COMPUTER ARCHITECTURE

Project 1

Hua Jiang

Rivier University
Graduate School
Mathematics & Computer Science Department
Instructor: Prof. Dave Pitts
CS 556AH1 Computer Architecture

Fall, 2014

CONTENTS

1 Goal

In this project, I independently developed a program that simulates cache memories. I used my program with memory traces to produce statistics to measure the performance of the simulated cache. My simulator is able to be configurable to allow experiment with a number of scenarios in order to compare the performance of different cache designs.

1.1 Configuration

My cache simulator is a command line based Java program. It takes in two kinds of configuration, i.e. unified cache and split cache. I treat long and int types very carefully here,

1) *UoSC=U UBlockSize=<int> USetSize=<int> UNumBlocks=<long> UNT=<double> MissPenalty=<double>*

Here "U" stands for unified cache where UBlockSize is how many bytes for each block, USetSize is the associativity of the cache, UNumberBlocks is the total number of blocks in the cache, UNT is user specified hit time in nano second and MissPenalty is also in nano second.

2) *UoSC=SC IBlockSize=<int> ISetSize=<int> INumBlocks=<long> DBlockSize=<int> DSetSize=<int> DNumBlock=<long> UNT=<double> MissPenalty=<double>*

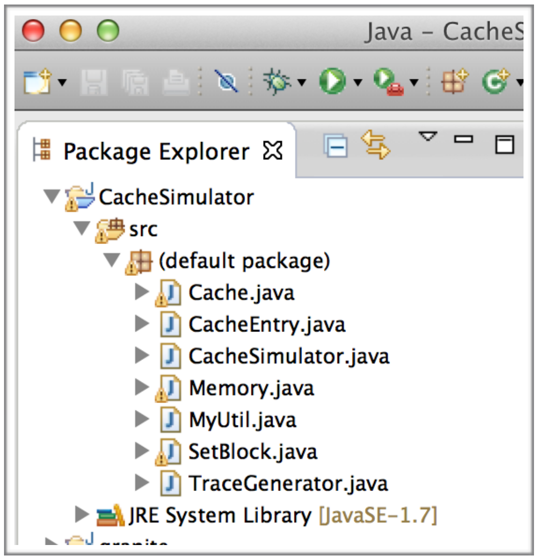
Here "SC" stands for split cache where every parameter starting with "I" means instruction cache configuration while "D" means data cache.

1.2 Cache Details

My simulator adopts write-back scheme, i.e. only update memory when data in cache gets override. Each cache block has two boolean variables (valid field and dirty field), tag field containing tag information of current memory block, and data block containing the block bytes. The simulator utilizes the LRU replacement policy.

2 Implementation

I used Java to realize this cache simulator. Eclipse is the platform I am working on for this project. So far the cache simulator is consisted of CacheSimulator.java, Memory.java, Cache.java, SetBlock.java, CacheEntry.java, and MyUtil.java as well as a trace file generator called TraceGenerator.java (The following screen shot shows the file structure). In the following sections, I will explain the functionality of each class.



```
CacheSimulator
  src
    (default package)
      Cache.java
      CacheEntry.java
      CacheSimulator.java
      Memory.java
      MyUtil.java
      SetBlock.java
      TraceGenerator.java
  JRE System Library [JavaSE-1.7]
```

2.1 Trace Generator

We need memory trace files as input of the simulator. Those files are in the Dinero format where each line in the file is a single memory reference. Here are few memory reference examples generated through TraceGenerator.java,

1 DA139ECB

2 2E6E517

0 EA45FA7B

The first digit describes the kind of reference, 0 for read, 1 for write, and 2 for instruction fetch. Because my simulator uses 32 bits memory address, the following characters like “DA139ECB” is the memory reference address in hexadecimal.

TraceGenerator.java now generates 3 trace files, i.e. 0.trace, 1.trace and 2.trace. Each trace file has 800,000 and 1,000,000 memory references by random number generation.

