高速缓存实习报告

1600012896 金庆哲

代码架构

本次的实验代码主要根据所给的cache模板实现，同时一些功能写在了之前的Simulator架构中

- storage.h

  模板中的代码文件，定义了storage类，其中有很多的统计数据以及访存的HandleRequest操作

- cache.cc cache.h

  模板中的代码文件，定义了cache类，处理几乎一切和cache有关的行为，包括miss、hit的处理等，同时定义了许多和cache有关的方法，如用SetConfig设置cache，用SetLower将多层cache连接等

  一个重要的数据结构是CacheBlock

```
struct CacheBlock {
  bool valid;
  bool dirty;
  int lru;
  uint64_t tag;
  char data[256];
  CacheBlock() {
    valid = false;
    dirty = false;
    lru = 0;
  }
};
```

  这就是我们熟知的Cache中的一项，包括了数据和一些标志位（valid、dirty、tag），每一个Cache类要维护自己的CacheBlock数组，对它进行各种操作

  这里使用lru作为block的替换方法

  另一个数据结构是CacheConfig

```
class CacheConfig {
  public:
    int blocksize;
    int capacity;
    int associativity;
    int blocknum;
    int set_num; // Number of cache sets
    int write_through; // 0|1 for back|through
    int write_allocate; // 0|1 for no-alc|alc
    CacheBlock* blocks;
    CacheConfig() {}
```

```
    CacheConfig(CacheBlock* b, int blocksize, int assoc, int capacity, int writet, int writea) {
      this->blocksize = blocksize;
      blocks = b;
      associativity = assoc;
      this->capacity = capacity;
      this->set_num = capacity / (assoc * blocksize);
      this->blocknum = this->set_num * assoc;
      write_allocate = writea;
      write_through = writet;
    }
}
```

- MM.cc MM.h

  在之前的Simulator实现中，它们是用来定义和内存访问相关的变量和方法的，在这里增加了继承Storage的Memory类，主要是重载它的HandleRequest方法从而使多层cache能和内存连接在一起，这里的HandleRequest主要使用memcpy来实现

这里面最重要的就是HandleRequest方法，其模板如下

```
void HandleRequest(uint32_t addr, int bytes, int read,
                           char* content, int &hit, int &time)
```

我们需要传入的参数包括地址、按字节计算的访存规模、读写标记、内容content（读的时候读到这里，写的时候content是要写的内容）、代表是否命中的hit、这次操作所用的总时间time（hit和time传入的是引用，因此在函数操作中它们的值会改变）

对于cache层而言，这个函数所要做的就是根据地址搜索对应的内容是否在cache中，如果在，就读写；如果不在，就要根据不同的策略进行不同的操作

对于memory层而言，这个函数要做的就是根据地址访存

对于cache层而言，我们需要算出地址中的tag、set、offset部分，然后在cache中根据set号和相连度定位到对应的set，再查看这个set中是否有对应的tag，如果找到了这个tag并且block可用的话，就说明访问命中，直接进行读写就可以了（如果是writeback的话就只需要设置dirty位，如果是writethrough要对下层也使用HandleRequest进行写操作）；比较麻烦的是miss的情况，首先要找到一个放置对应块的位置（如果有空位放到空位中，没有空位就根据替换策略选择），如果是读的话就从下一层读这个块到对应位置即可，如果是写的话就要看写策略，如果是Write Allocate的话就要将对应块读入再写，如果不是的话就向下层写即可

