Homework 1: Cache Simulator

Minsoo Rhu, Jingwen Leng

Fall 2012

1 Cache Simulator Overview

The source files for our cache simulator are listed in Table 1:

Name	Description
Cache.cpp Cache.h misc.h	contains the definition and declarition of Cache class
main.cpp	main function for multicore cache
singlecache.cpp	main function for single cache
autotest.cpp	genrating the trace to test the cache hierachy and paramters.
Makefile	make file for cache simulator

Table 1: Source files in our cache simulator

1.1 Simulator Input

The make file is going to generate two binaries: ***cachesim*** and ***singlecache***. To run the single cache, use the following command line:

./singlecache 32768 32 2 5 trace-file-name

To run the multi core cache, use the following command line:

./cachesim 4 32 1024 4 4 4 4 trace-file1 trace-file2 trace-file3 trace-file4

The meaning of each parameter is the same as the home work specification.

1.2 Simulator Output

Figure 2 and Figure 2 show the sample output from our cache simulator. The access counts, hit/miss counts and hit/miss ratio are outputed for L1 and L2 cache, as well as for the individual bank in L2.

1.3 Simulation Results

Name	Total Access	Total Hit	Total Miss	Hit Rate (%)	AMAT
gcc	8922	0	8922	0	205
dealII	9601	224	9377	2.33	200.42
mcf	43995	650	43345	1.47	202.23
soplex	27333	975	26358	3.57	198.06

Table 2: Simulaion results for the single cache

Table 2 shows the simulation results for single L1 cache with the 32768 Bytes capacity, 32 Bytes block, associativity 2, hit latency 5 cycles and miss latency 200 cycles. The simulation results for multi core cache are shown and discussed in Section 3.2.

```
NumLines to Simulate=Whole File
NumLines to Simulate=-1
Current CPU-CYcle=23915706


=====================================
[Core-0] Stats
- Num of access    = 12190
- Num of serviced = 12190
-------------------------------------
- Total latencies accumulated  = 283378
- (AMAT) Average Mem Access Time = 23.246760
=====================================

[Core-0][L1 Cache][BankId=1][Stats]
- Total Accesses  = 12190
- Total Hits      = 1417
- Total Misses    = 10773
=> Hit  rate      = 0.116243
=> Miss rate      = 0.883757
```

Figure 1: Sample out for single cache.

```
[Core-726][L2 Cache][BankId=0][Stats]
- Total Accesses  = 4014
- Total Hits      = 2636
- Total Misses    = 1378
=> Hit  rate      = 0.656702
=> Miss rate      = 0.343298
[Core-726][L2 Cache][BankId=1][Stats]
- Total Accesses  = 4867
- Total Hits      = 3009
- Total Misses    = 1858
=> Hit  rate      = 0.618245
=> Miss rate      = 0.381755
```

Figure 2: Sample out for multi core cache.

2 Description

Single Cache Module. The core cache module used in our multi-core cache simulator is constructed in a way such that it is easy to debug and re-usable on multiple cache-levels (Figure 3(a)). The class structure of a cache module consists of i) a tag-array class that stores all the necessary information required to track a cache line's state (i.e. tag, line status, accessed time, etc), ii) an incoming-access-request-queue structure (using C++ STL containers) that stores cache-access requests from the core (in case of L1D caches) or the L1D caches (in case of L2D cache), iii) an output-serviced-request-queue structure that spits out serviced request, along with whether it

