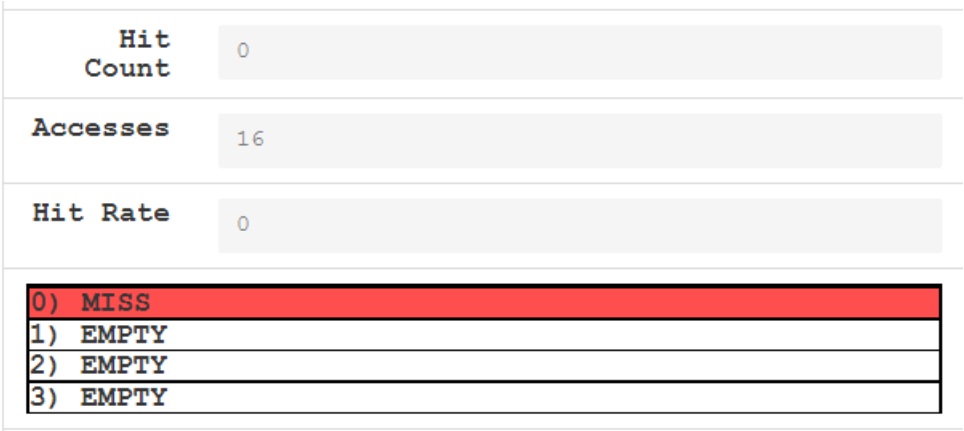
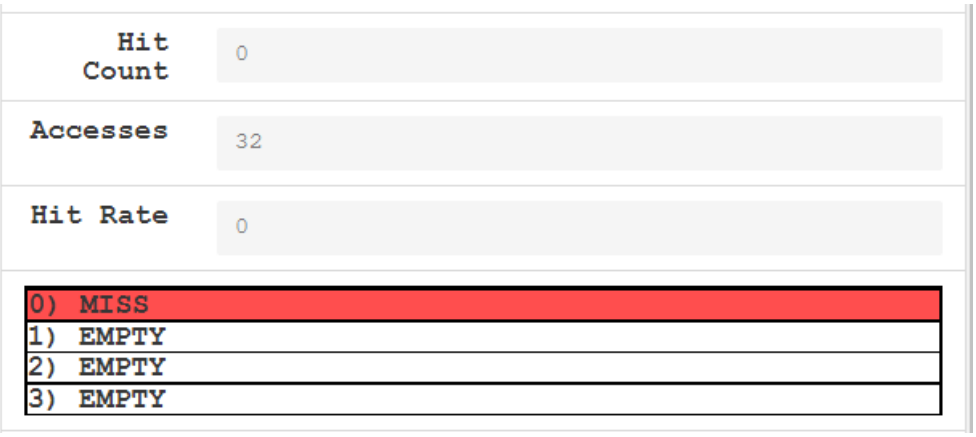
Cache实验

Exercise 1: Cache Visualization

场景1



- Cache 命中率是多少?
  - 0%
- 为什么会出现这个 cache 命中率?
  - array大小为128 byte，可存 32 个int。数据块大小为 8 bytes，cache 一共有 4 个数据块，即cache的大小为 32 byte，一个block里面可存2个int，最多可存8个int。访问步长为 8，即下一次访问的int不在cache内。由于使用直接映射，每次访问都会映射到cache中的同一个block，因此每一次访问都会替换掉上一次访问的block，所以每次访问都会miss。
- 增加 Rep Count 参数的值，可以提高命中率吗? 为什么?
  - 不可以。因为相邻两次访问的地址相差 32 byte = cache的大小，由于使用直接映射，每次访问都会映射到cache中的同一个block，因此每一次访问都会替换掉上一次访问的cache，所以命中率一直为0。
  - rep count = 8 时:



- 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（program parameters ）?
  - 修改 array size 为 32 byte，此时cache中的block不会因为后续访问被替换，只有第一次访问时会miss，后续访问均会命中。所以只要增大重复次数，当重复次数足够大时，命中率趋于100%。

```
main:   li  a0, 32      # array size in BYTES (power of 2 < array size)
    li  a1, 1       # step size  (power of 2 > 0)
    li  a2, 100     # rep count  (int > 0)
    li  a3, 0       # 0 - option 0, 1 - option 1
```

Hit Count	796
Accesses	800
Hit Rate	0.995

```
0) HIT
1) HIT
2) HIT
3) HIT
```

场景2

Hit Count	48
Accesses	64
Hit Rate	0.75

```
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
8) HIT
9) HIT
10) HIT
11) HIT
12) HIT
13) HIT
14) HIT
15) HIT
```

