Lab3 Cache Simulator

2019 年 5 月

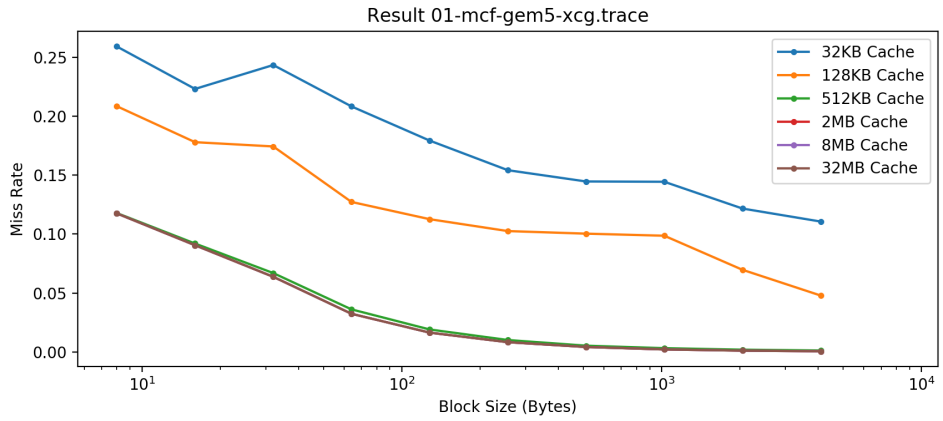
1 单级 Cache 模拟

1.1 Block Size 的影响

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。

以下数据均固定 Cache 为 8 ways 组相联，WriteBack 并 Write Allocate，不加入其它优化技术。数据采样时参数均为 2 的幂次（因此可能会有锯齿表现不出来）。

[01-mcf-gem5-xcg.trace]



这个折线图具有两个值得注意的特点。

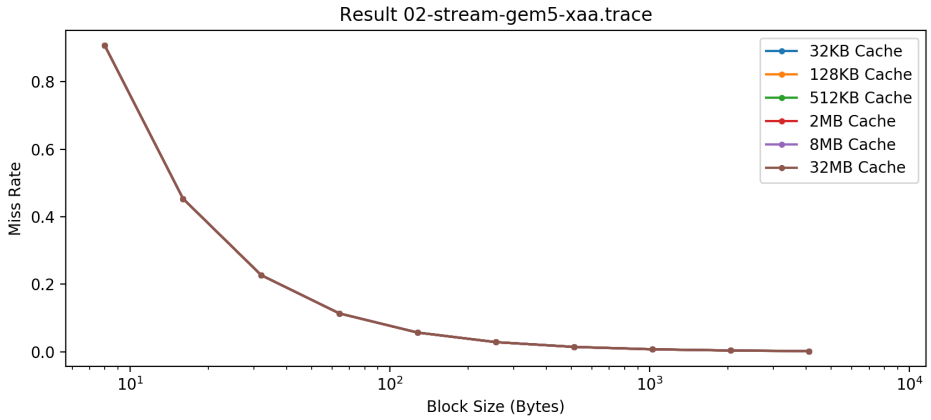
第一是在 32KB 大小的 Cache 时，Miss Rate 在 32B Block Size 时会突然反常地上涨，这个是由于 BlockSize 的上涨**导致 Set 数减少**，原本不冲突的地址发生了冲突不命中。

第二点是在 Cache 大小足够大时，Miss Rate 保持不变。原因是在相同 Block Size

的情况下，大 Cache 会比小 Cache 有更多的 Set，小 Cache 不冲突，大 Cache 也不会冲突。

折线图整体趋势是当 Block Size 不是很大时，Block Size 增加，Miss Rate 减少（当然 Block Size 过大时会导致 Set 数过少而冲突）。而在不同 Cache Size 下，大 Cache 的 Miss Rate 更小。

[02-stream-gem5-xaa.trace]