2.2 Cache Simulator

- CacheSimulator.java — is the place where main function gets called. There are three phases including configuration parameters collecting, cache initialization and trace file parsing, and displaying statistics.
- Memory.java — a simple memory structure. Main memory contains an array of data. Each data is the same value of its address.
- Cache.java — a key part for cache simulator. It stores all cache information like cache size, number of sets, tag fields, index fields, offset fields, and also all the global statistical variables. It's the entry point for write, read memory references.
- SetBlock.java — stands for a set of blocks. It's the place the simulator will perform LRU replacement policy.
- CacheEntry.java — tracks the valid, dirty and tag fields.
- MyUtil.java — contains all the self-defined converting function like hex to int, hex to long, hex to binary, binary to int, binary to long, binary to hex, int to hex, long to hex etc.

3 Output and Analysis

I experiment a lot for general purpose and also for specific question. The specific question I picked up is "How does cache size affect the number of hits and misses?" Section 3.1 shows the general purpose experiments with unified cache and split cache configurations for 0.trace file. There are lots more general purpose experiments for 1.trace and 2.trace, which I I kept all the output files in "generalResults" folder. It would be nice to have future feature to get user specified trace file name. Now you should change line 60 in CacheSimulator.java to switch between those trace files (0.trace, 1.trace, 2.trace).

3.1 General Purpose Output and Analysis

✦ Configuration: UorSC=U UBlockSize=16 USetSize=2 UNumBlocks=1024 UNT=5 MissPenalty=100

```
memorySize: 4294967295
--------- Configuration -----------
Unified Cache, Block size: 16bytes, Set size: 2blocks, Number of Blocks:
1024blocks, Hit time: 5.0ns, Miss penalty: 100.0ns
After calculation, cacheSize: 16384bytes, numSets: 512sets
tag: 19bits, index: 9bits, offset: 4bits
...... Parsing trace files ......
--------- End of trace files parsing -----------
--------- Displaying Statistics -----------
Total Memory References: 637683 Out of 955317
Average Memory Access Time: 100.0

Cache hits: 6 <<>> Cache misses: 637677
Hit rate total: 9.409e-06
Miss rate total: 1.000

Miss Read: 318065 & Miss Write: 319612
Miss Read Rate: 1.000 & Miss Write Rate: 1.000
Compulsary misses: 320126.0 & Conflict misses: 317554.0
Compulsary Miss Rate: 0.5020 & Conflict Miss Rate: 0.4980
```

✦ Analysis:

The above output for 0.trace. The first line shows the total memory size is 4294967295 bytes = 2^32 because the simulator uses 32 bit memory address all the time. The result printed divided by line separator to three parts which reflects the three phases.

During the first phase, the simulator reads in all the configuration parameters, then did calculation to get cache size = 16384 bytes = Block size * Number of blocks = 16 bytes * 1024 blocks; number of sets = 512 sets = Number of blocks / Set size = 1024 blocks / 2 blocks = 512 sets; then we can get tag, index, offset fields as well, offset = 4 bits = lg block size = lg 16 bytes, index = 9 bits = lg number of sets = lg 512 sets, then tag = 19 bits = 32 bits - index - offset.

Secondly, we get different memory references and action address by parsing the trace file (0.trace) specified in the program.

Third phase, the simulator calculates all the statistics then print out the result. Total Memory References are the total number of read and write memory references. “637683 Out of 955317” means for unified cache, for 0.trace, we got 637683 memory references. But 0.trace does have 955317 memory references since “2” is for instruction fetch which only happens in split cache scheme. Average Memory Access Time is 100 ns = Hit rate * Hit time + (1-Miss rate) * Miss penalty. Cache hits are 6 compared to cache misses are 637677, so the hit rate is really low while cache miss rate is high. I separated miss read and miss write and also calculated each rates. Finally it provides the compulsory misses and conflicts misses times and their rates.

