EEE3530-01 **Spring 2024**

Assignment 5: Cache

Due: Sunday, June 23, 2024, 11:59 PM

1 Introduction

- The objective of this assignment is to annex a small buffer, called *victim cache*, to the data cache. This assignment requires working on the Kite C++ code.
- To begin the assignment, go to the `kite/` directory that you worked on for previous assignments.
- Download `cache.sh`, and execute it inside the `kite/` directory to update Kite.
- In this assignment, `Makefile` uses `-DBR_PRED` and `-DDATA_FWD` options to enable branch prediction and data forwarding in the processor pipeline.

```
$ cd kite/
$ wget https://casl.yonsei.ac.kr/teaching/eee3530/cache.sh
$ chmod +x cache.sh
$ ./cache.sh
$ make
g++ -g -Wall -O3 -DBR_PRED -DDATA_FWD  -o alu.o -c alu.cc
g++ -g -Wall -O3 -DBR_PRED -DDATA_FWD  -o br_predictor.o -c br_predictor.cc

...
```

- The following describes how Kite handles cache operations in `data_cache.cc`.
- **Cache initialization:**
  - The constructor function of the `data_cache_t` class creates a cache array (i.e., `blocks[]`) and calculates several parameters needed later to find the set index and tag of a memory address during cache access.
  - The skeleton code creates a 512-byte direct-mapped cache with a 32-byte line size.
  - For the 2^5-byte block size, `block_offset` is 5 bits, and `block_mask` is `0b11111` (i.e., five 1s).
  - The direct-mapped cache has 16 sets, which is calculated as `num_sets = cache_size / block_size`.
  - For 2^4 sets and 2^5-byte block size, `set_offset` is 9 bits, and `set_mask` is `0b111100000` (i.e., four 1s followed by five 0s), which will be used to extract a set index from a memory address.
- **Reading or writing a cache block:**
  - The `read()` or `write()` function of the data cache is called when the processor encounters a `ld` or `sd` instruction in the memory stage of the pipeline.
  - The `read()` function first calculates the `set_index` and `tag` of a memory address.
  - In the direct-mapped cache, `blocks[set_index][0]` is the only location that the cache block can go to. The way number is hard-coded as `[0]` for direct mapping.
  - If the cache line is valid and has a matching tag, it is a cache hit.
  - On a cache hit, the current clock cycle (i.e., `ticks`) is recorded as the last access time, but this information is not used in this assignment since there is no need for LRU replacement in the direct-mapped cache.
  - The memory instruction's `rd` value is read via doubleword-granular addressing (i.e., `block->data + ((addr & block_mask) >> 3)`) within the 32-byte cache block.
  - On a cache miss, the base address of the missed cache block (i.e., `addr & ~block_mask`) and the block size (i.e., the amount of data to fetch from memory) are sent to the memory.

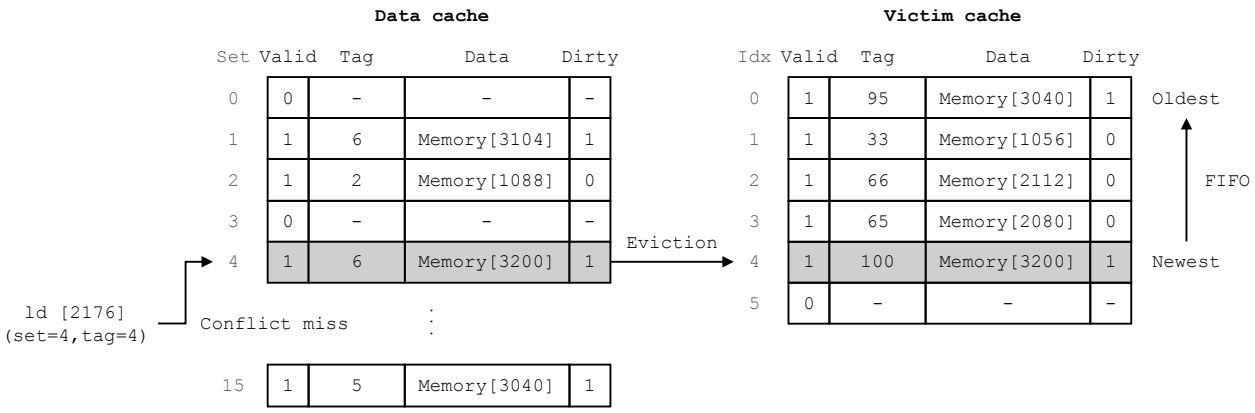

Fig. 1. On cache eviction, a victim entry is placed in the victim cache. In a later cache access, if a missed cache block is found in the victim cache, it is brought back to the data cache. When the victim cache becomes full, the oldest cache is evicted in the FIFO order.