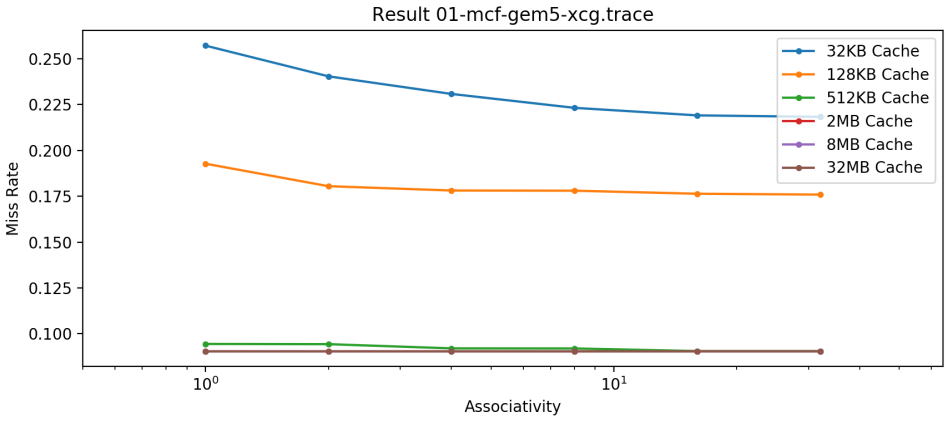
第二个 trace 相当有特点，它在不同 Cache Size 时的 Miss Rate 完全一样，而 Miss Rate 又随 Block Size 增加而缩小。这是源自于这个 trace 本身的特点，该 trace 的访存为三个基地址交替 +8 访问，在之后新增一个地址四个地址交替访问，其中一个地址不再变化。可以预见，当 Set 数足够将这几个地址分离开时，决定 miss 个数的显然就是 Block Size（[Block Size/8 -1] 次 hit）。而观察这些配置，Set 数都足够大，因此表现也相同。

1.2 Associativity 的影响

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

以下数据均固定 Block Size 为 16B，WriteBack 并 Write Allocate，不加入其它优化技术。数据采样时参数均为 2 的幂次。

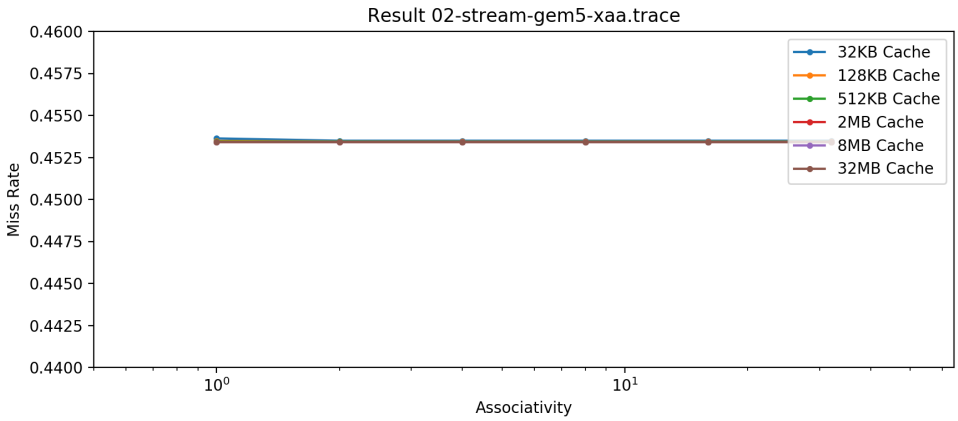
[01-mcf-gem5-xcg.trace]



对于 Cache 较小时，相联度的增加能够减少 Miss Rate，因为可以更加效率地使用 Cache（减少 conflict miss）。但是需要注意到增加组相联数的同时也会减小 Set 数，一样会产生上面提到的 Set 减少而导致冲突的反常现象。组相联数加大到一定数后效果会减缓，因为此时基本是 cold miss 和 capacity miss。

当 Cache 较大时，Set 部分的 Hash 能够将大多数地址从一开始分开，不太容易出现冲突的情况，因此组相联数基本不会产生影响。

[02-stream-gem5-xaa.trace]



