Assignment #4: Caches

CSI3102-02 (Architecture of Computers), Spring 2022

Welcome to the fourth programming assignment for CSI3102-02 @ Yonsei University for Spring 2022! Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system until the deadline. The deadline to submit your code is **23:59 KST on June 22nd, 2022 (Wed)**.

Caches

In this programming assignment, you will implement a cache.

Caches are the levels in the memory hierarchy located between the processor and the main memory. They provide faster data access times than the main memory, and thus are used to temporarily store the data of the main memory by exploiting spatial and temporal localities of the data.

A cache manages its data by splitting them into lines (also known as blocks). Each of the lines consists of a valid bit, a tag block, and a data block. The valid bit indicates whether the associated tag and data block contain useful data. The tag block stores the tag of the data stored in the associated data block. The tag is essentially some of the most significant bits of the data block's memory address. By comparing the tag stored in the tag block and that of a read/write access, we can identify whether the access is a hit. Of course, the corresponding valid bit must be set to 1 (i.e., valid) for the access to be a hit.

The lines get grouped into sets. The number of lines in a set depends on the type of the cache; a direct-mapped cache assigns only one line to a set, N-way set-associative caches group N lines as a set, and fully-associative caches assign all of their lines to a single set.

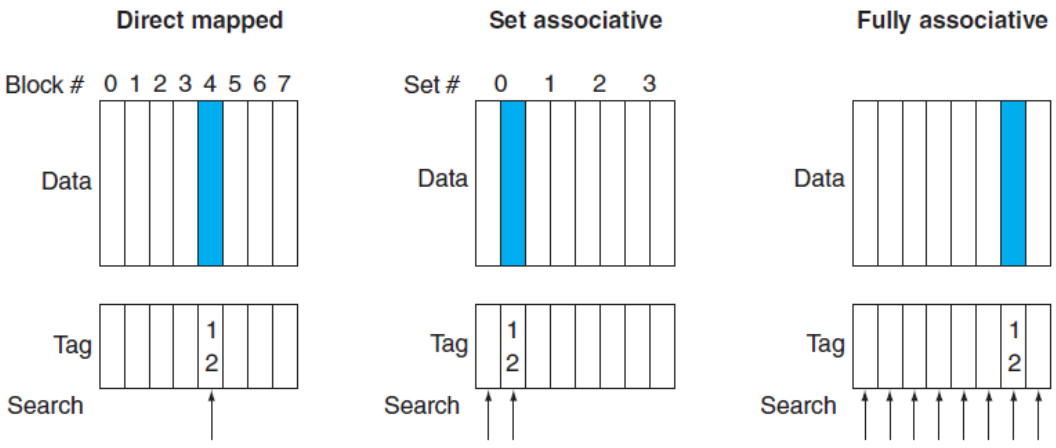

FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Given a read/write request, a cache first identifies the target set of the request. For doing so, the cache obtains the block address of the request by discarding some of the least-significant bits of the request's target memory address. The discarded least-significant bits are known as the byte offset of the request; after identifying the target cache line, the byte offset is used to specify which byte of the line's data block should be accessed for the request. Then, some of the least-significant bits of the block address are used

to identify the target set of the request; if the cache consists of 2^N sets, the N least-significant bits of the block address are used as index bits. In the figure shown above, the 3 and 2 least-significant bits of the block address will serve as the index bits for the direct-mapped and 2-way set-associative caches, respectively. The remaining (most-significant) bits of the block address form the tag of the request.

After identifying the target set of the request, the cache should examine the lines of the set to see if there is a valid line whose tag matches with that of the request. If such a line exists in the set, the request is a hit and the operation of the request gets performed on the line (i.e., read the requested data from the data block, write a new piece of data to the data block). However, if the request is identified as a miss, the cache should load the requested data from the main memory to one of the lines in the set. This is where the two types of cache policies kick in: the write policy and the replacement policy.

Such a series of operations of a cache equally applies to all the three types of caches (i.e., direct-mapped, set-associative, fully-associative). Therefore, in this assignment, you will implement a generalized cache which supports arbitrary numbers of lines, sets, and lines per set. The generalized implementation can simulate all the three types of caches by simply adjusting the aforementioned parameters. You can adjust the size of a cache by changing the number of lines. Then, you can simulate a direct-mapped cache by setting the number of sets to be equal to the number of lines, and thus the number of lines per set becomes one. Similarly, N-way set-associative caches can be mimicked by setting the number of sets as the number of lines divided by N. By setting the number of sets as one, you can emulate a fully-associative cache.