- Cache 命中率是多少?
  - 75%
- 为什么会出现这个 cache 命中率?
  - array大小为 256 byte，可存 64 个 int。数据块大小为 16 bytes，cache 一共有 16 个数据块，即cache的大小为 256 byte，一个block里面可存 4 个int，最多可存个 64 int。由于使用 4 路组相联映射，步长为2，故cache中每个block对应的内存空间被连续访问4次，只有第一次访问会miss，后续访问全部hit。
- 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%。因为cache的空间可以容纳整个数组，第一次miss后不会再发生cache替换，所以后续的访问全部hit。

Hit Count	6384
Accesses	6400
Hit Rate	0.9975

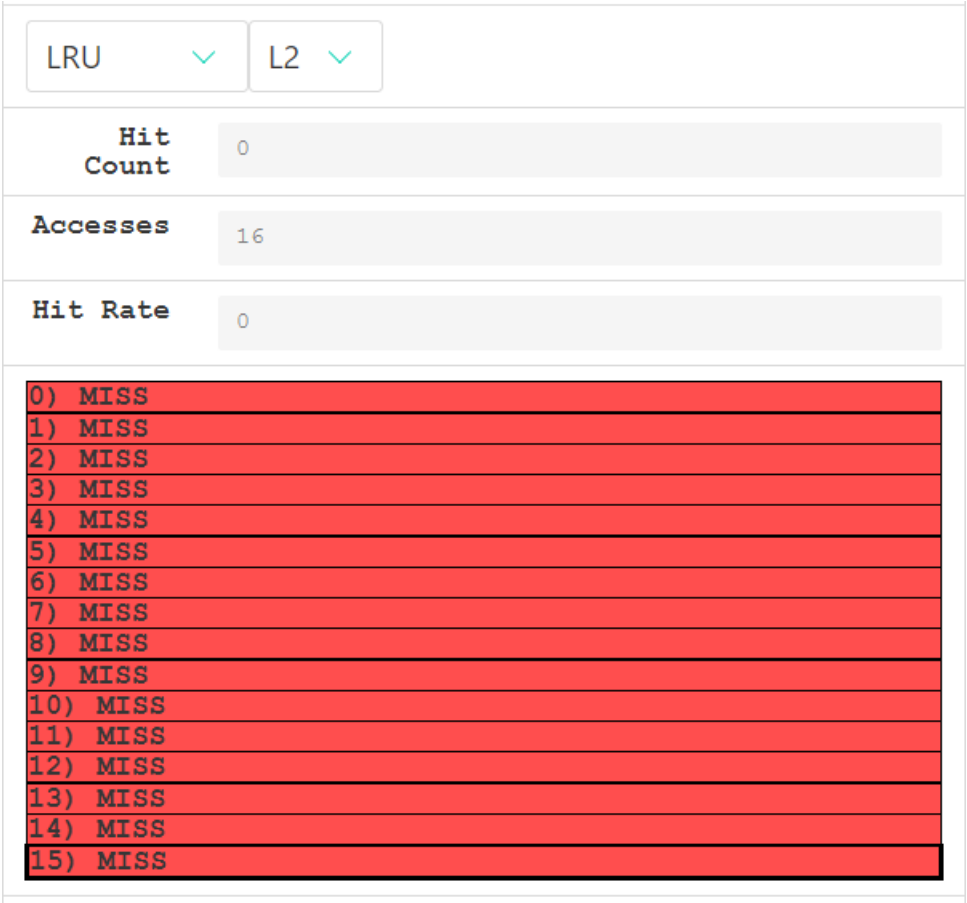
```
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
8) HIT
9) HIT
10) HIT
11) HIT
12) HIT
13) HIT
14) HIT
15) HIT
```