第二个 trace 的特点上面提到了，因此组相联数应该也不会影响太大（只要 Set 数不会小到对这四个地址产生冲突），事实上看也的确是近似一条直线。值得揣摩的是既然 Block Size 为 16B，那么一个 16B 块被访问时应该是 miss 一次 hit 一次，Miss Rate 应该在 50% 处，这里稍稍低了一点，是因为后面四地址交替处第四个地址不再变化一直 hit。

1.3 写时策略的影响

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

下面的访问延时单位为 ns，Block Size 为 16B。

[01-mcf-gem5-xcg]

Cache Size	WB & WA	WB & N-WA	WT & WA	WT & N-WA
32KB	7858503	8269539	12534163	12697265
128KB	6938247	7204929	11906361	11949171
512KB	5069511	5355555	10267751	10281905
2MB	5045799	5337033	10245523	10263595
8MB	5045799	5337033	10245523	10263595
32MB	5045799	5337033	10245523	10263595

在这个 trace 里，呈现出一个比较普遍的一个趋势，WriteBack 总比 WriteThrough 要快，而 WriteAllocate 也总比不 Allocate 要快。在 Cache Size 到一定大小时，时间不再变化（Set 数足够大了）。

[02-stream-gem5-xaa]

Cache Size	WB & WA	WB & N-WA	WT & WA	WT & N-WA
32KB	11745414	11424624	11818342	11424730
128KB	11563836	11424624	11818342	11424730
512KB	10705130	11424200	11818342	11424730
2MB	9208304	11424200	11818342	11424730
8MB	9208304	11424200	11818342	11424730
32MB	9208304	11424200	11818342	11424730

在这个 Trace 里，出现了与上面不同的结果，在写穿策略下，写不分配比写分配的效果要更好，原因还是在于 trace 的特殊性和参数的设置，这里 Block Size 为 16B，意味着一个 Block hit 一次 miss 一次后就没它的事了，因此不分配可能节省更多的调入调出时间。

2 CPU 模拟器联调

2.1 编译运行

本次的模拟器基于 Lab 2 模拟器进行修改，主要对原先简陋的存储结构进行了修改，将多级 Cache 加入模拟器，并在 Config 中加入了相应的选项。要想编译含 Cache 的 CPU 模拟器，输入指令 `make cache`，然后`./sim-cache` 运行模拟器。

新增的 flag：

- `-l <level>`: `level` 为 0-3，表示 Cache 存储结构的层级，0 为不使用 Cache，3 为使用三级 Cache
- `-t`: 表示运行一个 `trace` 而不是运行可执行程序

新增的 config(可从 cfg 文件夹中查看样例)：

- `L1C_BUS_CYC`: L1 Cache bus latency 消耗的 CPU cycle
- `L1C_HIT_CYC`: L1 Cache hit latency 消耗的 CPU cycle
- `L1C_SIZE`: L1 Cache 的大小
- `L1C_ASSOC`: L1 Cache 的组相联数
- `L1C_BSIZE`: L1 Cache 的行大小
- `L1C_WT`: L1 Cache 是否写穿
- `L1C_WA`: L1 Cache 是否写分配
- `L1C_WA`: L1 Cache 是否写分配