单级Cache模拟

首先实现的是单层cache

这里在main.cc中加入有关cache test的部分，命令为

```
./Simulator -c <tracepath> <testnum>
```

根据不同的testnum进行不同的测试，这里的测试号为1

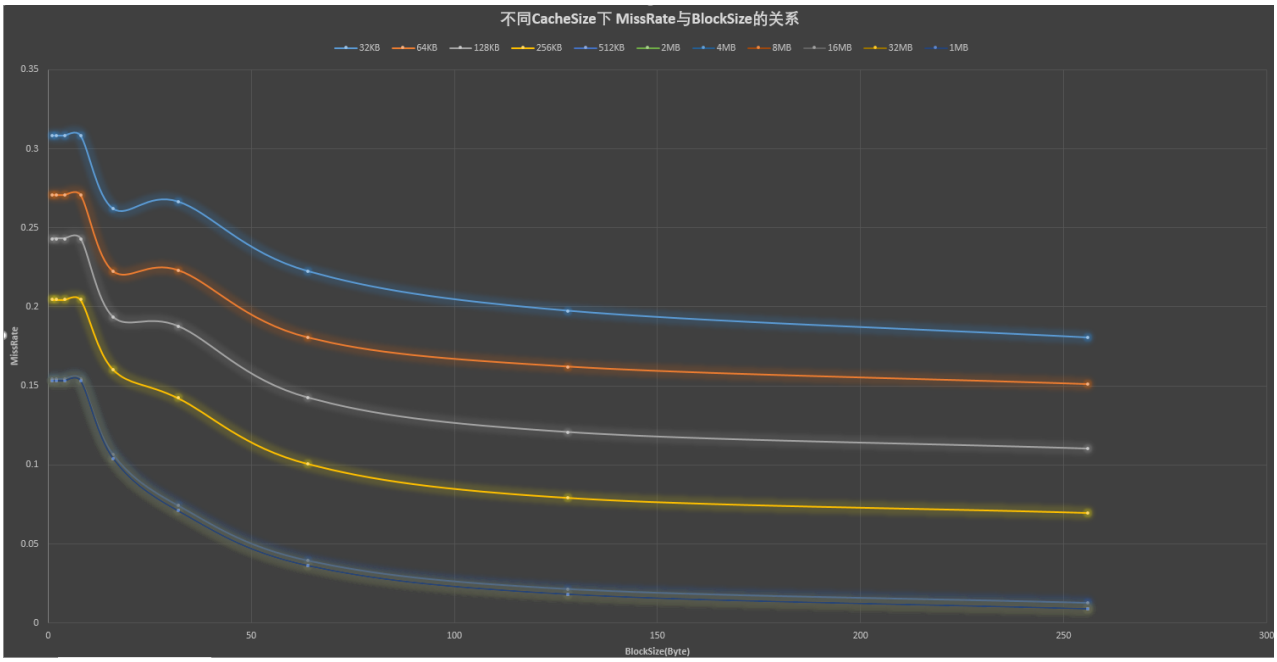
我们将内存空间全部置为0, trace中的r代表读操作，我们就从指定地址读1个byte的数据，w代表写操作，我们就向指定位置写一个byte的0

这里设置cache的hit latency为1，memory的hit latency为20（一个相对数值）

由于作业要求画图，受到另一位同学的启发，想到了可以将输出数据存成csv格式，从而可以使用excel进行简单的画图操作（输出文件为trace文件夹下的01-mcf-gem5-xcg.csv）