场景3

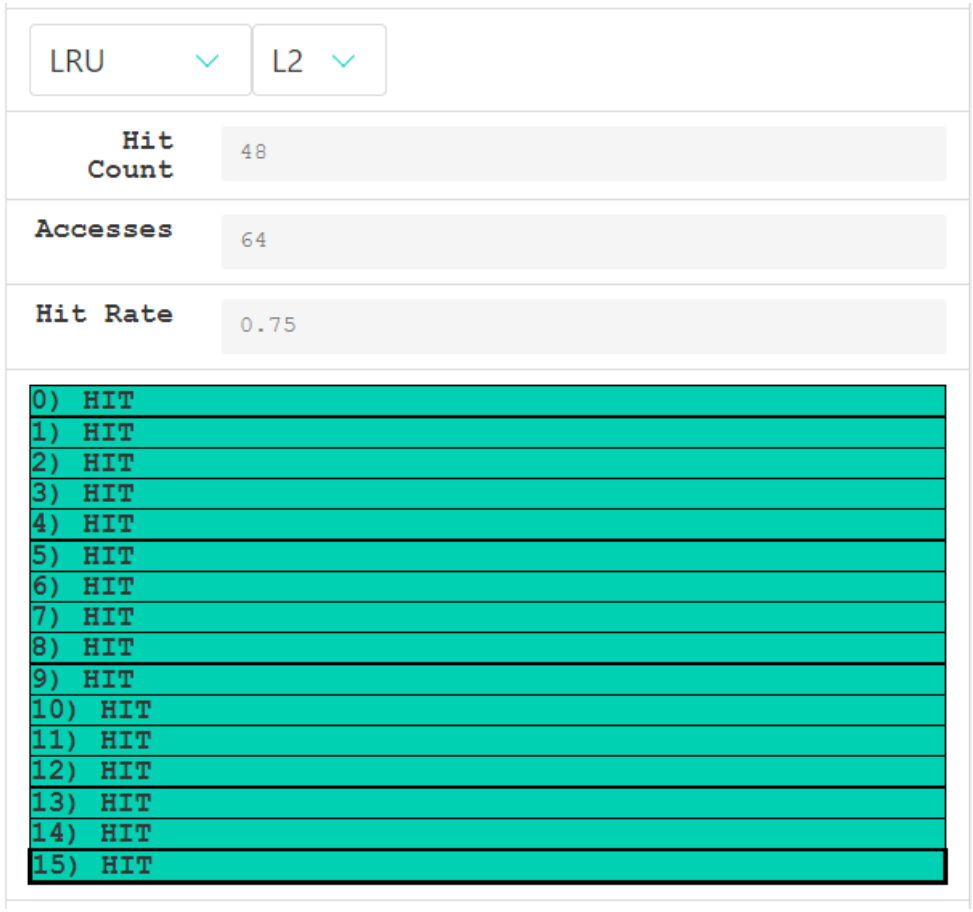
LRU L1

Hit Count	16
Accesses	32
Hit Rate	0.5

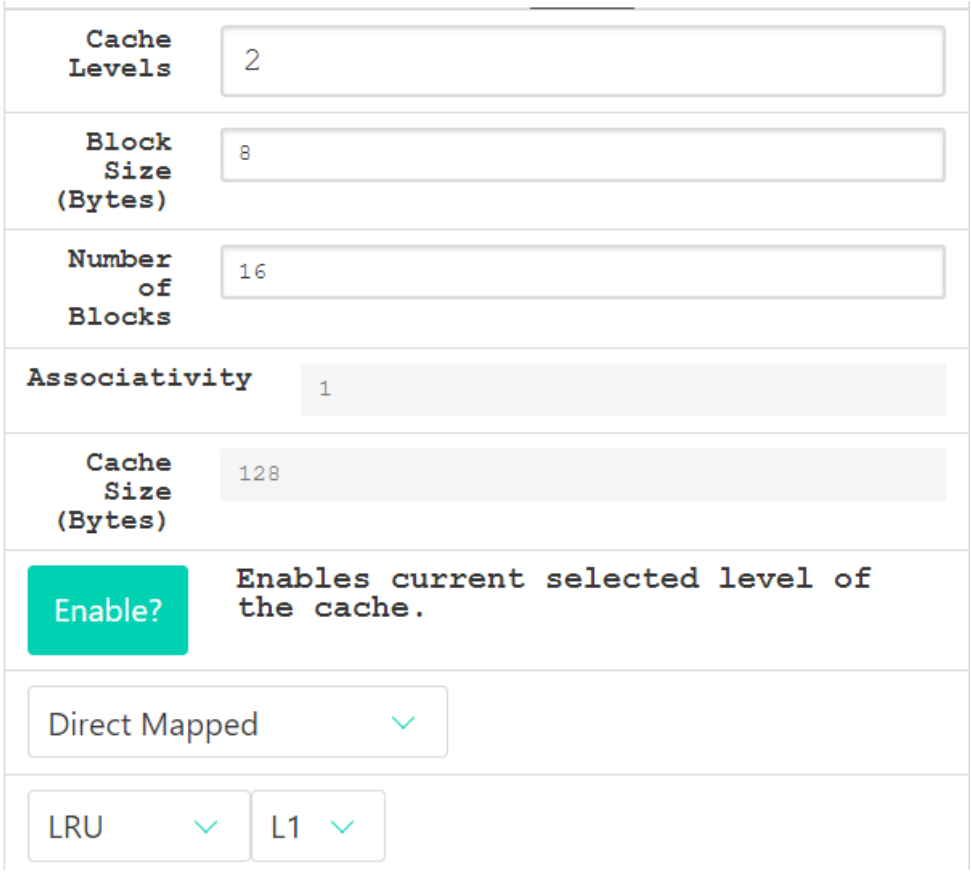
```
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
```



- L1 cache 和 L2 cache 的命中率分别为多少?
  - 50%
  - 0%
- 总共访问了 L1 cache 几次？ L1 Miss 次数为多少?
  - 32次
  - 16次
- 总共访问了 L2 cache 几次?
  - 16次
- 哪一个程序参数（寄存器 a0 ~ a3）可以增加 L2 hit rate, 并且保持 L1 hit rate 不变?
  - a2：rep count