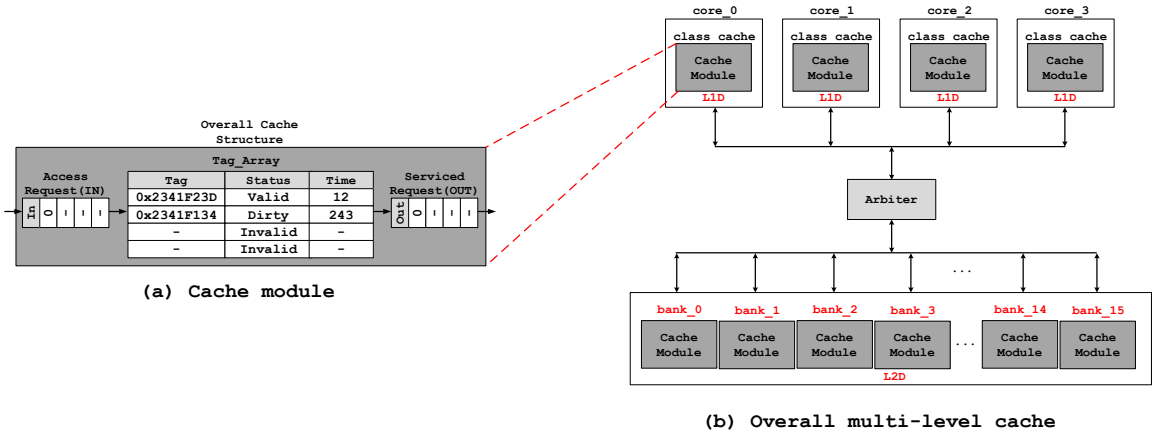


Figure 3: Overall structure of the multi-core cache simulator

was a hit/miss, and iv) access-methods that manipulate the cache module (i.e. probe(), access(), send_serviced_request, service_incoming_request() and etc). All the abovementioned modules that constitute a cache are structured in a way that operates in a *cycle-level* manner, so whenever a single clock cycle is elapsed, an internal *advance_cycle()* method is called. This method executes the necessary actions that need to be taken in order to *advance* one clock cycle further, thus allowing the overall cache simulator to not only functionally emulate the cache, but also to capture the cycle-level behavior.

Multi-Level Cache. The cache module explained above is used to instantiate a *single* bank amount of hardware that constitutes a cache, so each core instantiates *one* cache module to mimic its single-bank L1D cache. Accordingly, based on the required specifications of HW1, a total of four banks are instantiated for L1D caches across the four multi-cores – note that each banks work independently among each other in L1. Using the same cache module, we additionally instantiate *16* cache modules to provide the behavior of the 16-bank L2D cache. Based on how the L2 bank configuration is selected (i) equally partitioned, ii) unequally partitioned, and iii) shared L2 banks configuration), the *arbiter* coordinates the incoming requests from L1D to the appropriate L2D banks. As in the cache module, the arbiter also has an *advance_cycle()* method that enables its cycle-level behavior.

3 Evaluation

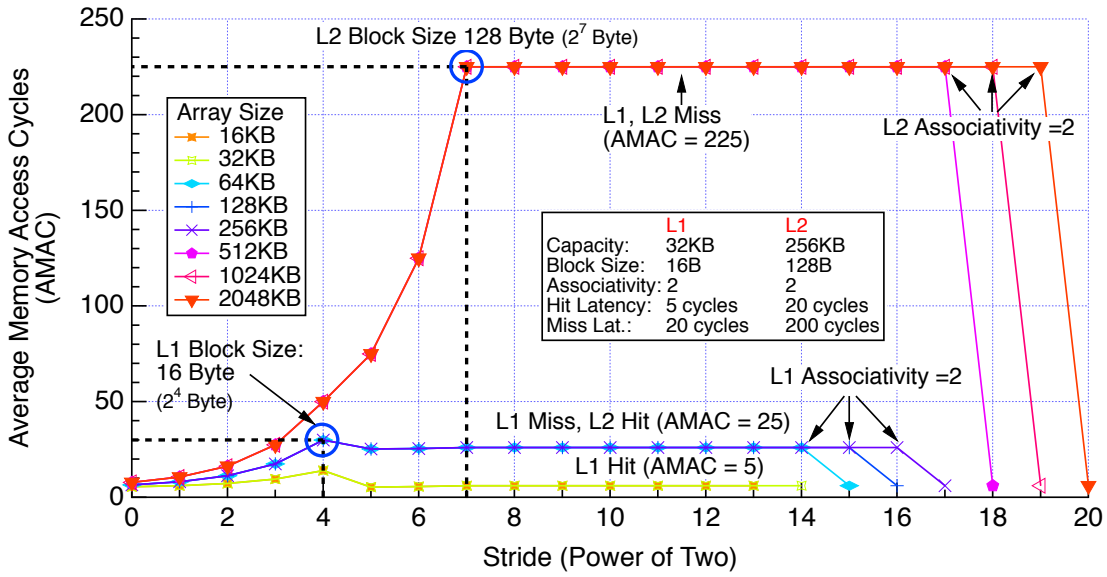
3.1 Single Cache

To test our single cache (core) version of the cache simulator, we generate the memory access trace for program shown in Figure 4, which uses a fixed stride to access an array (each element is one byte).

```
char *Array;
unsigned long stride;
for (unsigned long i=0; i < 500; i ++) {
  for (unsigned long index=0; index < Array_Size; index+=stride) {
    Load Array[index]
  }
}
```

Figure 4: Program to automatically test cache paramters.

