# 计算机组织与体系结构实习 Lab 3

# 高速缓存的模拟、配置和优化

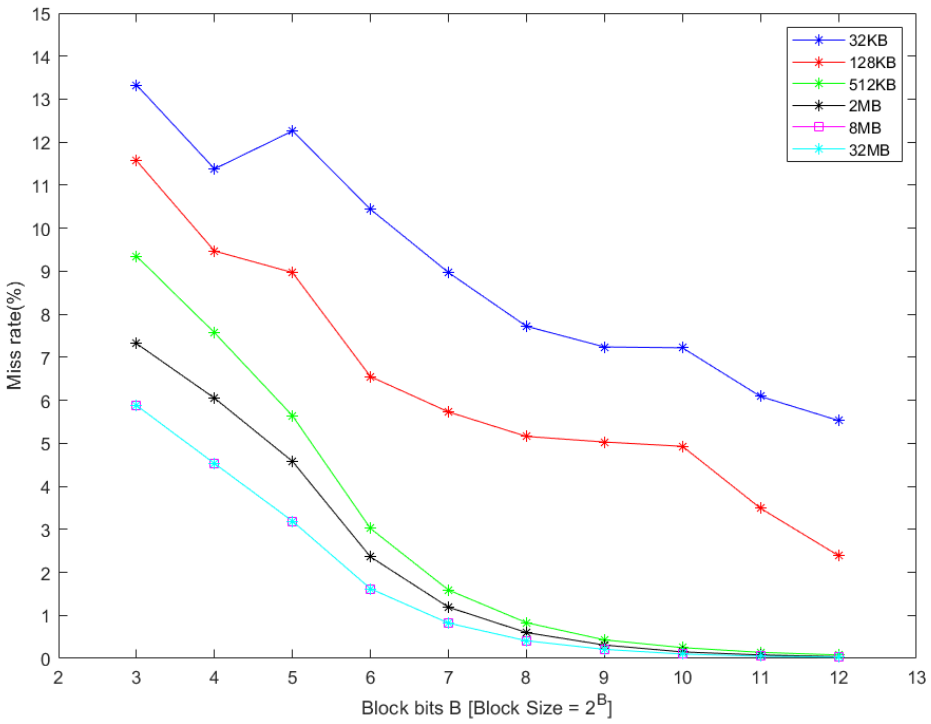
## 一、单级 Cache 模拟

使用所给模拟器框架，使用给定的测试踪迹 (trace)，完成单级高速缓存的模拟。

1. **在不同的 Cache Size (32KB ~ 32MB) 的条件下， Miss Rate 随 Block Size 变化的趋势，收集数据并绘制折线图。并说明变化原因。**

我选择了 32KB, 128KB, 512KB, 2MB, 8MB, 32MB 六种不同的 Cache Size，固定组相联度大小为 8，分别在 $2^3 \sim 2^{12}$ 的 Block Size 下进行 Miss Rate 的统计，写策略采用 Write Back & Write Allocate，Cache 替换策略采用 FIFO。结果如下：