- 如果将 L1 cache 中的块数增加， L1 、 L2 hit rate 有什么变化？
  - 没有变化



LRU L1

Hit Count	16
Accesses	32
Hit Rate	0.5

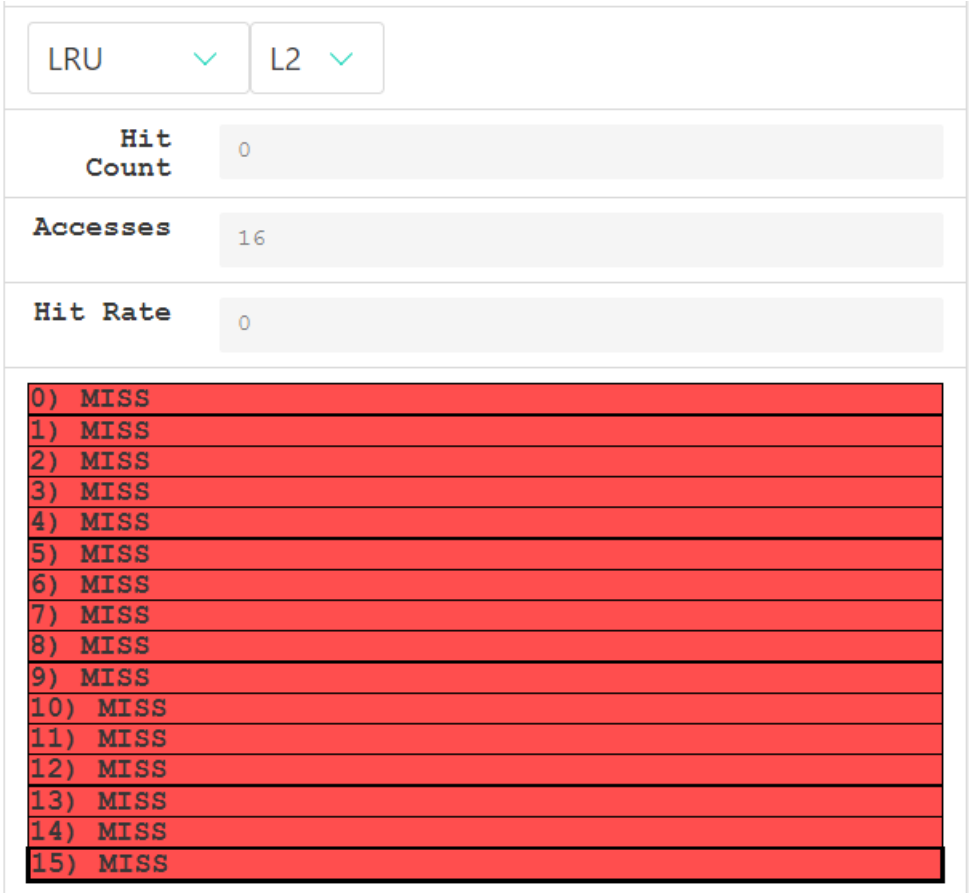
```
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
8) HIT
9) HIT
10) HIT
11) HIT
12) HIT
13) HIT
14) HIT
15) HIT
```

Cache Levels	2
Block Size (Bytes)	8
Number of Blocks	16
Associativity	1
Cache Size (Bytes)	128