这里使用的trace文件是01-mcf-gem5-xcg

- **1**



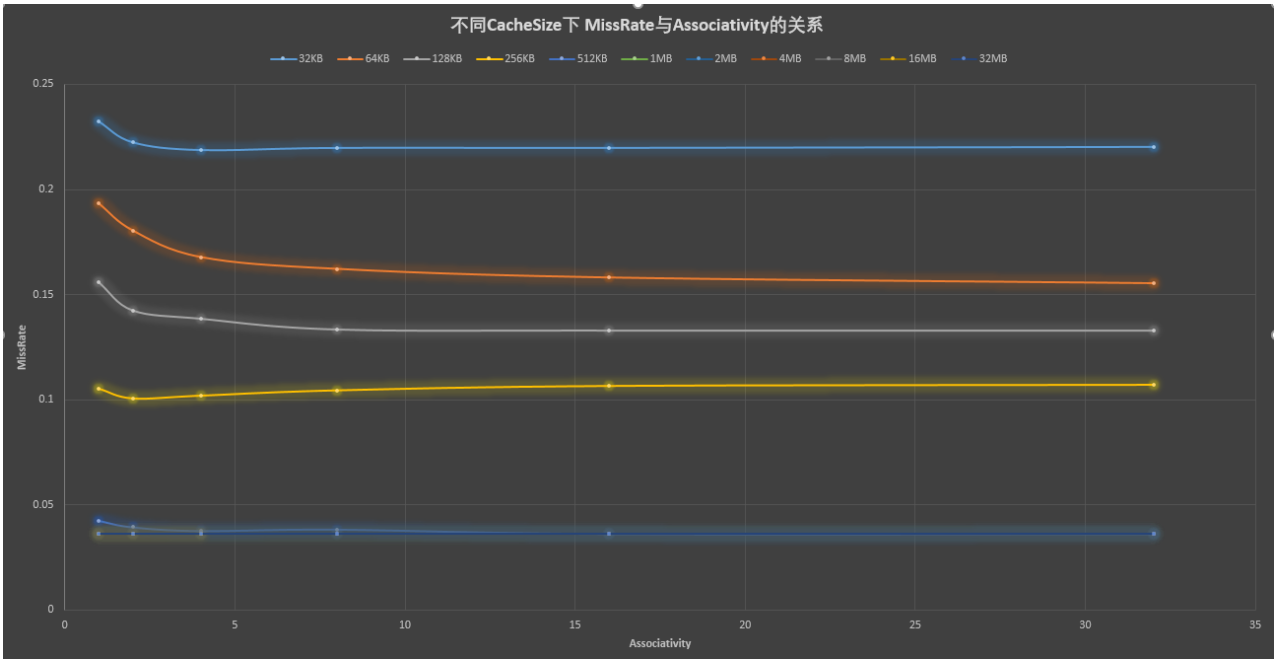
首先是不同CacheSize下MissRate和BlockSize的关系

这里相连度为8，写策略是WriteBack和Non-Allocate

可以看到，在Cache比较小的时候，随着BlockSize的增加，missrate大体为下降趋势，因为Blocksize越大，Block中储存的数据就越多，miss的概率就越小（这里32处的反常应该和trace的访问逻辑有关），而此时在BlockSize相同的情况下，CacheSize越大，missrate越小，这是因为不用过早的将block替换出cache

而Cache大到一定程度的话，CacheSize对于missrate几乎没有影响（因为已经足够大了，根本无法填满），而如果BlockSize很大的话，几乎保证不会miss（cache中很快的充满了所有所需的block）