**[01-mcf-gem5-xcg.trace]**

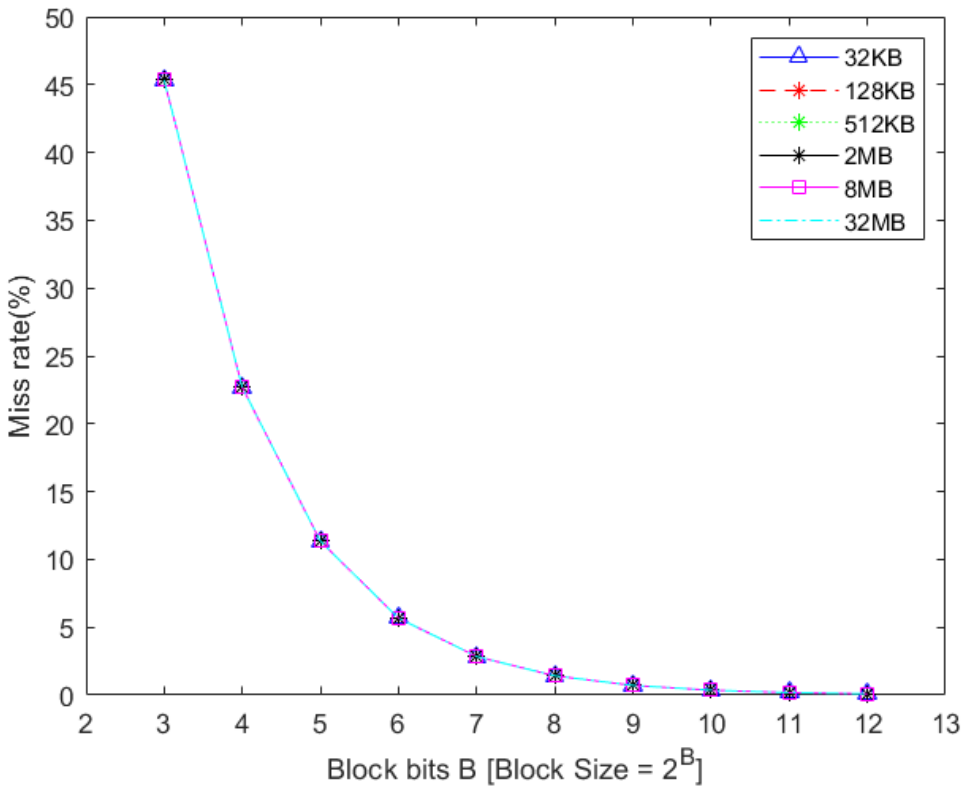



我们首先分析 `trace2017/` 文件夹下的第一个 trace。从折线图中我们可以得到如

下统计规律：相同的 Cache Size 下，Miss Rate 随着 Block Size 的增大而降低；相同的 Block Size 下，Miss Rate 也随着 Cache Size 的增大而降低。但值得注意的是，随着 Size 的增大，Miss Rate 降低的幅度越来越小，而大的 Block Size/ Cache Size 必定增加 Cache 的 Hit Latency 以及 Miss Penalty。因此从延时的角度来看，最优策略必定在中间而非 Miss Rate 最低点。

有一个反常的现象在 32KB 的 Cache 中，Miss Rate 在 32B Block Size 时会突然反常地上涨，原因可能是 Block Size 的上涨导致 Set 数减少，原本不冲突的地址发生了冲突。

**[02-stream-gem5-xaa.trace]**

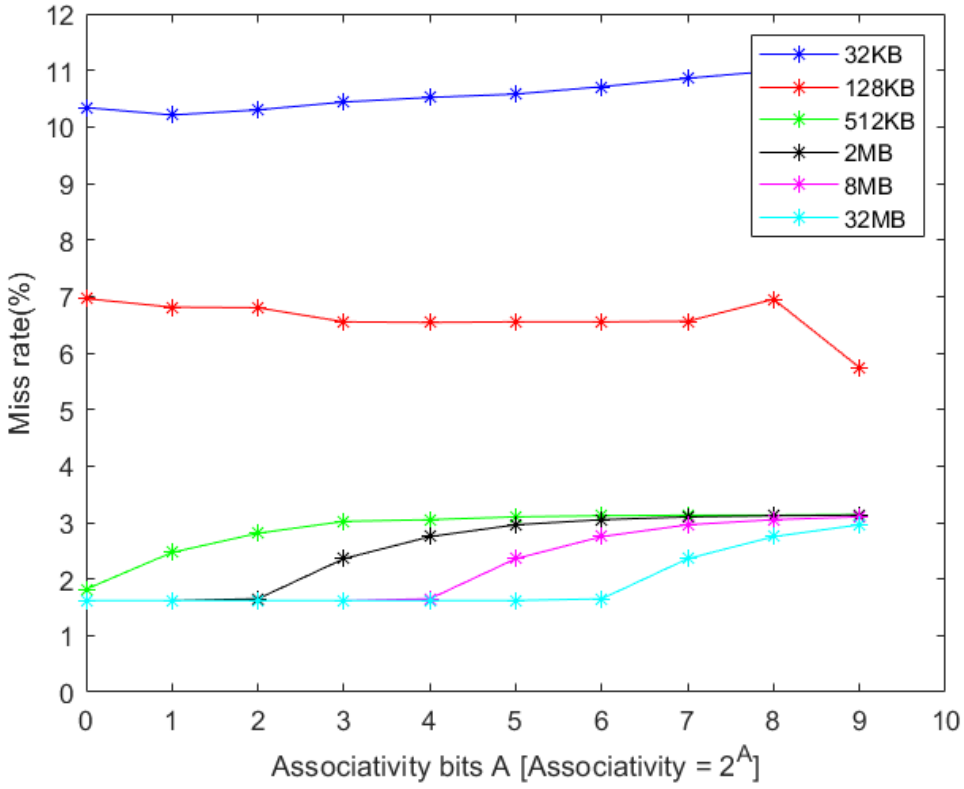



在第二个 trace 中，同样的 Miss Rate 随着 Block Size 的增大而降低，但是不同的 Cache Size 对 Miss Rate 没有影响，这是由于这个 trace 本身访存是在三个基地址上交替+8 访问，在之后新增一个地址，四个地址交替访问，其中一个地址不再变化。当 Set 数量足够将这几个地址分离开时，决定 Miss Rate 的显然就只剩下 Block Size。这些配置的 Set 数足够大，因此表现相同。