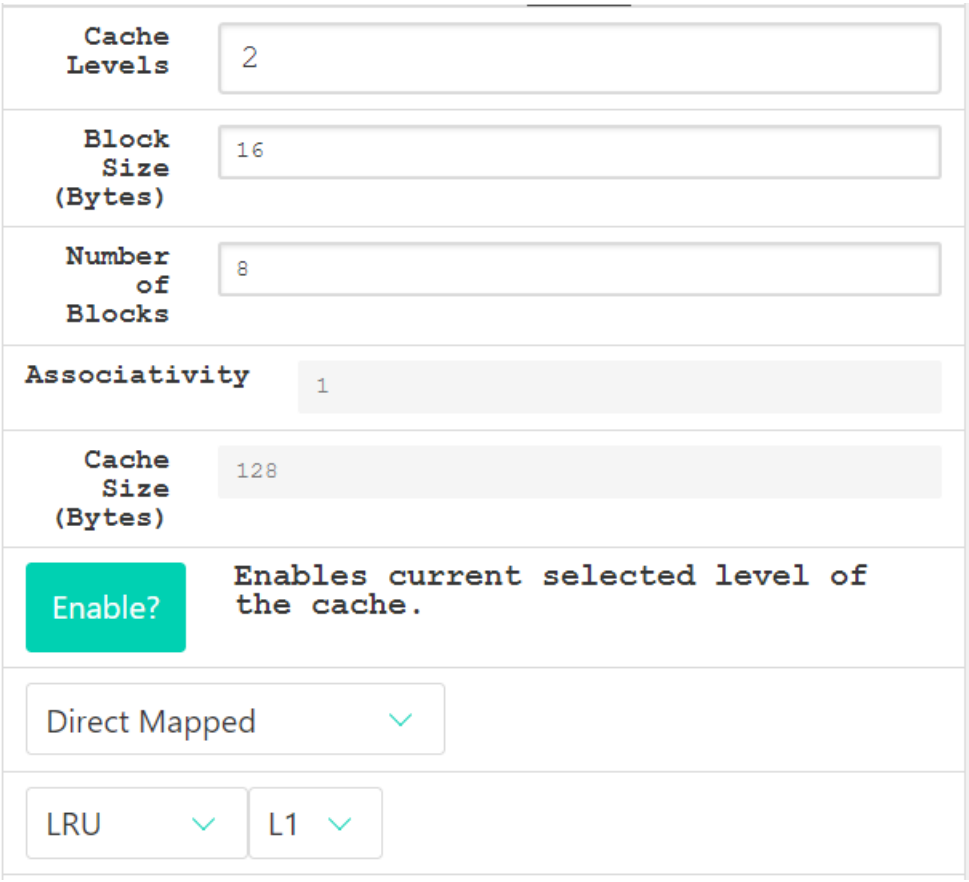
Enable? Enables current selected level of the cache.

