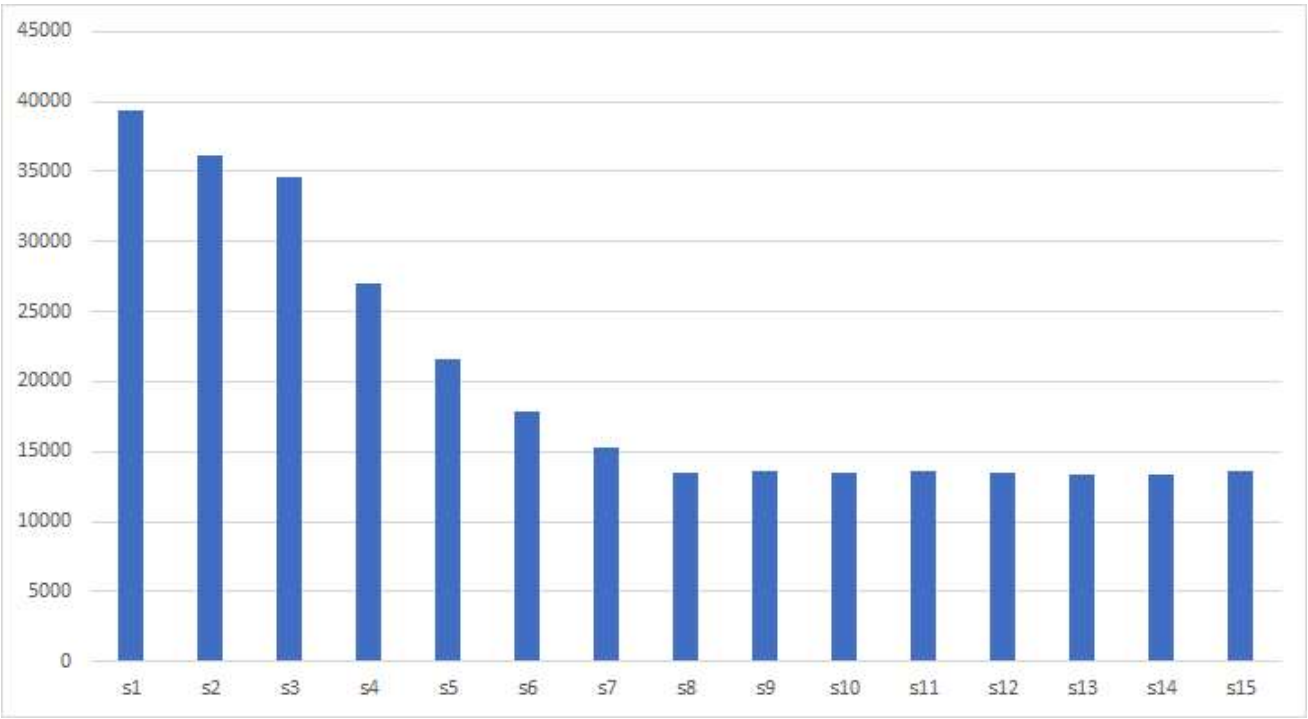
  一级数据缓存和一级指令缓存均为 32KB；二级高速缓存为 512KB；三级高速缓存为 16MB。



  将步长8的测试数据作图如下，可见明显的断层出现在 16M 和 32K 处，与系统配置大致相符。



- **高速缓存的块大小是多少？为什么？**



块的大小为 64 byte。步长高于 8 之后，继续增大步长吞吐率变化缓慢，说明此时每个读请求在 L2 都不会命中，读吞吐量理论为常数（由从 L3 传送到 L2 的速率决定）