Figure 5 shows the results for the program with different array size and stride mentioned previously. We consider this is the signature dictionary of our cache. Next, we are going to explain how to get the cache parameters through the graph. We have also tested the scenario with L2 cache disable, but we only show the results for the single case with both L1 and L2 cache. Parameters for both cache are shown in the Figure.



	L1	L2
Capacity:	32KB	256KB
Block Size:	16B	128B
Associativity:	2	2
Hit Latency:	5 cycles	20 cycles
Miss Lat.:	20 cycles	200 cycles

Figure 5: Average memory access cycles for different array size and stride.

Hit/Miss Latency Figure 5 shows the average memory access cycles for different scenarios, i.e. L1 hit, L1 miss but L2 hit and L2 miss. The average access cycles for these scenarios are 5, 25 an 225 respectively.

Cache Capacity As shown in the Figure 5, when the array size is 16KB and 32KB, the array size is smaller or equal than the L1 cache capacity (32KB in the Figure). Thus the L1 cache is able to hold the whole array, which result in always L1 hit (except the initial cold miss). Similarly, for array size 64KB, 128KB and 256KB, which are smaller or equal than the L2 capacity 256KB but bigger than the L1 cache capacity, the access is L1 miss but L2 hit. When the array size is bigger than both L1 and L2 size, the memory access will be miss in the L2 cache.

Block Size As shown in the Figure 5, when the stride to access memory changes from 8 (2^3) Bytes to 16 (2^4) Bytes, the memory access result in completely L1 miss. This is because the L1 cache block size is 16 Bytes. Access stride bigger than the block size would result in the completely L1 miss. Similarly, the transition point for L2 miss when access stride is 128 (2^7) Bytes matches the L2 cache block size (128 Bytes).

Cache Associativity In the Figure 5, when the access stride to access memory changes from (2^{14}) Bytes to (2^{15}) Bytes at array size 64KB, the memory access changes from L1 miss to L1 hit, although the array size is bigger than the L1 cache size. When the array size is 64KB (2^{16} Bytes) and access stride is (2^{14}) Bytes, the program shown in Figure 4 is only accessing four elements in the array (with index 0, 16KB, 32KB and 64KB). However, these addresses have same index and the associativity for L1 cache is 2, each access is suffering conflict miss. When access stride becomes 2^{15} Bytes, which is the half size of the array, the program is only accessing two elements in the array. Although they still access the same set in the cache, associativity 2 is enough to guarantee cache his in this scenario. This also applies the array size 128KB and 256KB. For the L2 cache, the transition point for array size 512KB, 1024KB and 2048KB matches the L2 associativity as well.

3.2 Multicore Cache

In this homework, the L1 cache is private for each core, thus there is no difference between single core and multi core version. The main difference between these two scenarios come from the L2 cache: it can be equally partitioned, unequally partitioned or shared by all cores. Section 3.1 already focuses on the evaluation of the cache hierarchies. Thus for the multicore cache part, we only focus on the evaluation of different L2 partition/shared cases.

Partitioned L2 Cache Typically L2 cache has multiple banks with one read port and one write port to achieve the same effect with a large L2 cache with multiple

Core ID	Bank ID	Four Traces	gcc Core 0	dealII Core 1	mcf Core 2	soplex Core 3
Core 0	0	4014	4014	0	0	0
	1	4867	4867	0	0	0
	2	4138	4138	1	1	1
	3	4019	4019	0	0	0
Core 1	4	3926	0	3926	0	0
	5	4060	0	4060	0	0
	6	4420	1	4420	1	1
	7	4028	0	4028	0	0
Core 2	8	20563	0	0	20563	0
	9	20693	0	0	20693	0
	10	20616	1	1	20616	1
	11	20264	0	0	20264	0
Core 3	12	11983	0	0	0	11983
	13	12914	0	0	0	12914
	14	12792	1	1	1	12792
	15	11735	0	0	0	11735

Table 3: L2 bank isolation test.

read/write ports, but with much less area/power overhead. When the L2 cache is partitioned among multiple cores, each core will be assigned with multiple banks that are dedicated to serve that core. The number of L2 cache banks assigned to each core can be equal or unequal. In the partitioned L2 cache, a core is not allowed to access banks assigned to other cores. To validate this property, we first simulate the scenario that four different traces are assigned to four different cores. Then we feed only one trace. (empty traces are fed to other cores)