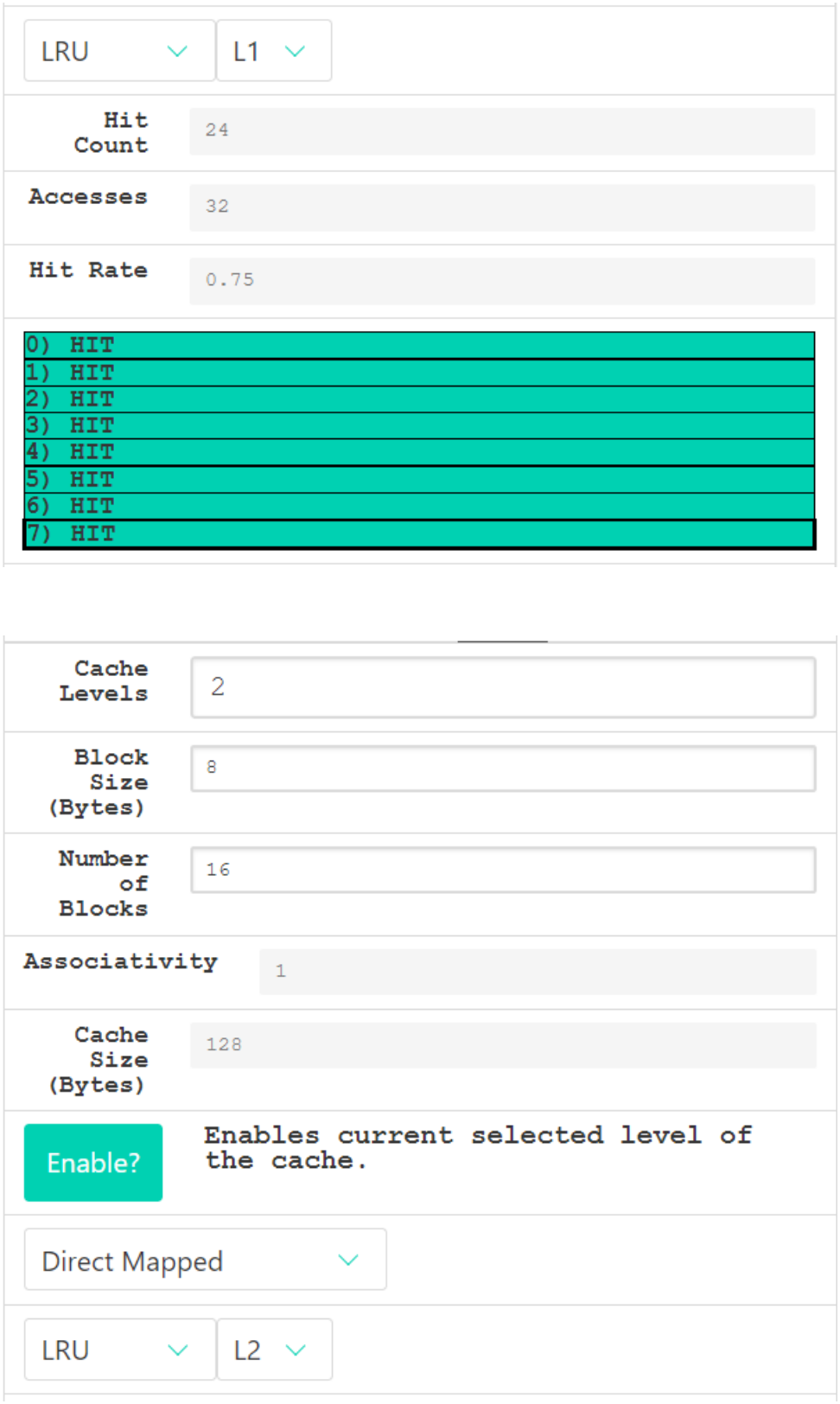
Direct Mapped

LRU L2



- 如果将 L1 cache 中的块大小增加，L1 、 L2 hit rate 有什么变化？
  - L1变大，L2没有变化




LRU
L1
Hit Count
24
Accesses
32
Hit Rate
0.75
0) HIT
1) HIT
2) HIT
3) HIT
4) HIT
5) HIT
6) HIT
7) HIT
Cache Levels
2
Block Size (Bytes)
8
Number of Blocks
16
Associativity
1
Cache Size (Bytes)
128
Enable?
Enables current selected level of the cache.
Direct Mapped
LRU
L2

Hit Count	0
Accesses	8
Hit Rate	0

0) MISS
1) EMPTY
2) MISS
3) EMPTY
4) MISS
5) EMPTY
6) MISS
7) EMPTY
8) MISS
9) EMPTY
10) MISS
11) EMPTY
12) MISS
13) EMPTY
14) MISS
15) EMPTY

Exercise 2: Loop Ordering and Matrix Multiplication

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ make ex2
gcc -o matrixMultiply -ggdb -Wall -pedantic -std=gnu99 -O3 matrixMultiply.c
./matrixMultiply
ijk:    n = 1000, 1.733 Gflop/s
ikj:    n = 1000, 11.046 Gflop/s
jik:    n = 1000, 1.844 Gflop/s
jki:    n = 1000, 0.098 Gflop/s
kij:    n = 1000, 8.746 Gflop/s
kji:    n = 1000, 0.091 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？
  - ikj最好，kji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， 如下图所示。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?
  - 一致。
  - 最内层循环中数据访问的步长越短，性能越高。
- 参考如下代码（CSAPP 3e 中文版 P448），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?
  - 有改善，但很少。
  - 将某层循环之前将在该层循环中要使用但是不变的量保存在寄存器中，可以在一定程度上提高程序性能。

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ make ex2
gcc -o matrixMultiply -ggdb -Wall -pedantic -std=gnu99 -O3 matrixMultiply.c
./matrixMultiply
ijk:    n = 1000, 1.783 Gflop/s
ikj:    n = 1000, 11.767 Gflop/s
jik:    n = 1000, 1.902 Gflop/s
jki:    n = 1000, 0.091 Gflop/s
kij:    n = 1000, 9.517 Gflop/s
kji:    n = 1000, 0.089 Gflop/s
```

- 教材《深入理解计算机系统》（CSAPP 3e 中文版 P449）在 Intel core i7 处理器上分析了 6 个版本的矩阵乘法的性能，可以发现：当矩阵大小为 700*700时，最块的版本比最慢的版本快超过 30 倍，在图 6-45 中的分析可以看出：这两种算法的 cache 失效率相差的倍数仅为 4 倍，为什么实际运算性能会差距如此大?
    - 预取硬件可以识别出步长为1的访问模式，提前将内存中的内容缓存，进一步提高命中率。

Exercise 3: Cache Blocking and Matrix Transposition

```
void transpose_blocking(int n, int blocksize, int *dst, int *src)
{
    // YOUR CODE HERE
    int block_n = (n-1)/blocksize + 1;
    for(int bx = 0; bx < block_n; bx++)
    {
        for(int by = 0; by < block_n; by++)
        {
            for(int y = by * blocksize; y < (by + 1) * blocksize && y < n; y++)
            {
                for(int x = bx * blocksize; x < (bx + 1) * blocksize && x < n;
x++)
                {
                    dst[y + x * n] = src[x + y * n];
                }
            }
        }
    }
}
```

Part 1: 改变矩阵的大小

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 100 20
Testing naive transpose: 0.003 milliseconds
Testing transpose with blocking: 0.006 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 500 20
Testing naive transpose: 0.193 milliseconds
Testing transpose with blocking: 0.227 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 1000 20
Testing naive transpose: 1.399 milliseconds
Testing transpose with blocking: 1.508 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 2000 20
Testing naive transpose: 31.769 milliseconds
Testing transpose with blocking: 7.645 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 20
Testing naive transpose: 317.136 milliseconds
Testing transpose with blocking: 47.062 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 10000 2
0
Testing naive transpose: 25711.8 milliseconds
Testing transpose with blocking: 2946.5 milliseconds
```

n	100	500	1000	2000	5000	10000
naive	0.003	0.193	1.399	31.769	317.136	25711.8
with blocking	0.006	0.227	1.508	7.645	46.062	2946.5