- **2**

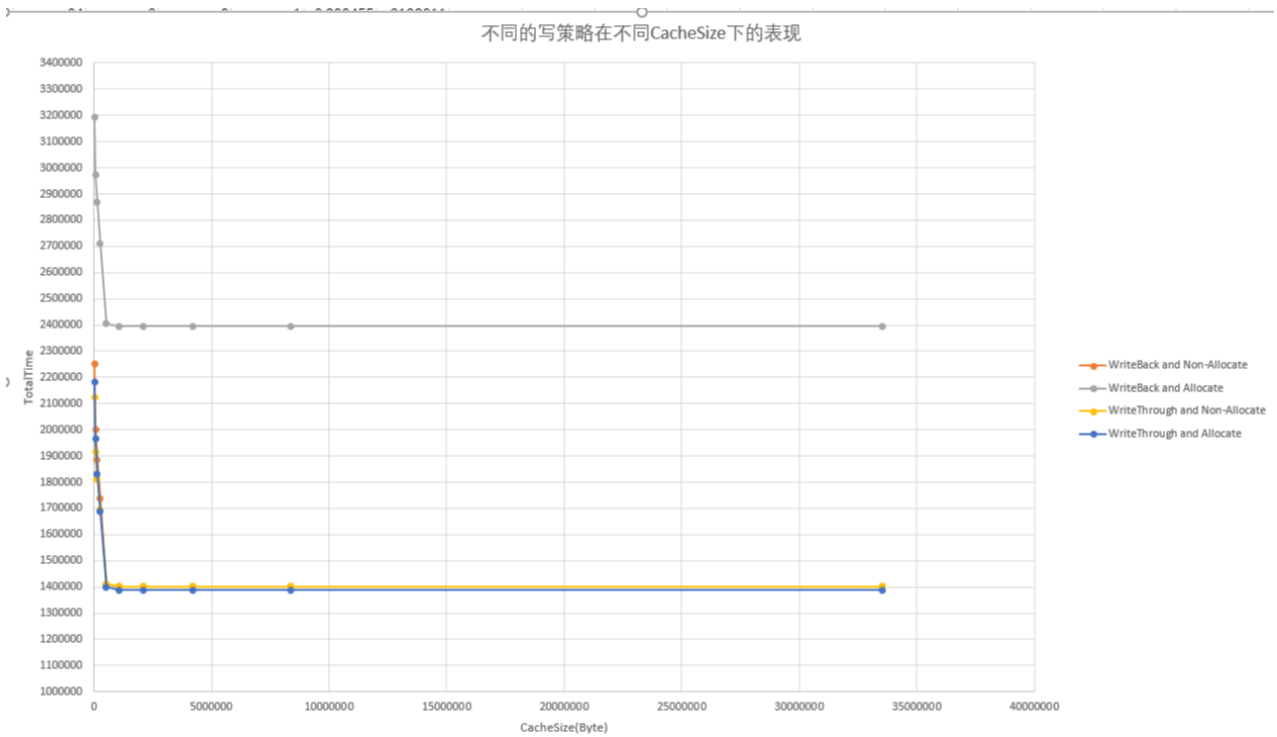


然后是不同CacheSize下MissRate和相连度的关系， 这里固定BlockSize为64B，策略为WriteBack和Non-Allocate

可以看到在Cache比较小的时候，随着相连度的增加MissRate基本呈现先下降后不变的趋势，但是在相连度为4的时候256KB的CacheSize会出现反常上升情况，应该也是和trace的特定访问逻辑相关；相同的相连度下，比较大的Cache的MissRate会比较小（理由和1中相同）

如果Cache大到一定程度，MissRate就不会受相联度的影响（MissRate完全没有变化），这是因为由于cache很大，set很多，所以block基本都分散在不同的set中，不受相联度影响

- **3**



这是不同的写策略在不同的cachesize下的表现，可以看到WriteBack和Non-Allocate共同作用下的访问时间要明显长鱼其他三个，而剩下三种策略表现十分接近，在不同的CacheSize下顺位交替变化

因为多次写入统一缓存时，WriteAllocate可以配合WriteBack，但是WriteThrough却和WriteBack逻辑相违背，导致性能下降，访问时间增加。

和CPU模拟器联调

实现了前面的准备工作后，这里的实现也很简单

我们只需要根据配置定义三层cache，用SetLower将它们连接起来，同时更改之前的memory阶段的访存操作为Memory的HandleRequest方法即可

使用cache就在Simulator运行elf文件时加入-u参数

这里为了和不使用cache进行对比，我们需要给不用cache的访存也加入一个latency，这里设置为25cycles（略大于llc，比l1大得多）

然后我们先来看看运行程序的正确性，和之前一样，我们使用有输出结果的ack.riscv进行测试（放置在elfs文件夹中）

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator  e
lfs/ack.riscv -u
A(0,0)=1
A(0,1)=2
A(0,2)=3
A(0,3)=4
A(1,0)=2
A(1,1)=3
A(1,2)=4
A(1,3)=5
A(2,0)=3
A(2,1)=5
A(2,2)=7
A(2,3)=9
A(3,0)=5
A(3,1)=13
A(3,2)=29
A(3,3)=61
Simulator Exiting!
-----------------------------------
-----------------------------------
Cycles: 1042163
Instructions: 99033
CPI: 10.5234
Good Predict: 3292
Bad Predict: 1753
Predict Acc: 0.6525
Mem Load Hazards: 48535
Control Hazards: 91331
```

可以看到能够正确的输出结果，说明运行没有问题

而CPI为10.5234

再来看看不用cache的CPI如何

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator  e
lfs/ack.riscv
A(0,0)=1
A(0,1)=2
A(0,2)=3
A(0,3)=4
A(1,0)=2
A(1,1)=3
A(1,2)=4
A(1,3)=5
A(2,0)=3
A(2,1)=5
A(2,2)=7
A(2,3)=9
A(3,0)=5
A(3,1)=13
A(3,2)=29
A(3,3)=61
Simulator Exiting!
-------------------------------------
-------------------------------------
Cycles: 1069531
Instructions: 99052
CPI: 10.7977
Good Predict: 3293
Bad Predict: 1754
Predict Acc: 0.6525
Mem Load Hazards: 48535
Control Hazards: 91360
```

CPI为10.7977，这说明在ack这种访问局部性很好的程序中，引入cache可以降低CPI，加速操作

再来看看上次的五个测试程序结果如何

Program	Cache	No-Cache
simple-function	6.4883	6.0152
qsort	6.2052	11.4693
n!	5.8680	5.1249
mul-div	6.2740	5.8235
add	6.4220	5.9560