L2、L3 Cache 配置同上，又：本 lab 的 3 个 config 文件已经存放在`./cfg` 文件夹中

2.2 正确性测试

Lab 提供的配置位于`./cfg/lab3_2.cfg`，使用`-c ./cfg/lab3_2.cfg` 指定配置文件。

同时使用 Lab 2 中编写的流水线模拟器和本次编写的 Cache 模拟器进行结果对照，看结果是否正确。

下面为三级缓存运行 `qsort` 用户程序的结果：

```
~/Documents/CS/archlab: ./sim-cache -f test/qsort -l 3 -c cfg/lab3_2.cfg
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
User program exited.

---------------- Machine ---------------
BASIC STATUS:
- Cycle Count:       29510
- Inst. Count:       22686
- Run Time:          0.0083
- Pipeline Cyc CpI:  1.30
- CPU Cyc CpI:       1.86

- Pred Strategy:     always not taken
- Branch Pred Acc:   49.54% (1248 / 2519)
- Load-use Hazard:   3335
- Ctrl. Hazard:      1271 * 2   (cycles)
- ECALL Stall:       85 * 3 (cycles)
- JALR Stall:        346 * 2    (cycles)

...

----------------- Cache ----------------
L1 Cache:
- Cache Size:        32768
- Block Size:        64
- Set Number:        64
- Associativity:     8
- Replace Method:    LRU
- Prefetch Num:      1
- Access Times:      37549
- Miss Times:        82   (HIT:37467)
- Miss Rate:         0.22 %
  [Write Back]      [Write Alloc]
```

```
L2 Cache:
- Cache Size:        262144
- Block Size:        64
- Set Number:        512
- Associativity:     8
- Replace Method:    LRU
- Prefetch Num:      1
- Access Times:      118
- Miss Times:        82   (HIT:36)
- Miss Rate:         69.49 %
  [Write Back]      [Write Alloc]

L3 Cache:
- Cache Size:        8388608
- Block Size:        64
- Set Number:        16384
- Associativity:     8
- Replace Method:    LRU
- Prefetch Num:      1
- Access Times:      118
- Miss Times:        82   (HIT:36)
- Miss Rate:         69.49 %
  [Write Back]      [Write Alloc]
----------------------------------------

---------------- Memory ----------------
BASIC STATUS:
- Page Size:       4096
- Phys. Mem Size:  81920000 (20000 * 4096)
- Phys. Mem Use:   0.00% (7 / 20000)
- Stack Size:      16384
- Stack Top Ptr.:  0x80000000
----------------------------------------
```

可以发现快排结果正确。

2.3 CPI 统计

user prog.	CPI(with L3Cache)	CPI(with L1Cache)	CPI(without cache)
n!	5.92	5.70	128.93
add	5.07	4.96	132.42
simple-function	5.05	4.94	132.25
qsort	1.86	1.82	128.57
mul-div	4.92	4.82	131.82
matmul	1.38	1.37	127.02

对于 `n!` 等小程序，由于冷启动 miss，一开始访存开销很大，基本 CPI 都在 3 以上。而对于运行时间长且局部性好的程序 `matmul` 的 CPI 就低了很多。此外，可以发现不加入 Cache 的 CPI 都高达 100，是因为在 Fetch 阶段只要不是 bubble 就必定会访存，而访存开销是巨大的。因此真实 CPU 都会增加指令 Cache 并数据预取来缩短消耗的时间。

可以发现一个比较奇怪的现象，一级 Cache 比三级 Cache 的速度还要快，这是因为这些用户程序访问的内存空间都很小，而给定的 Cache 空间很大，一级 Cache 就可以搞定，剩余的两级 Cache 基本都没有用上，反而会增加回写等操作的延迟。

2.4 两次实验 CPI 变化原因

在 Lab2 中，因为考虑到没有加入 Cache 但是又要使得 CPI 尽量接近真实水平，因此对于 Fetch 部分的指令访存全部视为在指令寄存器里，周期为 1。而 Memory 部分的指令考虑到 Cache 的效能，对内存的访问周期设定得较小，所以并不是那么真实。

在 Lab3 中，加入了真实的 Cache 模拟，因此对于相应的周期设定全部放开，对于内存的访存延迟设置到了 100（在 Lab 2 中这个数被设定为 20），并且在 Fetch 阶段的访存也包括在内。所以可以发现，在运行很小的程序时，由于冷启动，CPI 大多都较高。而在运行长程序或局部性较好的程序时（比如上面的 `qsort` 或 `matmul`），CPI 就比较接近 Lab2 了，都是向 1.0 趋近。