- 将 blocksize 固定为 20，n 分别设置为 100，500，1000，2000，5000 和 10000。矩阵分块实现矩阵转置是否比不用矩阵分块的方法快? 为什么矩阵大小要达到一定程度，矩阵分块算法才有效果?
    - n较小时，分块转置并没有明显比不分块更快，只有当矩阵大小增大到一定大时，分块才明显比不分块快。
    - n较小时，原始矩阵可以完整或大部分放入缓存之中，因此分块的性能提升不大。

Part 2: 改变分块大小（Blocksize）

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 20
Testing naive transpose: 312.583 milliseconds
Testing transpose with blocking: 45.634 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 50
Testing naive transpose: 311.194 milliseconds
Testing transpose with blocking: 53.288 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 100
Testing naive transpose: 312.432 milliseconds
Testing transpose with blocking: 49.975 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 200
Testing naive transpose: 306.902 milliseconds
Testing transpose with blocking: 50.613 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 500
Testing naive transpose: 310.3 milliseconds
Testing transpose with blocking: 46.543 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 1000
Testing naive transpose: 320.67 milliseconds
Testing transpose with blocking: 60.799 milliseconds
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06$ ./transpose 5000 5000
Testing naive transpose: 313.74 milliseconds
Testing transpose with blocking: 303.716 milliseconds
```

Blocksize	50	100	200	500	1000	5000
naive	311.194	312.432	306.902	310.0	320.67	313.74
with blocking	53.288	49.975	50.613	46.543	60.799	303.716

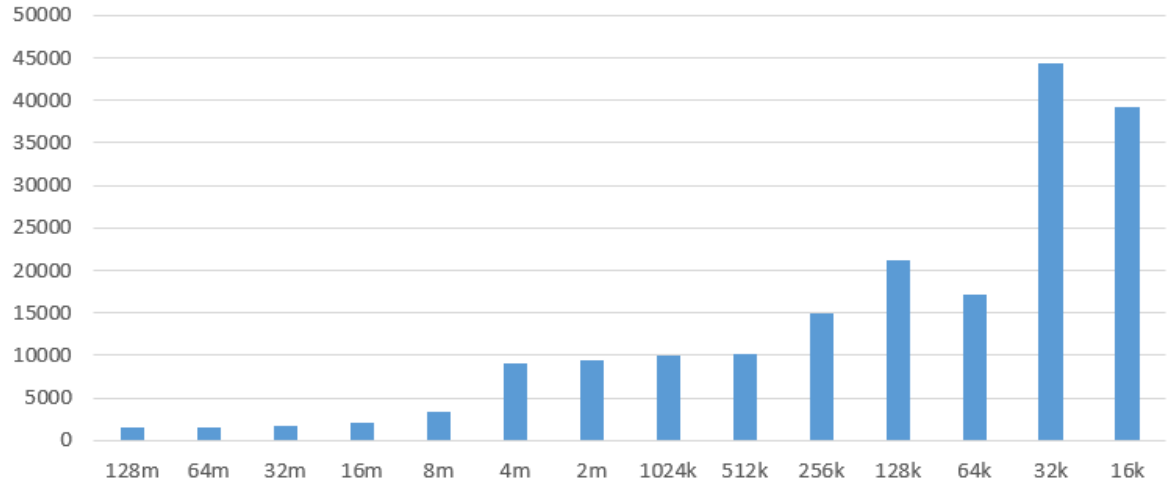
- 将 n 的值固定为 5000, 将 blocksize 设置为 50，100，200，500，1000，5000 分别多次运行 transpose 程序. 当blocksize 增加时性能呈现什么变化趋势? 为什么?
    - 因虚拟机自身性能问题，n固定为10000时运行结果误差较大，故将n置为5000。
    - 性能先提高后降低。blocksize过小时，缓存块中的数据不能被充分利用；blocksize过大时，单个块无法被完整缓存，就需要进行缓存替换。

Exercise 4: Memory Mountain

- 请罗列出运行结果。

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06/mountain$ ./mountain
Clock frequency is approx. 1992.0 MHz
Memory mountain (MB/sec)
        s1      s2      s3      s4      s5      s6      s7      s8      s9      s10     s11     s12     s13     s14     s15
128m    15187   7943    4952    3649    2838    2404    1757    1563    1593    1245    1175    1086    1026    953     941
64m     15530   8133    5165    3773    3046    2503    1893    1603    1430    1293    1363    1113    1056    1000    975
32m     15027   8691    5612    4135    3298    2404    1979    1631    1548    1410    1274    1204    1081    1062    999
16m     17061   10463   6995    4985    4013    3343    2416    2139    1833    1752    1644    1509    1467    1415    1551
8m      20735   15657   11176   7675    5423    4645    3870    3328    3333    3417    3437    3408    3708    3843    4157
4m      26925   22063   18323   15451   12726   11534   10160   9081    8616    8114    7622    7196    6872    6874    6735
2m      35211   25504   20781   16884   14158   12164   10634   9405    8855    8352    7831    7386    7283    7058    6898
1024k   36747   26610   21700   17800   14884   12775   11221   9913    9321    8797    8244    7780    7505    7209    7100
512k    37139   26880   21987   18237   15204   13058   11444   10087   9593    9132    8693    8267    8389    8263    8412
256k    41072   31013   27481   24243   21141   18716   16741   14865   14249   13800   13563   13535   13584   13748   13695
128k    43705   35122   34454   33134   29485   26812   23772   21193   20244   19992   19779   19779   19163   19286   18838
64k     43371   34739   33759   31261   26373   22806   19167   17070   26809   29466   26547   26213   25814   30876   34673
32k     53285   51805   50130   49152   47641   44042   45046   44344   45318   44697   42987   42486   42184   41993   40659
16k     52982   50678   48348   45838   45318   42156   40888   39227   37376   37505   34093   34419   34382   32826   31526
```