可以看到，除了qsort这个局部性很好的程序使用Cache会大幅度降低CPI之外，剩下的程序使用Cache都会带来CPI的升高，这说明在当前的Cache设置和访存延时设置下，需要很好的局部性和比较复杂的逻辑才会让使用Cache带来好处，而一些过于简单的小规模程序可能不使用Cache是更佳的选择

高速缓存优化

- 1

  首先需要使用cacti65来查看默认配置的latency，修改cache.cfg，运行./cacti -infile cache.cfg得到latency

- L1

```
Cache Parameters:
    Total cache size (bytes): 32768
    Number of banks: 1
    Associativity: 8
    Block size (bytes): 64
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 32

    Access time (ns): 1.47944
    Cycle time (ns):  2.1629
```

  hit latency为1.47944ns，如果CPU的频率为1GHZ，那么约等于1.5个周期

- L2

```
Cache Parameters:
    Total cache size (bytes): 262144
    Number of banks: 1
    Associativity: 8
    Block size (bytes): 64
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 32

    Access time (ns): 1.9206
    Cycle time (ns):  3.46728
```

  hit latency为1.9206ns，约等于2个周期

- **2**

  依旧在main的CacheTest中进行测试，testnum为2，假设访存需要50个cycle

  操作为

```
./Simulator -c ./trace/01-mcf-gem5-xcg.trace
...
```

  运行多次后得到二者missrate（Write Allocate配合Write Back，事实上如果使用Non-Allocate效果会差很多）

  这是Non-Allocate的效果

```
[howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -
c ./trace/01-mcf-gem5-xcg.trace 2
L1 missrate:0.171493
L2 missrate:0.145257
[howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -
c ./trace/02-stream-gem5-xaa.trace 2
L1 missrate:0.269464
L2 missrate:0.4843
```

  这是Allocate的效果

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -
c ./trace/02-stream-gem5-xaa.trace 2
L1 missrate:0.0808691
L2 missrate:0.104882
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -
c ./trace/01-mcf-gem5-xcg.trace 2
L1 missrate:0.156494
L2 missrate:0.107908
```

- 01-mcf-gem5-xcg.trace

  L1：平均 Miss Rate = 0.156494

  L2：平均 Miss Rate = 0.107908

  AMAT = （1.5 + 6） * （1-0.1565）+ （1.5 + 2 + 6）*0.1565 * （1- 0.108） + （1.5 + 2 + 50 + 6）*0.1565 * 0.108 = 8.6581

- 02-stream-gem5-xaa.trace

  L1：平均 Miss Rate = 0.0808691

  L2：平均 Miss Rate = 0.104882

  AMAT = 8.08（计算逻辑同上，本质就是算期望）

- **3**

下面的优化策略都实现在cache.cc当中。为了选择不同的策略，给CacheConfig增加了一些和策略相关的参数

- 置换策略改变

  事实上在我们能实现的范围内，lru基本上可以说是最优的一种置换策略了，其余的置换策略还包括随机置换、FIFO等，而随机置换虽然效率很高、但是效果是最差的，因此我们这里考虑尝试一下FIFO算法的效果

  下图是L1使用FIFO，L2使用LRU的结果

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
 ./trace/01-mcf-gem5-xcg.trace 2
L1 missrate:0.167599
L2 missrate:0.107341
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
 ./trace/02-stream-gem5-xaa.trace 2
L1 missrate:0.0874969
L2 missrate:0.103409
```

  这个情况下，两个trace中L1的命中率都大辐下降，L2的命中率都略微上升，效果不如LRU

  下图是L1使用LRU，L2使用FIFO的结果

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
 ./trace/01-mcf-gem5-xcg.trace 2
L1 missrate:0.156494
L2 missrate:0.1092
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
 ./trace/02-stream-gem5-xaa.trace 2
L1 missrate:0.0808691
L2 missrate:0.104882
```

  可以看到，trace2中没有变化，trace1中L1不变、L2的missrate略微上升

  下图是L1、L2均使用FIFO的结果

```
[howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
  ./trace/01-mcf-gem5-xcg.trace 2
L1 missrate:0.167599
L2 missrate:0.107341
[howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
  ./trace/02-stream-gem5-xaa.trace 2
L1 missrate:0.0874969
L2 missrate:0.103409
```

可以看到效果明显差于LRU

综上，我们可以得出结论，在这两个trace中使用FIFO并不会带来效率上的提升，因此我们还是考虑使用LRU替换算法

- 数据预取

使用LRU的策略，对L1、L2分别设置数据预取大小为0、1、2、3、4个block，对于trace02运行效果如下（太长只截取一部分），这里用公式直接算出了AMAT的值，以便选择最优解

