Lab 4

Group13: fl94 Fengyi Li, nn75 Nan Ni, jz270 Jiaran Zhou

1. **Design Pattern**

Our cache simulator works as the following figure shows. It first read the config from the user, which include A, B, C and level of caches, etc. Then it initializes the caches based on the configurations. After that, we parse the address line by line from the din file, and it reads, writes or fetches from the cache based on the operation code. When the whole file is executed, we calculate the performance of our cache and print the results.

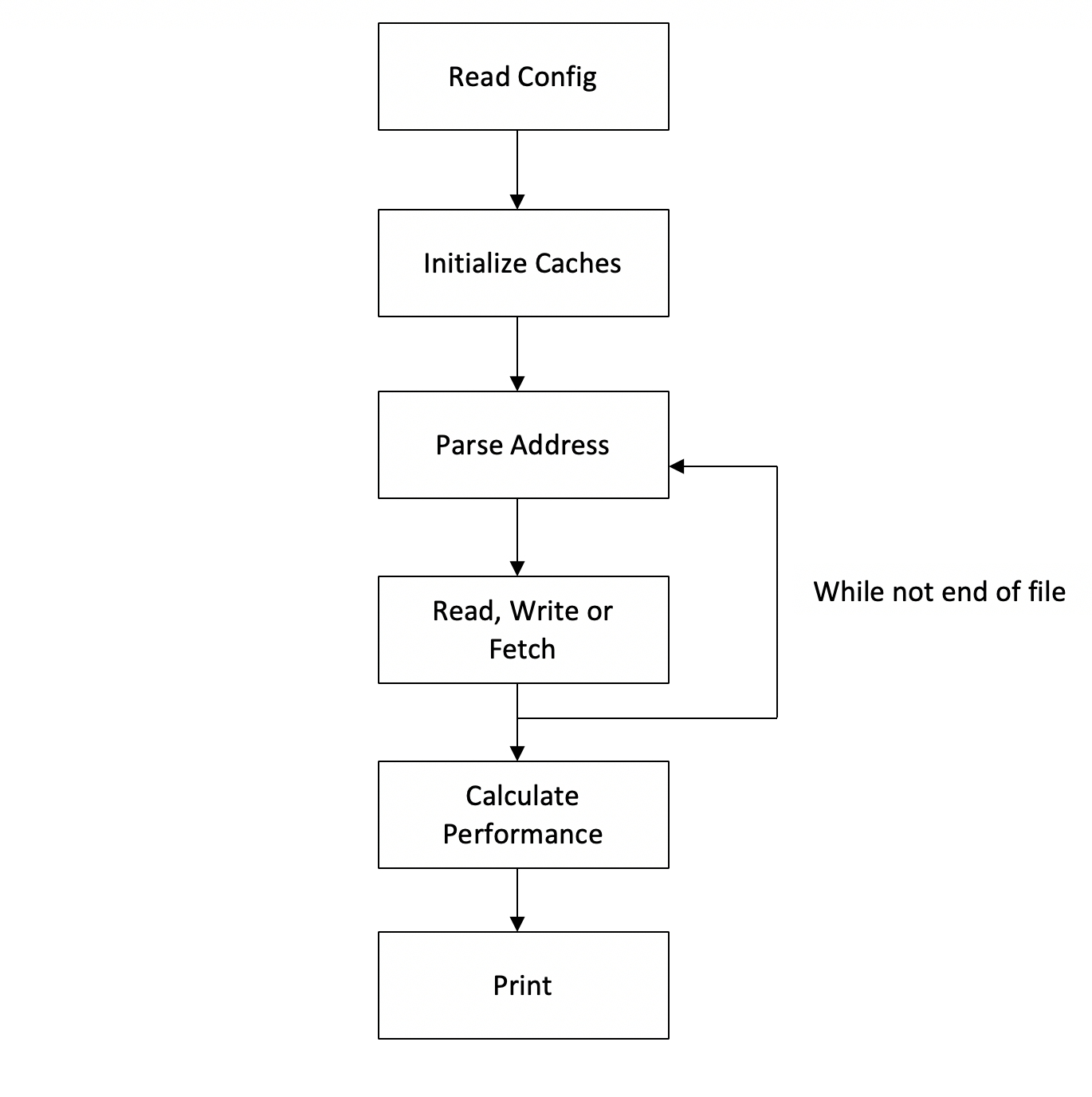


Fig 1. Design Pattern of cache simulator

1. **Implementation Details**
   1. **Classes**

We have a cache class, a block class and a cache\_simu class.

A block is a way in a real cache. It has the following members: a valid bit, a dirty bit, a tag and a lastaccesstime to indicate which way to cache next.

A cache has sets, and a set has several ways. So a cache is actually a vector of vector of blocks. It also has the following members: associativity, block\_size and capacity which indicate A, B and C, and the number of total read, write and fetch as well the number of hits, and the level which indicate if it is L1 or L2, etc.

A cache\_simu class is a cache simulator. It initializes several levels of caches based on the configuration, each level of cache is vector of caches. It has the following functions: a void parse which reads din file line by line and divides it into operation code and address. LRU\_index and random\_index calculate the next way to cache separately. And read, write and fetch base on the operation code. Finally, a print function prints the performance details in a chart.

* 1. **Read**

The read function increases the number of read at first. Then it calculates the tag, setID and offset. After that, it looks into the cache. If hit, it increases the number of hits, then set its access time to be newly accessed. If not hit, and if it is already dirty, it will first write back what’s in it to lower level caches, then it will look into lower level of caches to read. At last, it will increase the number of misses, set the flag, set dirty to false and set its access time to be newly accessed.