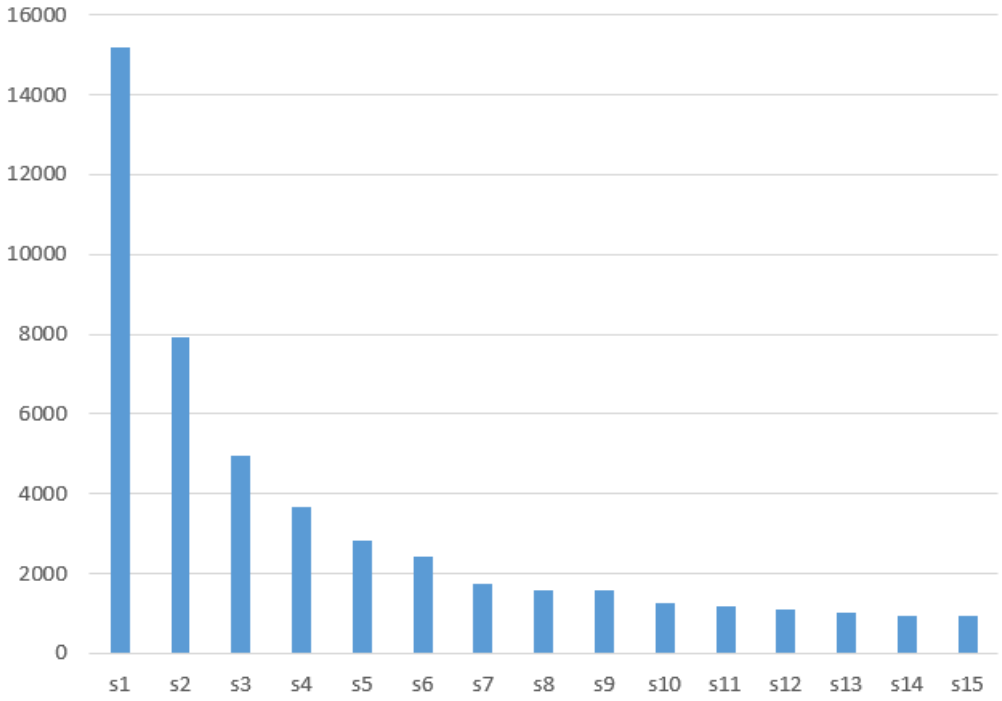
- 程序运行所在的系统，一级高速缓存、二级高速缓存的大小分别为多大？ 有三级高速缓存吗？ 如果有，容量为多少？
    - 一级高速缓存为32KB，二级高速缓存为256KB，三级高速缓存为4MB。



- 查看系统中高速缓存的配置，并截图。对比一下你的判断是否和系统配置一致。
    - 一级指令高速缓存为32768B = 32KB，一级数据缓存为32KB；二级高速缓存为262144B = 256KB；三级高速缓存为8388608B = 8MB。

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw06/mountain$ getconf -
a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

- 继续观察程序运行结果，固定工作集大小，模仿下图，例如数组长度为 4MB，观察步长从 1 变化到 15 的情况下读数据的吞吐率。回答问题：高速缓存的块大小（block size）是多少？ 为什么？
    - 块大小为 64 byte。
    - 从图中可以看出，当步长达到8之后，继续增大步长，吞吐率变化率小。


16000
14000
12000
10000
8000
6000
4000
2000
0
s1
s2
s3
s4
s5
s6
s7
s8
s9
s10
s11
s12
s13
s14
s15