Two Write Policies of a Cache

The write policy of a cache controls the behavior of the cache when the cache receives a write request. There are two popular write policies: write-through and write-back. In this assignment, you will implement both the write-through and the write-back policies.

First, the write-through policy updates not only the data stored in a cache line, but also relays a write request to the main memory. The write-through policy ensures that both the cache and the main memory always have the latest data for a particular memory address; however, the policy can incur significant performance loss in real-world scenarios as accessing the main memory takes much longer than accessing the cache.

Second, the write-back policy only updates the data stored in a cache line without relaying a write request to the main memory. Since the latest data gets stored in the cache, an inconsistency between the cache and the main memory can occur. However, it is okay to have the inconsistency since the processor will always access the cache first, followed by the main memory upon a missed read/write request, due to the nature of a memory hierarchy.

To summarize, when a write request is a hit, a write-through cache should update both its line and the main memory, whereas a write-back cache only updates its line.

The Least Recently Used (LRU) Replacement Policy

When a read/write request misses in a cache, the data for the missed request should be fetched from the main memory to the cache. When there exists an invalid line in the target set of the cache, the cache can simply populate the invalid line with the requested data. Unfortunately, as the cache processes many read/write requests and has a limited number of lines, the lines of a set quickly become fully populated. In such a case, the cache must select a victim line among the lines of the target set and then evict the victim line to the main memory. Evicting the victim line creates one empty line for the target set, allowing the

(evicted) victim line to be used for serving the missed request. The victim line will then be populated with the data needed by the missed request.

Selecting a proper victim line among the lines of a target set is crucial to achieving high performance. In this assignment, we will implement a popular policy called the Least Recently Used (LRU) replacement policy. The LRU policy is designed to exploit temporal locality which states that a recently-used data is highly likely to be used again by the processor in the near future. Among the lines of the target set, the LRU policy selects the least recently used line as the victim. The LRU policy associates a line with a counter. When a line gets populated by loading a piece of data to the line, the counter for the line is set to 0. Then, the counter is incremented by one every time the line is not referenced but the other lines belonging to the same set get accessed. In this way, the cache can easily track the least recently used line within a set by searching for the line having the highest counter value.

There can be various possible implementations of the LRU replacement policies; however, in this assignment, you should implement the following working model:

1. Receive a read/write request
2. Increment all the counters of the lines in the target set of the read/write request by one
3. If the request hits in its target set,
    a. Reset the counter of the target line to zero.
4. If the request misses in its target set,
    a. If there exists one or more invalid lines in the target set, then select the invalid line with the smallest index as the victim line. Otherwise, select the valid line which has the highest counter value, and thus the least recently used line of the set, as the victim line.
    b. Replace the victim line with the data stored in the main memory for the missed request
    c. Set the (replaced) victim line's counter value to zero

Downloading the Assignment

Type the following commands in the terminal to download the assignment.

```
csi3102@csi3102:~$ wget https://hpcp.yonsei.ac.kr/files/csi3102/assn4.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn4.tar.gz
csi3102@csi3102:~$ cd ./assn4
csi3102@csi3102:~/assn4$ ls
Cache.cc  Cache.h  data.0  data.1  data.2  Makefile  outputs  testCache.cc
```

Among the files, you only need to modify `Cache.cc` for completing this assignment.

The Cache Class

Now, open and inspect the `Cache` class defined in `Cache.h`. The `Cache` class defines all the necessary members and methods you need for implementing a cache. Some important members and methods are:

- **`Cache(…)`**: The constructor for the `Cache` class. It takes as input the number of lines, the number of sets, the line size (in bytes), and the write policy of a cache. As enforced by the `assert` statements in the constructor, you may safely assume that the line size is always a multiple of words (i.e., 4 bytes). In addition, you can assume that both the number of lines and the number of sets are powers of two, and that the number of lines is a multiple of the number of sets.

- **`typedef struct {…} Line`**: This type defines the structure of a cache line. It consists of a valid bit (`bool valid`), a tag block (`WORD tag`), a data block (`BYTE *data`), and a counter for the LRU replacement policy (`size_t lruCounter`). The type also includes a dirty bit (`bool dirty`) which is only used for write-back caches. When a write-back cache is being emulated and the cache fills a line by loading data from the main memory, the dirty bit of the line should be set to `false` to indicate that the data block has not been modified. When the data block of the line is modified due to a hit write request, the dirty bit should be changed to `true` to denote that the data block has been updated after it had been loaded from the main memory.

- **`Line **_lines`**: The lines of a cache are organized as a two-dimensional array whose first index denotes the index of the set and second index denotes the index of the line within the set. For example, the second line of the third set of a cache can be accessed through `_lines[1][2]` since indices start from zero in C++.