**2. 在不同的 Cache Size 的条件下， Miss Rate 随 Associativity 变化的趋势，收集数据并绘制折线图。并说明变化原因。**

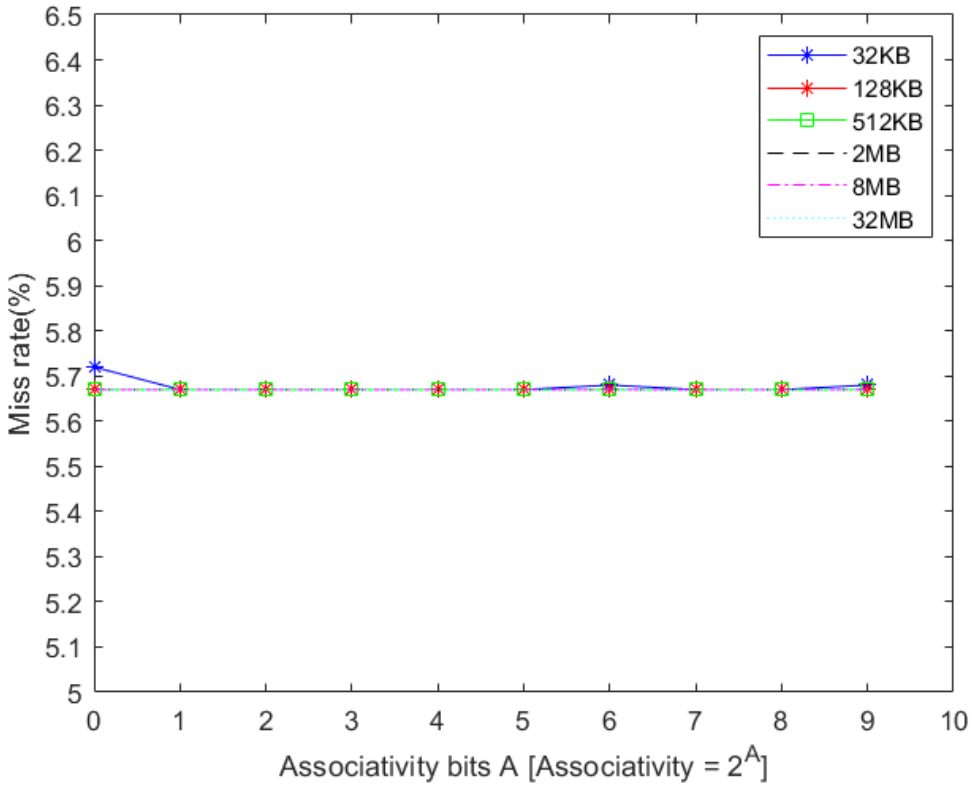
我选择了 32KB, 128KB, 512KB, 2MB, 8MB, 32MB 六种不同的 Cache Size，固定 Block Size 为 64B，分别在 $2^0$ ~ $2^9$ 的组相联度下进行 Miss Rate 的统计，写策略采用 Write Back & Write Allocate，Cache 替换策略采用 FIFO。结果如下：

**[01-mcf-gem5-xcg.trace]**




对于 32KB 和 128KB 的 Cache，我们可以发现 Miss Rate 随着 Associativity 变化在一个值附近，基本不发生改变。我们可以定性判断，直接映射存取和全相联，区别也仅仅在排列方式上，因此不会有较大差异。而 512KB, 2MB, 8MB, 32MB 大小的 Cache 有一个有趣的现象，随着 Associativity 的增大，Miss Rate 首先保持不变，达到一个阈值之后开始上涨，趋向一个高一点的恒量。而 Cache Size 越大，对应的 Associativity 阈值越大。

**[02-stream-gem5-xaa.trace]**




第二个 trace 的特点在之前有所提及，因此组相联数对其命中率的影响也不大（只要 Set 数不会小到对这四个地址产生冲突），事实上也的确是近似一条直线，维持在 5.6%左右。