✦ Configuration: UorSC=SC IBlockSize=64 ISetSize=1 INumBlocks=2048 DBlockSize=128 DSetSize=2 DNumBlock=16384 UNT=1 MissPenalty=120

```
memorySize: 4294967295
--------- Configuration -----------
Separate Cache, Instruction Block size: 64bytes, Instruction Set size: 
1blocks, Instruction Number of Blocks: 2048blocks
                         Data Block size: 128bytes, Data Set size: 2blocks, 
Data Number of Blocks: 16384blocks, Hit time: 1.0ns, Miss penalty in ns: 
120.0ns
After calculation, cacheSize: 2097152bytes, numSets: 8192sets
tag: 12bits, index: 13bits, offset: 7bits
After calculation, cacheSize: 131072bytes, numSets: 2048sets
tag: 15bits, index: 11bits, offset: 6bits
...... Parsing trace files ......
--------- End of trace files parsing -----------
--------- Displaying statistics -----------
Total Memory References: 955317 Out of 955317
Average Memory Access Time: 120.0
```

```
Cache hits: 266 <<>> Cache misses: 955051
Hit rate total: 0.0002784
Miss rate total: 0.9997

Miss Read: 635525 & Miss Write: 319526
Miss Read Rate: 0.9997 & Miss Write Rate: 0.9997
Compulsary misses: 329751.0 & Conflict misses: 625389.0
Compulsary Miss Rate: 0.3453 & Conflict Miss Rate: 0.6548
```

✦ Analysis:

The above output is still for 0.trace but the cache is split one. The first line shows the total memory size is 4294967295 bytes = 2^32 because the simulator uses 32 bit memory address all the time.

First phase, the simulator reads in all the configuration parameters, then did calculations to get 2 sets of cache size, number of sets, tag, index, offset fields as the previous unified cache for data cache and instruction cache.

Secondly, we get different memory references and action address by parsing the trace file (1.trace) specified in the program.

Third phase, as before, calculate all the statistics result and display it. Since Cache.java used static global variables for the statistics purpose, instruction and data caches shared the same pieces. Then the final statistics results reflects both caches memory references states.

More general purpose experiments results, please refer to folder “generalResults”.

3.2 Specific Experiment Output and Analysis

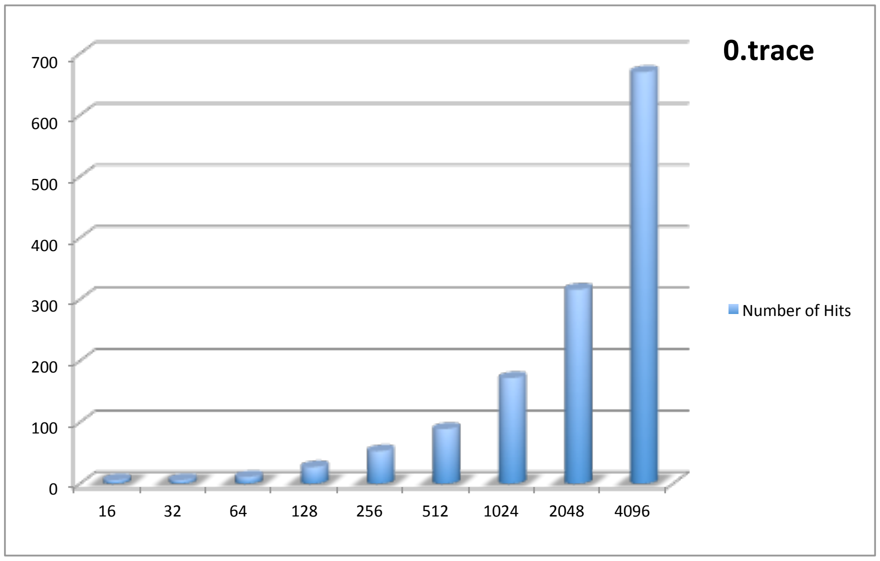
For the specific experiment, I want to see how the cache size affects the number of hits and misses.

The configuration is “UorSC=U UBlockSize=2^x USetSize=1 UNumBlocks=1024 UNT=5 MissPenalty=100”. I uses unified cache, associativity fixes at 1, number of blocks is always 1024 blocks. Only doubling block size, we can get gradually increased cache size. All the experiment output results are similar like those shown in section 3.1 before. For more detailed results, please check folder “SpecificResults”. Here, generalized tables and graphs are more clear for comparisons.