- **`void print()`**: This method prints the current status of the cache including the values of each line's valid bit, tag block, data block, LRU counter, and dirty bit. You can use the method to debug your code. Analyzing the code of the method can help you better understand the roles and meanings of the various members in the `Cache` class.

Your Task: The `Cache::access` Method

As mentioned earlier, you will be implementing a cache in this assignment. Specifically, you will implement the `Cache::access` method declared in `Cache.h` and will be implemented by you in `Cache.cc`. You may assume that the method gets invoked for each read/write request (which should have been made and sent to the cache by the processor in real-world scenarios).

The method simulates the behaviors of a cache upon receiving a read/write request. To emulate the reception of the request by the cache, the method takes as input three arguments upon an invocation: the 32-bit target byte memory address (`WORD addr`), a boolean value indicating whether the current request is a read or a write (`bool isWrite`, true if write and false if read), and the pointer to a word-sized buffer which serves as an input-output data port (`WORD *data`). You may assume that the target memory address is word-aligned (i.e., it is always a multiple of 4 bytes) and the granularity of the access is a word (i.e., 4-byte read/write requests to mimic the processor's lw/sw instructions). When the current request is a read request, the cache should perform necessary operations (e.g., lookup, replacement) and return the data requested by the read request by writing it to `*data`. If the current request is a write request, the data to be written to the target memory address can be accessed by the cache by reading the value of `*data`.

Your implementation of the `Cache::access` method will be invoked by the `parseFile` function defined in `testCache.cc`. The `parseFile` function parses the provided data file which consists of three types of commands to the cache: memory initialization (`I`), read request (`R`), and write request (`W`). As an example, open and inspect the `data.0` file. You will see that each line of the file consists of three tokens. All the three types of commands consists of three tokens as shown below:

- **`I addr data`**: Initialize the four-byte memory starting from the byte address `addr` with the 4-byte data. Note that data needs to be an 8-digit, and thus 4-byte, hexadecimal value.

- **`R addr .`**: Send a read request to the cache whose starting address is `addr` and the size of the data to be fetched from the cache is 4 bytes. You should not omit the third token, `.`, for the `parseFile` function to operate correctly.

- **`W addr data`**: Send a write request to the cache whose starting address is `addr` and the four-byte `data` to be written from the starting address. Similar to the memory initialization commands, the data needs to be an 8-digit hexadecimal value.

Where is the Main Memory?

By reading this document up to this point, you might wonder how the main memory to be used by the cache is defined in where. For simplicity, the main memory is defined as an array of bytes in `testCache.cc` through `BYTE _memory[MEMSIZE]` statement. Given that the cache needs to interact with the main memory to fetch the data to serve a missed request, your implementation of the `Cache::access` method should directly modify the contents of the `_memory` variable. Your implementation can access the main memory thanks to the `extern BYTE _memory[MEMSIZE]` statement located at the top of `Cache.cc`.

One thing to remember is that both the main memory and the cache employ little-endian. You can infer this by analyzing the `Cache::print` method and the `parseFile` function.

Examples

We provide three example data files you can use to test your `Cache::access` method. After implementing the method, you can type the following commands to the command-line and examine the output to validate the functional correctness of your implementation:

```
csi3102@csi3102:~/assn3$ make clean && make && ./testCache 8 4 4 4 WriteBack ./data.0
...
...  <-- your output for a write-back cache having
...      8 lines, 4 sets, line size of 4 bytes with data.0.
...  <-- compare your output against ./outputs/output-8-4-4-WriteBack-data.0!
...
```

Notice that the generated `testCache` binary takes as input six arguments: the number of lines of a cache, the number of sets of the cache, the line size of the cache, the write policy of the cache, and the path to the data file to be used for evaluating the cache. You may assume that the line size of the cache is always a multiple of 4 bytes.

As the outputs are too long to be included in this document, we provide you sample outputs with some representative input arguments to the testCache binary. See the files in the outputs directory. The names of the files follow the convention of `output-numLines-numSets-lineSize-writePolicy-dataFile`. Please compare your implementation's outputs against the files in the output directory and their corresponding command-line arguments.

Submitting Your Code

After completing your assignment, you should submit your **`Cache.cc`** to the LearnUs system. The deadline to submit your code is **23:59 KST on June 22nd, 2022 (Wed)**. **No late submissions will be accepted**.

In addition, the TAs will not confirm whether you have successfully submitted your code to the LearnUs system. It is your responsibility to check and validate whether your submission is successfully submitted.