  - The `write()` function follows the same procedure as `read()` except that i) it marks a cache block dirty and ii) writes the `rs2` value of a memory instruction to a doubleword-granular location in the cache block.

- **Handling a cache miss:**
  - A memory request sent out by `read()` or `write()` on a cache miss returns via `handle_response()` after memory access latency, which is set to 50 cycles in this assignment.
  - On the return of the missed memory block, its set index and tag are calculated to determine where it should be placed in the data cache.
  - If the located cache entry is valid and dirty, it is written back to the memory; the default code assumes a writeback cache. Otherwise, a clean cache entry can be simply overwritten.
  - Since cache blocks in Kite are directly tied to memory data using C++ pointers (i.e., `data` in the `block_t` class), Kite does not model writeback operations but simply traces the number of writebacks. Such a code design makes Kite look like having an infinite-sized write buffer (or no write buffer stalls).
  - After replacing the cache entry with the fetched memory block, the cache access is replayed by calling `read()` or `write()`, which must hit in the cache then.
- **Others:**
  - The `run()` function of the `data_cache_t` class works simply as a timer to call `handle_response()` at proper timing after memory access latency.
  - Lastly, `print_stats()` displays cache stats at the end of a simulation.
- Your work in this assignment is i) to fill in the functions of the new `victim_cache_t` class, and ii) call its functions in the `read()`, `write()`, and `handle_response()` functions of the data cache.

2 Implementation

- This section describes the operations of *victim cache* you will have to implement for this assignment.
- Suppose the 512-byte direct-mapped cache with 32-byte line size has cache contents shown in Fig. 1.
- The processor executes a `ld` instruction with the memory address of 2176 (or `···0000 1000 1000 0000` in binary digits).
- The lowest 5 bits of the memory address are block offset, and the next 4 bits are used for cache indexing. The remaining upper bits are used as a tag. Thus, the memory address of 2176 has a set index of 4 and a tag of 4, as shown in the figure.

- However, set #4 of the data cache is occupied by another cache block, resulting in a cache miss.
- The cache entry is replaced with the new block of memory address 2176, and the previous cache block of memory address 3200 is evicted from the data cache.
- **Cache eviction and victim cache:**
  - Instead of exploiting set associativity in the data cache, a small buffer, called *victim cache*, is annexed to the data cache as shown in Fig. 1.
  - The victim cache is a fully associative buffer that stores evicted cache blocks to give them a second chance. The skeleton code creates a 6-entry victim cache.
  - In Fig. 1, the evicted cache block of memory address 3200 is placed at the end of the buffer. Cache blocks in the victim cache are arranged in the first-in, first-out (FIFO) manner.
  - Since the victim cache is fully-associative not direct-mapped, the tag of the evicted cache block (or simply victim block) needs to be updated accordingly.
  - The victim block's tag can be calculated from its base address, such as `addr >> block_offset`, or retrieved from its tag and set index in the data cache, such as `(tag << (set_offset - block_offset)) | set_index`.
- **Victim cache hit:**
  - During a cache access in `read()` or `write()`, it looks up both data cache and victim cache. If one of them hits, it is regarded as a cache hit.
  - If the sought cache block is found in the victim cache (i.e., victim cache hit), it is moved back to the data cache and removed from the victim cache.
  - To maintain the FIFO sequence of the queue, subsequent cache blocks in the victim cache need to be shifted to fill the gap after the hit block leaves the victim cache.
  - Moving the hit block from the victim cache to the data cache should evict a cache block at the same entry in the data cache and place it at the end of the victim cache's FIFO queue.
  - For instance, if the cache block of memory address 2080 hits in the victim cache in Fig. 1, it is moved to set #1 in the data cache, and the cache block of memory address 3104 is evicted from the data cache and placed in the victim cache.
  - The cache block of memory address 3200 in the victim cache is shifted to index #3, and index #4 is filled with the newly evicted cache block of memory address 3104, whose tag must be updated to 97.
- **Victim cache eviction:**
  - A cache block is removed from the victim cache i) when the cache block is hit and moved back to the data cache or ii) when the victim cache has no more free entries to store an evicted block from the data cache.
  - The first case is already described as a part of the victim cache hit procedure.
  - In the second case, the oldest cache block in the victim cache is removed. Since the victim cache maintains cache blocks in the FIFO sequence, index #0 is always the oldest block.
  - If the cache block to be evicted from the victim cache is dirty, such as the cache block of memory address 3040 in Fig. 1, it must be written back to the memory.
  - With the victim cache, dirty blocks are written back to the memory when they are evicted from the victim cache, not from the data cache.
- The following shows the definition of the `victim_cache_t` class in `data_cache.h`.

```
// Victim cache
class victim_cache_t {
public:
    victim_cache_t(uint64_t m_size);
    ~victim_cache_t();
```

```
    block_t remove(uint64_t m_addr);            // Remove the matched block.
    void insert(block_t m_block);               // Insert a victim block.
    void print_stats();                         // Print victim cache stats.

private:
    uint64_t num_entries;                       // Number of valid entries
    uint64_t size;                              // Victim cache size
    uint64_t num_accesses;                      // Number of accesses
    uint64_t num_hits;                          // Number of victim cache hits
    uint64_t num_writebacks;                    // Number of writebacks
    block_t *blocks;                            // Victim cache entries
};
```