**3. 比较 Write Through 和 Write Back、 Write Allocate 和 No-write Allocate 的总访问延时的差异。**

同样地，考虑 32KB, 128KB, 512KB, 2MB, 8MB, 32MB 六种不同的 Cache Size，固定 Block Size 为 64B，组相联度为 8 ways，替换策略为 FIFO，使用 Write Through & Write Allocate, Write Through & No-write Allocate, Write Back & Write Allocate, Write Back & No-write Allocate 四种组合策略，探究总访问延时差异。

这里我们设置 L1 Cache 的 Hit Latency 为 1ns，Memory 的 Latency 为 50ns，忽略总线延迟。

**[01-mcf-gem5-xcg.trace]**

| 单位：ns | WT & WA | WT & N-WA | WB & WA | WB & N-WA |
|---|---|---|---|---|
| **32KB** | 16319550 | 16453050 | **4820550** | 5414600 |
| **128KB** | 15572600 | 15632150 | **2902700** | 3107100 |
| **512KB** | 14815400 | 14833800 | **915800** | 967700 |
| **2MB** | 14676500 | 14663350 | 547900 | **534450** |
| **8MB** | 14521450 | 14521450 | **376500** | **376500** |
| **32MB** | 14521450 | 14521450 | **376500** | **376500** |

**[02-stream-gem5-xaa.trace]**

| 单位：ns | WT & WA | WT & N-WA | WB & WA | WB & N-WA |
|---|---|---|---|---|
| **32KB** | 11223300 | 11223200 | **1810000** | 6000100 |
| **128KB** | 11222850 | 11222900 | **1693900** | 5586900 |
| **512KB** | 11222750 | 11222850 | **1232900** | 3620550 |
| **2MB** | 11222750 | 11222750 | **923650** | **923650** |
| **8MB** | 11222750 | 11222750 | **923650** | **923650** |
| **32MB** | 11222750 | 11222750 | **923650** | **923650** |

在两个 trace 中，我们都可以发现，在延时方面，Write Back 通常大幅度优于 Write Through，而写分配也会比写不分配表现更优（通过降低 Miss Rate），因此 Write Back & Write Allocate 在大多数情况下为最优策略。

## 二、与 lab2 中的 CPU 模拟器联调完成模拟

与 Lab 2 中的流水线模拟器联调，运行测试程序。该测试中 cache 的配置如下：

| Level | Capacity | Associativity | Line size(Bytes) | WriteUp Polity | Hit Latency |
|---|---|---|---|---|---|
| L1 | 32 KB | 8 ways | 64 | write Back | 1 cpu cycle |
| L2 | 256 KB | 8 ways | 64 | write Back | 8 cpu cycle |
| LLC | 8 MB | 8 ways | 64 | write Back | 20 cpu cycle |

在该实验中，我将 lab2 模拟器和 lab3 Cache 的代码进行联调，位于 `lab2+1ab3/` 文件夹下，输入 `make cache` 指令编译生成./sim 文件，运行 `./sim -f <filename> (-s)` 指令直接进行 Pipeline+Cache 的性能模拟，添加 -s 参数启用单步模式，默认分支预测策略为 Always taken。定义的内存访问时钟周期为 100。

**1. 测试程序运行结果正确，并打印动态执行的指令数和 CPI。**

我们分别运行 `test/` 下的 7 个可执行文件，展示运行结果截图。运行结果主要包括 CPU Cycle 的 CPI，以及 I-Cache, D-Cache, L2-Cache, LLC 的 Miss Rate 以及总的 Latency。

**[add]**

```
11 12 13 14 15 1 2 3 4 5

---------- Stat Info ----------
Instruction Count:     866
CPU Cycle Count:       5877
CPU Clock CPI:         6.79
-------------------------------

Total I-Cache access cpu cycle: 1107
I-Cache Miss rate: 2.53% (28/1107)
Total D-Cache access cpu cycle: 272
D-Cache Miss rate: 4.04% (11/272)
Total L2 access cpu cycle: 312
L2 Miss rate: 97.44% (38/39)
Total LLC access cpu cycle: 760
LLC Miss rate: 100.00% (38/38)
```

**[ackermann]**

```
To solve Ackermann(a, b), please input a:3
please input b:6
509

---------- Stat Info ----------
Instruction Count:     6113340
CPU Cycle Count:       7998060
CPU Clock CPI:         1.31
-------------------------------

Total I-Cache access cpu cycle: 7749383
I-Cache Miss rate: 0.00% (30/7749383)
Total D-Cache access cpu cycle: 2238684
D-Cache Miss rate: 0.08% (1855/2238684)
Total L2 access cpu cycle: 25744
L2 Miss rate: 13.11% (422/3218)
Total LLC access cpu cycle: 8440
LLC Miss rate: 100.00% (422/422)
```

