Lab 7: Memory Hierarchy

Assigned: Wed., 4/1; Due: **Fri., 4/17** (Midnight)

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

Important Notes

Submission. Please submit the lab by executing the following command. You may be penalized if your handin directory does not have the correct layout.

```
$ cp -r src /afs/ece/class/ece447/handin/lab7/andrewID
```

Please submit the extra credit for this lab by executing the following commands.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab7extra/andrewID
$ cp -r src /afs/ece/class/ece447/handin/lab7extra/andrewID
```

Late Submissions. Please submit all late submissions directly to the handin directory (`lab7` or `lab7extra`), without tarring them up. We will *not* write-lock the handin directories. Instead, we will monitor the contents of your handin directory for new/modified files to determine the number of late days you consume, if any. We will *not* accept email submissions.

Lab Section Checkoff. Please come to any of the lab sections *before* Fri., 4/24.

1. Introduction

In this lab, you will extend Lab 6 to implement a memory hierarchy that includes an L2 cache and a DRAM-based main memory. Similar to Lab 6, we will fully specify the cycle-by-cycle behavior of the L2 cache and DRAM.

As we have discussed in class, while the L1 caches are typically tightly integrated with the pipeline, the L2 cache is more decoupled. In this lab, we will maintain the semantics for the L1 caches that were described in Lab 6, but replace the constant L1 cache miss latency of 50 cycles with a variable latency, which involves at least the L2 cache and may involve DRAM. Whenever the processor would have begun a 50 cycle L1 cache miss access stall in Lab 6, it instead issues a request to the memory hierarchy. When the data is available from the L2 or main memory (as specified below), the memory hierarchy issues a *cache fill* notification back to the L1 caches. If the L1 cache is still stalling on the missing data (i.e., no cancellation occurred), the data is inserted into the cache and the stall ends on the **next cycle**. We now specify exactly how the memory hierarchy is structured and how requests are handled in more detail.

2. Microarchitectural Specifications

2.1. Unified L2 Cache

The unified L2 cache is accessed whenever there is miss in either of the two L1 caches (instruction or data).

Organization. It is a **16-way** set-associative cache that is **256 KB** in size with **32 byte** blocks (this implies that the cache has **512 sets**). When accessing the cache, the set index is equal to [13:5] of the address.

MSHRs. The L2 cache has **16** MSHRs (*miss-status holding registers*), which is more than enough for our simple in-order pipeline. For every L2 cache miss, the L2 cache allocates an MSHR that keeps track of the miss. An MSHR consists of three fields: *(i)* valid bit, *(ii)* address of the cache block that triggered the miss, and *(iii)* done bit. When main memory serves an L2 cache miss, it sends a fill notification to the L2 cache that sets the done bit in the corresponding MSHR.

Accessing the L2 Cache. Assume that the L2 cache has a large number of ports and that accesses to it are never serialized. When a memory access misses in either of the L1 caches, the L2 cache is "probed" immediately in the *same* cycle as the L1 cache miss.[1] Depending on whether the access hits or misses in the L2 cache, it is handled differently.

- **L2 Cache Hit.** After **15 cycles**, the L2 cache sends a fill notification to the L1 cache (instruction or data). For example, if an access misses in an L1 cache but hits in the L2 cache at the 0th cycle, then the L1 cache receives a fill notification at the 15th cycle.
- **L2 Cache Miss.** Immediately, the L2 cache allocates an MSHR for the miss. After **5 cycles**, the L2 cache sends a memory request to main memory (this models the latency between the L2 cache and the memory controller). When the memory request is served by main memory, the L2 cache receives a fill notification. After **5 cycles**, the retrieved cache block is inserted into the L2 cache and the corresponding MSHR is freed (this models the latency between the memory controller and the L2 cache). In the **same cycle**, if the pipeline is stalled because of the cache block, then the L2 cache sends a fill notification to the L1 cache. Also in the **same cycle**, the cache block is inserted into the L1 cache. In the **next cycle**, the pipeline becomes unstalled.

Management Policies. A new cache blocks is inserted into the L2 cache at the MRU position. For a L2 cache hit, a cache block is promoted to the MRU position. (In the same cycle, there can be as many as two L2 cache hits: one from the fetch stage and another from the memory stage. If they happen to hit in two different cache blocks that belong to the same set, then there is an ambiguity about which block is promoted to the MRU position. In this case, we assume that the memory stage accesses the L2 cache before the fetch stage – we promote the fetched block to the MRU position and the loaded/stored block to the MRU-1 position. The L2 cache adopts true LRU replacement.

Other.

- Assume that the L1 and L2 caches are initially empty.
- Assume that the program that runs on the processor **never** modifies its own code (referred to as self-modifying code): a given cache block **cannot** reside in both the L1 caches.
- Unlike an L1 cache access, an L2 cache access **cannot** be canceled once it has been initiated.
- Similar to Lab 6, we do **not** model dirty evictions that can occur from either the L1 data cache (to the L2 cache) or the L2 cache (to main memory). When a newly inserted cache block evicts a dirty cache block, we assume that the dirty cache block is written into the immediately lower level of the memory hierarchy *instantaneously.* Furthermore, this happens **without causing any side effects**: in the cache the LRU ordering of the cache blocks is not affected; in DRAM, buses/banks are not utilized and the row-buffer status does not change.
- When the L2 cache does not have any free MSHRs remaining (which should not happen in our simple in-order pipeline), then an L1 cache miss **cannot** even probe the L2 cache. In other words, a free MSHR is a prerequisite for an access to the L2 cache, regardless of whether the access would actually hit or miss in the L2 cache. A freed MSHR can be re-allocated to another L2 cache miss in the **same cycle**.