- The constructor and destructor of the class, `victim_cache_t()` and `~victim_cache_t()`, are called at the beginning and the end of a simulation to create and destroy the victim cache, respectively. These functions are complete in `data_cache.cc`, so you may not need to change them.
- The `remove()` function removes the cache block of a matching address in the victim cache and returns it. If no matching entry is found, an invalid cache block (i.e., `valid = 0`) is returned.
- The `insert()` function adds a new cache block to the victim cache. If the victim cache is full, it should evict the oldest block to create a room for the new victim block.
- The `print_stats()` function displays the simulation results of the victim cache. Do not modify this function in `data_cache.cc`.
- `num_entries` indicates the number of cache blocks stored in the victim cache, and `size` is the total number of entries in the victim cache.
- `num_accesses` traces how many times the victim cache has been referenced to find missing blocks. Since looking up the victim cache is meaningless if the data cache is hit, `num_accesses` should not increment when the data cache is hit.
- `num_hits` counts how many times missing cache blocks are found in the victim cache.
- `num_writebacks` counts how many times dirty cache blocks are written back to the memory. This counter increments only when the oldest cache block is evicted from the victim cache because it is full.
- Lastly, `*blocks` represents a cache block array.

3 Submission

- After implementing the victim cache and its operations in the data cache, run the `program_code` in Kite.
- The provided `program_code` implements simple matrix multiplication following an output-stationary dataflow.
- Table 1 compares the results of three cases: i) a 1024-byte direct-mapped cache without a victim cache, ii) a 1024-byte 4-way set-associative cache without a victim cache, and iii) a 512-byte direct-mapped cache with a 6-entry victim cache.
- You only need to implement the last case (i.e., 512-byte direct-mapped cache with 6-entry victim cache), and your assignment will be graded based on its results.
- When the assignment is done, execute the `tar.sh` script in the `kite/` directory. It creates a tar file named after your student ID, e.g., `2024143530.tar`. Upload the tar file on LearnUs. Do not rename the file.

```
$ ./tar.sh
$ ls *.tar
2024143530.tar
```

Table 1. Performance comparison between i) a 1024-byte direct-mapped cache, ii) a 1024-byte 4-way set-associative cache, and iii) a 512-byte direct-mapped cache with a 6-entry victim cache.

```
/* Case I:
   1KB direct-mapped cache */


[Kite Pipeline Stats]
...
Cycles per instruction = 2.774
...

Data cache stats:
    Number of loads = 360
    Number of stores = 40
    Number of writebacks = 14
    Miss rate = 0.215 (86/400)
...
```

```
/* Case II:
   1KB 4-way set-assoc cache */


[Kite Pipeline Stats]
...
Cycles per instruction = 1.685
...

Data cache stats:
    Number of loads = 360
    Number of stores = 40
    Number of writebacks = 0
    Miss rate = 0.058 (23/400)
...
```

```
/* Case III:
   0.5KB direct-mapped cache
   with 6-entry victim cache */

[Kite Pipeline Stats]
...
Cycles per instruction = 1.685
...

Data cache stats:
    Number of loads = 360
    Number of stores = 40
    Miss rate = 0.058 (23/400)

Victim cache stats:
    Number of writebacks = 5
    Hit rate = 0.733 (63/86)
...
```

4 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

  -5 points: The tar file includes redundant tags such as a student name, `hw5`, etc.

  -5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

  -5 points each: `Case III` in Table 1 shows six lines of results, `Cycles per instruction`, `Number of loads`, `Number of stores`, `Miss rate`, `Number of writebacks`, and `Hit rate`. Each incorrect result line loses 5 points. Other stats will be a part of grading.

  -30 points: No or late submission.

  Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

  Final grade = F: The code is tweaked to deceive a grader as if it is a working implementation. Such an attempt is considered unethical and will be seriously penalized.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If you and the TA do not reach an agreement, elevate the case to the instructor for review of your assignment. Refer to the course website for the contact information of the TA and instructor: `https://casl.yonsei.ac.kr/eee3530`

- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.