**[double]**

```
---------- Stat Info ----------
Instruction Count:     17805
CPU Cycle Count:       32476
CPU Clock CPI:         1.82
-------------------------------

Total I-Cache access cpu cycle: 20958
I-Cache Miss rate: 0.40% (84/20958)
Total D-Cache access cpu cycle: 923
D-Cache Miss rate: 1.52% (14/923)
Total L2 access cpu cycle: 784
L2 Miss rate: 92.86% (91/98)
Total LLC access cpu cycle: 1820
LLC Miss rate: 100.00% (91/91)
```

**[matmul]**

```
---------- Stat Info ----------
Instruction Count:     108777
CPU Cycle Count:       139349
CPU Clock CPI:         1.28
-------------------------------

Total I-Cache access cpu cycle: 129216
I-Cache Miss rate: 0.03% (38/129216)
Total D-Cache access cpu cycle: 20492
D-Cache Miss rate: 0.16% (32/20492)
Total L2 access cpu cycle: 560
L2 Miss rate: 95.71% (67/70)
Total LLC access cpu cycle: 1340
LLC Miss rate: 100.00% (67/67)
```

**[mul-div]**

```
5 10 15 20 25 1 2 3 4 5

---------- Stat Info ----------
Instruction Count:     891
CPU Cycle Count:       6038
CPU Clock CPI:         6.78
-------------------------------

Total I-Cache access cpu cycle: 1132
I-Cache Miss rate: 2.47% (28/1132)
Total D-Cache access cpu cycle: 272
D-Cache Miss rate: 4.04% (11/272)
Total L2 access cpu cycle: 312
L2 Miss rate: 97.44% (38/39)
Total LLC access cpu cycle: 760
LLC Miss rate: 100.00% (38/38)
```

**[n!]**

```
3628800

---------- Stat Info ----------
Instruction Count:      746
CPU Cycle Count:        5950
CPU Clock CPI:          7.98
-------------------------------

Total I-Cache access cpu cycle: 914
I-Cache Miss rate: 2.84% (26/914)
Total D-Cache access cpu cycle: 177
D-Cache Miss rate: 7.91% (14/177)
Total L2 access cpu cycle: 320
L2 Miss rate: 97.50% (39/40)
Total LLC access cpu cycle: 780
LLC Miss rate: 100.00% (39/39)
```

**[qsort]**

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39

---------- Stat Info ----------
Instruction Count:      21034
CPU Cycle Count:        37248
CPU Clock CPI:          1.77
-------------------------------