Table 3 shows the result for the bank access isolation test. We used equally partitioned L2 cache for four cores, and each core is assigned with 4 banks. The first two columns of the Table show how banks are assigned to each core. The other columns show the access counts to each bank under different cases. In the *Four Traces* scenario, *gcc* trace is fed to core 0, *dealII* trace is for 1, *mcf* for core 2 and *soplex* for core 3. In the *gcc Core 0* case, gcc trace is fed to core 0 and other cores are fed with a "pseudo" empty trace with only one memory access (that's why other cores have one access in the L2), similarly for other three scenarios. The access counts in Table 3 show that in the partitioned case, a core is not able to access other cores' banks.

Shared L2 Cache In shared L2 cache, each core is free to access any bank. Here, we compare the metric of *Average Memory Access Time* (AMAT) for the shared L2 cache with partitioned L2 cache.

We used 16-banked L2 cache for 4 cores. The trace fed to each core is the same in the third column of Table 3. Figure 6 shows the hit rate for each bank for different schemes. For equal partitioned, we can clearly see core 0 and core 1 have the highest bank hit while core 3 has the lowest hit rate. In the shared L2 cache scheme, each bank has a more evenly distributed bank hit rate. With the knowledge that core 3 has

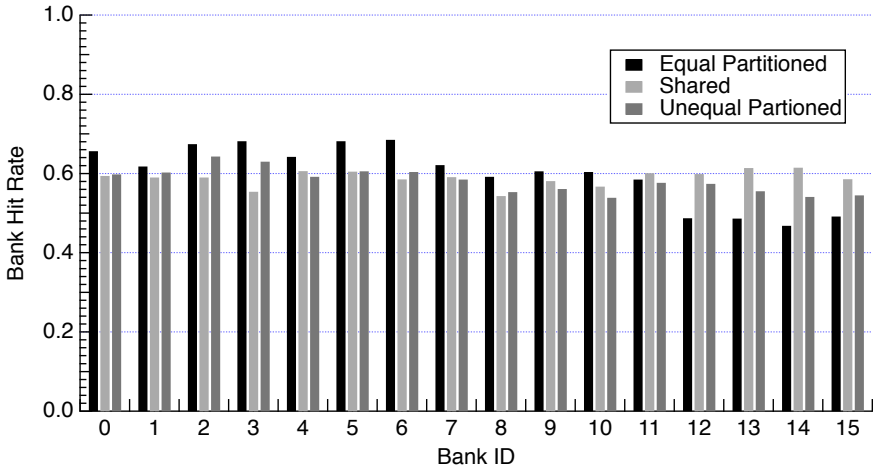


Figure 6: Bank hit rate for different L2 cache scheme.

the lowest hit rate, we increase the number of banks assigned to core 3 to 8 (resulting the bank allocation for each core 2 2 4 8, still 16 total banks). The result in Figure 6 shows the improvement of bank hit rate for core 3. Core 0 and core 1 are not affected that much since they don not have the desire to a larger cache.

Figure 7 shows the AMAT for each benchmark. We can see the shared L2 cache and the unequal partitioned L2 cache with the knowledge of benchmark characteristics can reduce the AMAT for benchmark *soplex* without affecting other program's performance, resulting a smaller AMAT averaged for these 4 benchmarks.

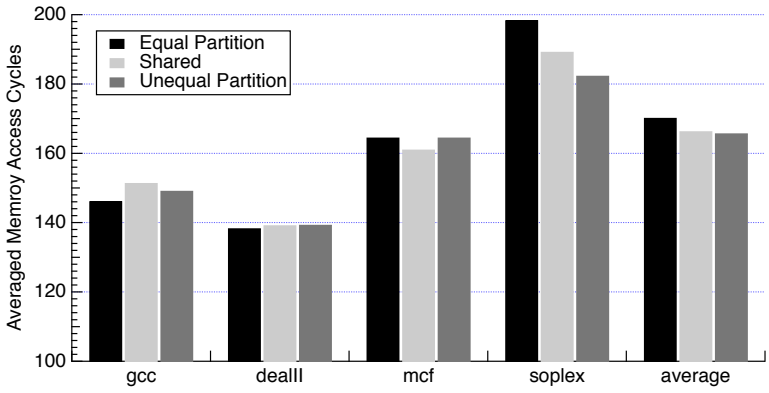


Figure 7: AMAT for different benchmarks.