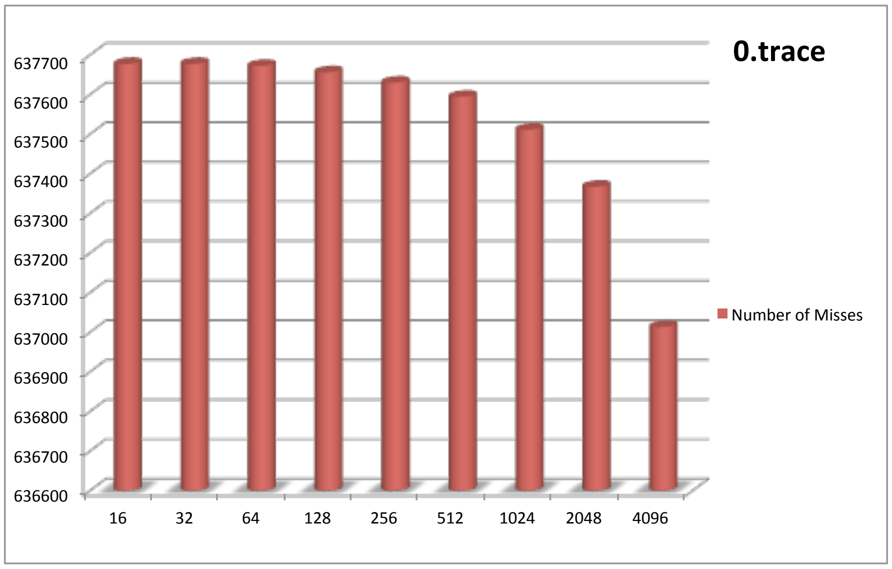
✦ 0.trace result:

Block Size (Bytes)	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
Number of Hits	6	6	11	27	53	89	172	316	670
Number of Misses	637677	637677	637672	637656	637630	637594	637511	637367	637013

0.trace Number of Hits Graph



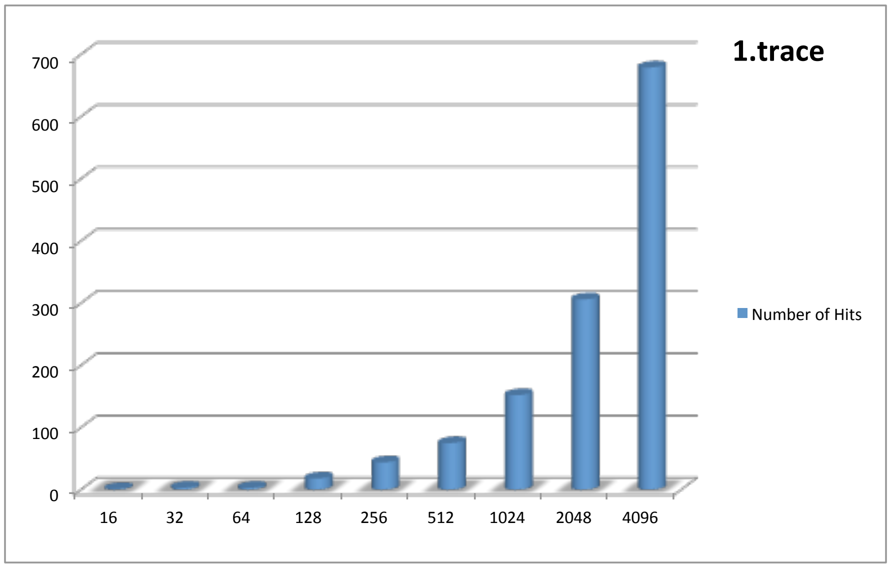
0.trace Number of Misses Graph



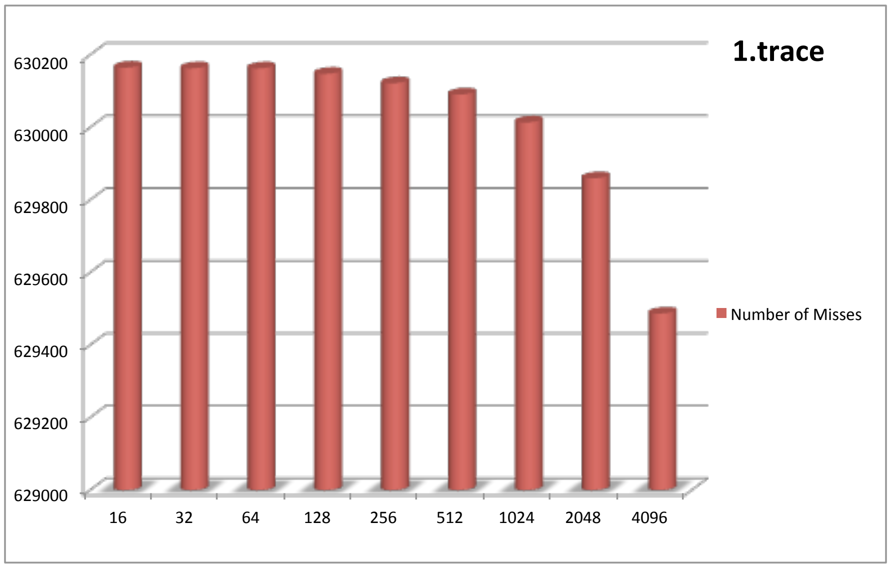
✦ 1.trace result:

Block Size (Bytes)	16	32	64	128	256	512	1024	2048	4096
Number of Hits	1	3	3	18	44	75	152	306	679
Number of Misses	630165	630163	630163	630148	630122	630091	630014	629860	629487

1.trace Number of Hits Graph



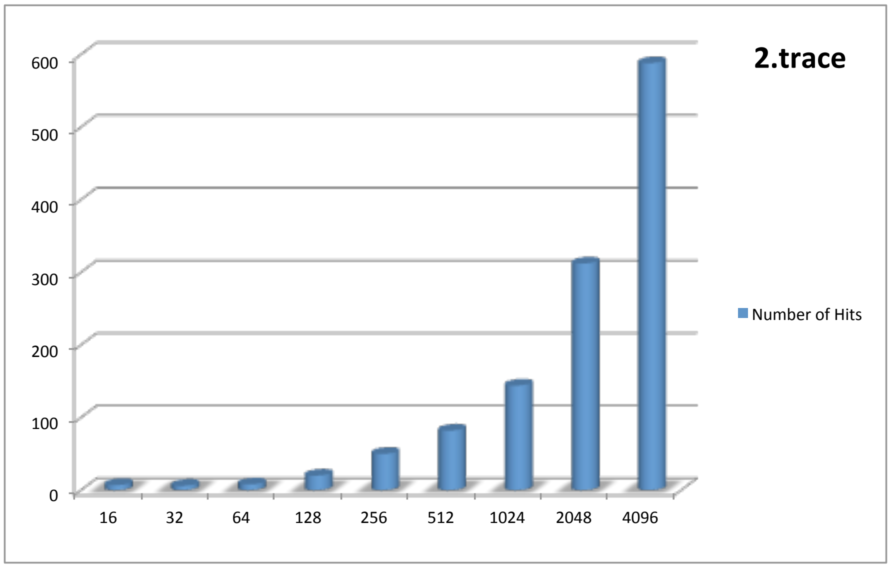
1.trace Number of Misses Graph



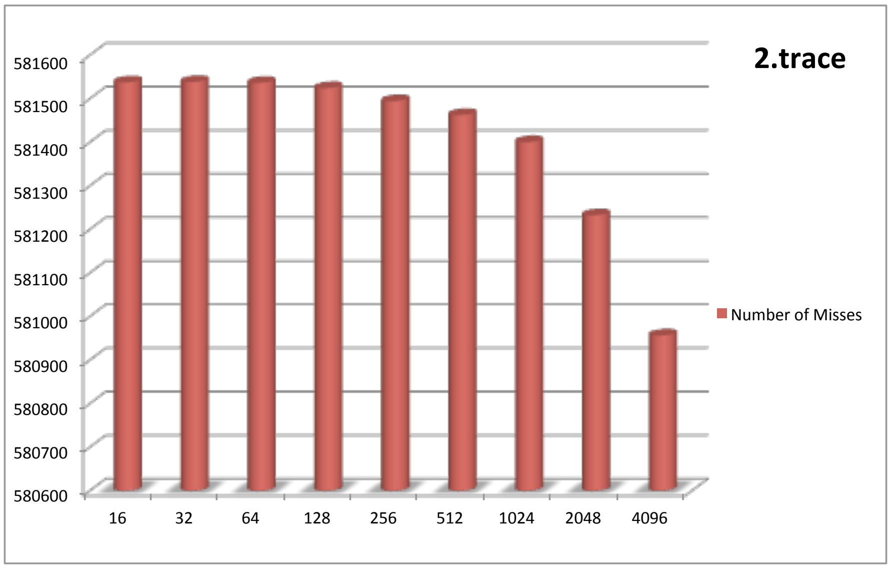
✦2.trace result:

Block Size (Bytes)	16	32	64	128	256	512	1024	2048	4096
Number of Hits	7	6	8	20	50	82	144	312	588
Number of Misses	581537	581538	581536	581524	581494	581462	581400	581232	580956

2.trace Number of Hits Graph

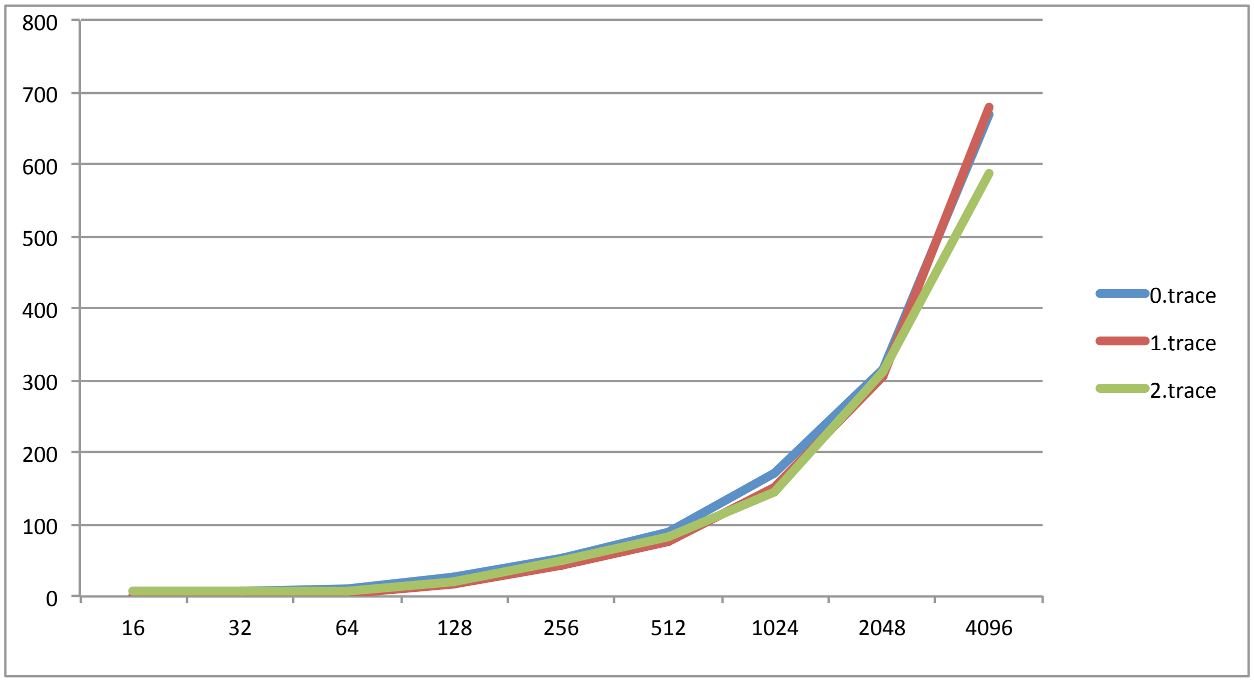


2.trace Number of Misses Graph



✦Merged Number of Hits Result and Graph:

Block Size (Bytes)	16	32	64	128	256	512	1024	2048	4096
0.trace	6	6	11	27	53	89	172	316	670
1.trace	1	3	3	18	44	75	152	306	679
2.trace	7	6	8	20	50	82	144	312	588



✦Analysis:

Since those memory references are totally random, the number of misses is much larger than the number of hits. I didn't mix number misses with hits together since misses value is too high compared to hits. If I put them together, we definitely can't see clearly the growth of the number of hits with the increasing cache size. For those three trace files with random memory references, they show the same trend that is a larger cache size, higher hit rate while lower miss rate. However, larger cache size means more time to initialize memory and cache for my simulator. Therefore, AMAT decreases.