Total I-Cache access cpu cycle: 27257
I-Cache Miss rate: 0.13% (36/27257)
Total D-Cache access cpu cycle: 8374
D-Cache Miss rate: 0.49% (41/8374)
Total L2 access cpu cycle: 616
L2 Miss rate: 98.70% (76/77)
Total LLC access cpu cycle: 1520
LLC Miss rate: 100.00% (76/76)
```

**2、对比 lab2 中的流水线模拟器，分析 CPI 的变化原因。**

首先我们可以发现，使用三级缓存运行这几个可执行文件，其 LLC 的 Miss Rate 为 100%，L2-Cache 的 Miss Rate 也趋近 100%，并且访问 L2 和 LLC 的次数除了 ackermann 函数外都没超过 100。这是由于程序规模都比较小，使用单级的 I-Cache 和 D-Cache 就以及完全足够，剩余的两级 Cache 基本都没有用上，反而会增加访问 L2 和 LLC 的延迟，我们可以猜想，使用单级的 Cache 将得到更小的 Latency 和 CPI。

接着我们将 lab2 中的 CPU Clock CPI 与本次实验结果进行对比：

| CPU Clock CPI | **No Cache** | **3 级 Cache** |
|---|---|---|
| **add** | 32.41 | 6.79 |
| **ackermann(3,6)** | 37.55 | 1.31 |
| **double** | 6.32 | 1.82 |
| **matmul** | 19.75 | 1.28 |
| **mul-div** | 31.53 | 6.78 |
| **n!** | 24.76 | 7.98 |
| **qsort** | 40.72 | 1.77 |

首先我们声明为了避免过多的 latency，我们将 lab2 中指令的存取时钟周期都设置乘了 1, 因此实际的无 Cache 模拟器的运行 CPI 应该更长。即使在这种假设下，我们可以看到，使用 Cache 后 CPI 将显著降低，其中除了 add, mul-div, n!之外，其它的可执行程序的 CPU Clock CPI 都接近 Stage Clock CPI（lab2 报告中展示了不同程序的 Stage Count CPI 大小基本都在 1.3 左右）。而 add，mul-div 和 n!的程序规模过小，指令数数量级远远低于其它程序。这说明了在规模较大的指令访存序列下,平均每个流水级都仅仅使用了大约 1 个 CPU 时钟周期。我们看到 I-Cache 和 D-Cache 的命中率，几乎百分之百命中，由此说明了实际的可执行文件内存访问具有较高的 locality。

## 三、高速缓存管理策略优化

根据实习指导的要求，对默认配置下 Cache 进行优化。并使用所给测试踪迹（trace），对优化前后的 cache 性能进行比较。**我们假设处理器的主频为 2.0GHz**。

**1. 请填写以下参数。**

- 默认配置下，32nm 工艺节点下，L1 Cache 的 Hit Latency 为（ **1.47944** ）ns，约等于（ **3** ）cycle

- 默认配置下，32nm 工艺节点下，L2 Cache 的 Hit Latency 为（ **1.9206** ）ns，约等于（ **4** ）cycle

- 默认配置下，主存的 Access time 为（ **25** ）ns，约等于（ **50** ）cycle

| Level | Capacity | Associativity | Line size(Bytes) | WriteUp Polity |
|---|---|---|---|---|
| L1 | 32 KB | 8 ways | 64 | write Back |
| L2 | 256 KB | 8 ways | 64 | write Back |

(默认配置如上图)

2. **默认配置下，运行 trace2017 中的两个 trace，结果如下：**

**[01-mcf-gem5-xcg.trace]**

- 运行 trace 共( 10 )遍

- L1 Cache： 平均 Miss Rate = （ **10.05%** ）

- L2 Cache： 平均 Miss Rate = （ **25.61%** ）

- AMAT =（ **2.3159** ）ns

**[02-stream-gem5-xaa.trace]**

- 运行 trace 共( 10 )遍

- L1 Cache： 平均 Miss Rate = （ **5.67%** ）

- L2 Cache： 平均 Miss Rate = （ **51.30%** ）

- AMAT =（ **2.3155** ）ns

***该部分数值与实际运行结果一致***

3. **请填写最终确定的优化方案，并陈述理由。对于涉及到的算法，需要详细描述算法设计和实现思路，并给出优缺点分析。**

L1 Cache：Cache Size 256KB, Associativity 2 ways, Block Size 256B, WB & WA

L2 Cache：Cache Size 1MB, Associativity 2 ways, Block Size 1KB, WB & WA

- 优化配置下，32nm 工艺节点下，L1 Cache 的 Hit Latency 为（ **1.91445** ）ns，约等于（ **4** ）cycle

- 优化配置下，32nm 工艺节点下，L2 Cache 的 Hit Latency 为（ **2.01887** ）ns，约等于（ **4** ）cycle

注意到优化配置下使用到了 Block Size 很大的块进行存储，原因是使用大块能够极大地降低 Miss Rate，而使用 `cacti65` 模拟的得到的 Access time 没有很大的增加，因此使用大块优化策略。

LRU：利用 C++的 STL 库，使用链表配合哈希表存储缓存行，同时每次将访问过的块放置到该行的链表末尾，发生替换时将链首的块替换下，模拟了 LRU 替换最近最少访问的 Block 的行为。并且读写都是 O(1)的时间复杂度，十分高效。

4. **优化配置下，运行 trace2017 中的两个 trace，结果如下：**

**[01-mcf-gem5-xcg.trace]**

- 运行 trace 共( 10 )遍

- L1 Cache： 平均 Miss Rate = （ **3.30%** ）

- L2 Cache： 平均 Miss Rate = （ **1.73%** ）

- AMAT =（ **1.9953** ）ns

- 用到的优化策略包括 LRU 策略（链表+哈希表）

**[02-stream-gem5-xaa.trace]**

- 运行 trace 共( 10 )遍

- L1 Cache： 平均 Miss Rate = （ **1.42%** ）

- L2 Cache： 平均 Miss Rate = （ **16.80%** ）

- AMAT =（ **2.0028** ）ns

- 用到的优化策略包括 LRU 策略（链表+哈希表）

***该部分数值与实际运行结果一致***