3 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。

注：由于实习指导没有明确给出，因此按照去年的要求做，Cache 默认配置为

Level	Capacity	Associativity	Line size	WriteUp Polity	Bus Latency
L1	32 KB	8 ways	64	write Back	0 cpu cycle
L2	256 KB	8 ways	64	write Back	6 cpu cycle

请填写以下参数

- 默认配置下，32nm 工艺节点下，L1 Cache 的 Hit Latency 为（**1.48** ）ns，约等于（**1** ）cycle

- 默认配置下，32nm 工艺节点下，L2 Cache 的 Hit Latency 为（**1.92** ）ns，约等于（**2** ）cycle

默认配置下，运行 trace2017 中的两个 trace，结果如下：

[01-mcf-gem5-xcg]

- 运行 trace 共（ **5** ）遍

- L1 Cache：平均 Miss Rate = （**20.05%** ）

- L2 Cache：平均 Miss Rate = （**38.59%** ）

- AMAT =（ **9.5204** ）

[02-stream-gem5-xaa]

- 运行 trace 共（ **5** ）遍

- L1 Cache：平均 Miss Rate = （**11.35%** ）

- L2 Cache：平均 Miss Rate = （**60.03%** ）

- AMAT =（ **8.0881** ）

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：**4 行数据预取、旁路**

L2 Cache：**双队列替换策略、2 行数据预取**

[双队列]

该算法是一种块的替换算法，是 LRU 的改进算法。导致 LRU 效果不好的一种情况就是组相联数过小，一个非常用地址进来会踢掉之前的一个常用地址。双队列算法是多优先级队列的一种简化版，相当于对一个组中的所有行分为两个队列，一个为只访问了一次的偶然队列（FIFO），一个则为访问了两次或更多的常用队列（LRU）。当一个地址第一次被访问时会先进偶然队列，然后再次访问时会被加入常用队列。在选择牺牲行时，我们会优先考虑偶然队列，然后再考虑常用队列（实质双优先级）。

可以很明显的发现，这个算法有效的前提在于一个常用地址在被踢出偶然队列之前能不能被再次访问，这其实就要求了偶然队列要足够地长，其实也就相当于 Cache 要足够地大。因此这个算法最好不要用在高层级的 Cache 上，在这里，我尝试在 **L2 Cache** 上使用它并的确取得了更好的效果。

多余空间消耗：记录一个行在哪个队列里。电路上的实现会比 LRU 复杂很多。

[数据预取]

数据预取策略通过提前预测将来行为将内容提前载入来提升命中率。一般从空间局部性的角度来分析，当前地址所紧邻的数据更容易被访问到，因此预取一般预取之后的若干行。但是预取的数量显然不是越多越好，过多的预取会占用 Cache 内的空间，甚至把有用的数据挤出去，因此选择一个好的预取量成为了算法的关键。

在测试后，选择在 L1 Cache 使用 4 行的数据预取，而在 L2 Cache 上使用 2 行的数据预取。

[旁路]

旁路是指提前判断是否跳过该级缓存直接请求下级缓存。由于旁路不能轻易决定，在这里我保守地认为当一个行恰好被刚刚踢出时又被访问时应该旁路它，因为这种情况恰好是几个块在进行竞争，我们需要让其中一个块放弃竞争避免更多的换入换出消耗。

优化配置下，运行 trace2017 中的两个 trace，结果如下：

[01-mcf-gem5-xcg]

- 运行 trace 共（**5**）遍

- L1 Cache：平均 Miss Rate =（**17.53%**）

- L2 Cache：平均 Miss Rate =（**28.49%**）

- AMAT =(**6.8225**)

- 用到的优化策略包括 **双队列、数据预取、旁路**

[02-stream-gem5-xaa]

- 运行 trace 共 (**5**) 遍

- L1 Cache：平均 Miss Rate = （**2.84%** ）

- L2 Cache：平均 Miss Rate = （**60.34%** ）

- AMAT =(**2.7737**)

- 用到的优化策略包括 **双队列、数据预取、旁路**