```
howllow@HowllowdeMacBook-Pro ~/SharedWithUB/simulator ±master⚡ » ./Simulator -c
 ./trace/02-stream-gem5-xaa.trace  2
------L1 prefetch:0 Blocks, L2 prefetch:0 Blocks------
L1 missrate:0.112937
L2 missrate:0.166
AMAT:8.66325
AverageCycles:12807234
------L1 prefetch:0 Blocks, L2 prefetch:1 Blocks------
L1 missrate:0.112937
L2 missrate:0.0554138
AMAT:8.03879
AverageCycles:11519302
------L1 prefetch:0 Blocks, L2 prefetch:2 Blocks------
L1 missrate:0.112937
L2 missrate:0.0369605
AMAT:7.93458
AverageCycles:11279269
------L1 prefetch:0 Blocks, L2 prefetch:3 Blocks------
L1 missrate:0.112937
L2 missrate:0.0277428
AMAT:7.88253
AverageCycles:11158948
------L1 prefetch:0 Blocks, L2 prefetch:4 Blocks------
L1 missrate:0.112937
L2 missrate:0.0221997
AMAT:7.85123
AverageCycles:11086738
------L1 prefetch:1 Blocks, L2 prefetch:0 Blocks------
L1 missrate:0.0376783
L2 missrate:0.149524
AMAT:7.85705
AverageCycles:12021499
------L1 prefetch:1 Blocks, L2 prefetch:1 Blocks------
L1 missrate:0.0376783
L2 missrate:0.0748305
AMAT:7.71633
AverageCycles:11300791
------L1 prefetch:1 Blocks, L2 prefetch:2 Blocks------
L1 missrate:0.0376783
L2 missrate:0.0499248
AMAT:7.66941
AverageCycles:11661058
------L1 prefetch:1 Blocks, L2 prefetch:3 Blocks------
L1 missrate:0.0376783
L2 missrate:0.0250352
AMAT:7.62252
AverageCycles:11300704
------L1 prefetch:1 Blocks, L2 prefetch:4 Blocks------
L1 missrate:0.0376783
L2 missrate:0.0333101
AMAT:7.63811
AverageCycles:11540911
------L1 prefetch:2 Blocks, L2 prefetch:0 Blocks------
L1 missrate:0.0251331
L2 missrate:0.149549
AMAT:7.7382
AverageCycles:11558927
```

最后得到最优的情况是都预取4个block

L1 missrate为0.0150884

L2 missrate为0.0300065

AMAT为7.55281

而对于trace01也是类似，得到最优情况为L1预取4个Block，L2预取3个Block

L1 missrate为0.127189

L2 missrate为0.108924

AMAT为8.44708

可以看出数据预取对于trace02的提升要远大于L1，这是因为02中的trace拥有更好的局部性

同时从trace01的结果中可以看出，并不是预取的block数量越多越好，当Block数量足够匹配局部性时，增大Block数可能反而会增加missrate

（这里还验证了替换L1、L2的策略为FIFO并且和数据预取结合的情况，效果确实不会优于LRU）

- cache旁路

  这里使用的cache旁路策略是，当cache miss时，以一定的概率选择是否绕过cache直接从内存读写（随机使用rand和srand），而且这里只能为L1 Cache设置旁路（因为给L2设置旁路没有意义，L2本来就要从内存读写）。

  测试结果表明，在各种概率下，无论是trace01还是trace02，使用旁路策略的平均miss rate和total cycle都要高于不使用旁路（trace的局部性很好，同时使用数据预取的策略时一次绕过cache带来的损失很大）

  因此我们也不选择使用cache旁路

综上所述，我们选用数据预取策略，由于没有随机性，只运行一次即可

01-mcf-gem5-xcg

L1 Cache Miss Rate:0.127189

L2 Cache Miss Rate:0.108924

AMAT:8.44708

L1数据预取4个block，L2数据预取3个block

02-stream-gem5-xaa

L1 Cache Miss Rate:0.0150884

L2 Cache Miss Rate:0.0300065

AMAT:7.55281

L1数据预取4个block，L2数据预取4个block