[1] Probing is when the tags are searched to determine the hit/miss status of an access.

2.2. Main Memory (DRAM)

Organization. It consists of a **single rank** on **single channel**, where a channel consists of the command/address/data buses. The rank has **eight banks**. Each bank consists of **64K rows**, where each row is **8KB** in size. When accessing main memory, the bank index is equal to `[7:5]` of the address and the row index is equal to `[31:16]` of the address. For example, the 32 byte cache block at address `0x00000000` is stored in the 0th row of the 0th bank. As another example, the 32 byte cache block at address `0x00000020` is stored in the 0th row of the 1st bank.

Memory Controller. The memory controller holds the memory requests received from the L2 cache in a request queue. The request queue can hold an infinite number of memory requests. In each cycle, the memory controller scans the request queue (including the memory requests that arrived during the current cycle) to find the requests that are "schedulable" (to be defined later). If there is only one schedulable request, the memory controller initiates the DRAM access for that request. If there are multiple schedulable requests, the memory controller prioritizes the requests that are "row-buffer hits" (to be defined later). If there are multiple row-buffer hitting requests, as a final tie-breaker, the memory controller selects the memory request that arrived the earliest. If two requests arrived equally the earliest, as the final-final tie-breaker, the request from the memory stage takes precedence over the request from the fetch stage.

DRAM Commands & Timing. A DRAM access consists of a sequence of commands issued to a DRAM bank. There are four DRAM commands as we discussed in class: `ACTIVATE` (issued with the bank/row addresses), `READ/WRITE` (issued with the column address), and `PRECHARGE` (issued with the bank address). All commands utilize both the command/address buses for **4 cycles**. Once a DRAM bank receives a command, it becomes "busy" for **100 cycles**. 100 cycles after receiving a `READ` or `WRITE` command, the bank is ready to send/receive a 32 byte chunk of data over the data bus – this transfer utilizes the data bus for **50 cycles**.

Row-Buffer Status. Each DRAM bank has a row-buffer. Depending on the status of the bank's row-buffer, a different sequence of commands is required to serve a memory request to the bank. For example, let us assume that a memory request wants to access the 0th row in the 0th bank. At this point, there are three possible scenarios. First, *closed row-buffer* is when there is no row loaded in the row-buffer. Second, *row-buffer hit* is when the 0th row is already loaded in the row-buffer. Third, *row-buffer miss* is when a row other than the 0th row is loaded in the row-buffer. The following table summarizes the sequence of commands that is required for each of the three scenarios.

Scenario	Commands (Latency)
Closed Row-Buffer	`ACTIVATE`, `READ/WRITE`
Row-Buffer Hit	`READ/WRITE`
Row-Buffer Miss	`PRECHARGE`, `ACTIVATE`, `READ/WRITE`

"Schedulable" Request. A memory request is defined to be schedulable when all of its commands can be issued without any conflict on the command/address/data buses, as well as the bank. For example, at the 0th cycle, a request to a bank with a closed row-buffer is schedulable if and only if **all** of the following conditions are met. First, the command/address buses are free at cycles 0, 1, 2, 3, 100, 101, 102, 103. Second, the data bus is free during cycles 200–249. Third, the bank is free during cycles 0–99 and 100–199.

Other.

- Assume that the row-buffers of all the banks are initially closed.

3. Lab Resources

We provide no additional lab resources except for the reference simulator (`refsim`) that is available at: `/afs/ece/class/ece447/labs/lab7`. Please reuse the lab resources from Lab 6.

4. Extra Credit: Prefetching

We will offer up to 20% extra credit *for this lab* (equivalent to 1% additional credit for the course) for implementing *prefetching* from main memory into the L2 cache. Only the "top" submissions (judged by IPC on several benchmarks) will receive extra credit. Furthermore, a fully correct implementation of Lab 7 is a prerequisite for extra credit.

The prefetcher should create new requests at the L2 cache that are inserted into the MSHRs (in the same way that L1 misses create requests) for cache blocks that it predicts will be needed in the future. The prefetcher is allowed to observe only the addresses of accesses that come to the L2 from the L1 caches, and whether each access hits or misses in the L2 cache. For this extra credit, you are free to implement any previously-proposed prefetcher or a custom-designed prefetcher. We recommend that you start with a simple stream prefetcher [1, 2].

Because we are now using a C-based timing simulator, it is possible to "cheat" in your results if you are not careful (or if you are not honest!). At one extreme, the simulator could actually execute ahead in the program and find out which cache misses will occur, then prefetch exactly those cache blocks, all in in one cycle. Real hardware would not be able to do this. In general, it is possible to implement very complex algorithms in software that might not be possible to efficiently build in hardware (within a reasonable cycle time). For this extra credit, you should implement an algorithm that can be reasonably implemented with a small amount of storage and simple calculations that can be done in one or a few cycles. For example, a stride prefetcher can be built by tracking the last several cache miss addresses and using adders and comparators to determine whether a stride (regular distance between miss addresses) exists. The TAs and the instructor will be the final judges of whether a prefetcher design is acceptable. If there is any doubt, you should talk to the TAs about your extra-credit prefetcher design before the lab deadline (this is encouraged!).

In addition to your source code, please submit a readme (`README.txt`) that briefly summarizes your prefetcher design and the performance improvements that it achieves.

References

[1] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

[2] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, 2007.