* 1. **Write**

The write function increases the number of read at first. Then it calculates the tag, setID and offset, just as what read does. After that, it looks into the cache. If hit, it will increase the number of hits, then set its access time to be newly accessed. And it will set dirty to true. If miss, and if it is write allocate, if the block is dirty, it will write back to L2 first, then it will set the tag, set dirty bit to true, increase the number of misses and set its access time to be newly accessed. If it is not write allocate, it will write to L2 directly.

* 1. **Fetch**

The fetch function works very similarly to read, except that when the cache is split, it fetches from I cache and read reads from D cache. When the cache is unified, it is exactly the same as read.

1. **Results**

We run our cache simulator on three different configurations.

1. Unified L1 and No L2:

L1 is direct mapped, block size of 32 bytes, capacity of 8192 bytes, DO NOT allocate on write miss.

2. Split L1 and Unified L2:

L1 I$ direct mapped, block size of 32, capacity of 8192, allocate on write miss.

L1 D$ 2-way set associative, block size of 32, capacity of 8192, allocate on write miss, RND replacement policy.

L2  8-way set associative, block size of 32, capacity of 32768, allocate on write miss, RND replacement policy.

3. Split L1 and Split L2:

L1 I$ direct mapped, block size of 64, capacity of 8192, allocate on write miss.

L1 D$ 4-way set associative, block size of 64, capacity of 16384, allocate on write miss, LRU replacement policy.

L2 I$ 8-way set associative, block size of 64, capacity of 32768, allocate on write miss, LRU replacement policy.

L2 D$ 16-way set associative, block size of 64, capacity of 65536, allocate on write miss, LRU replacement policy.

The result is as following figure.

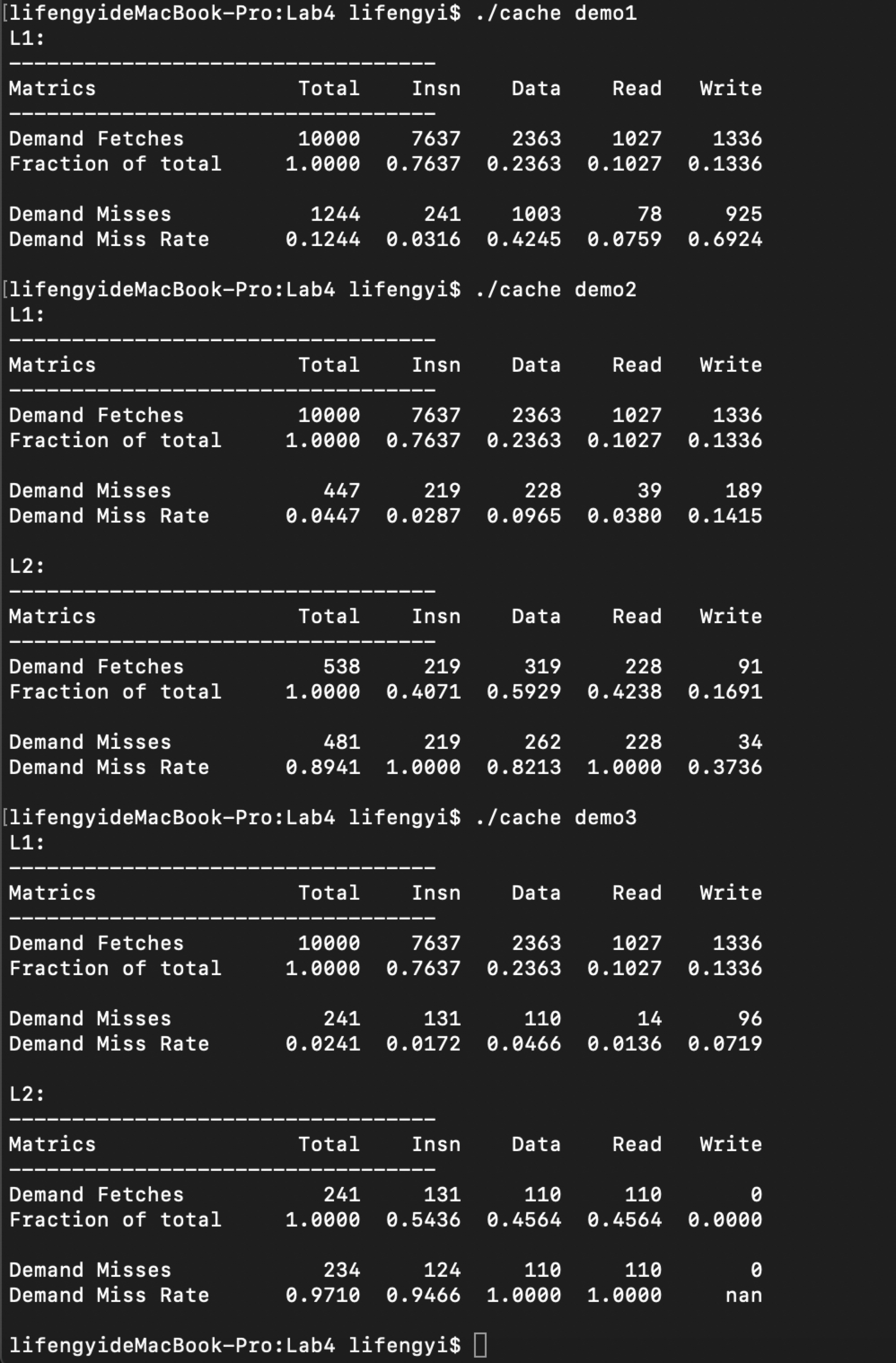


Fig 2. Cache Simulation Result