Lab 4: Memory Hierarchy

Assigned: Mon., 19.11; Due: **Mon., 3.12** (Midnight)

Instructor: Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Jeremie Kim, Juan Gomez Luna, Geraldo Francisco de Oliveira, Minesh Patel, Giray Yaglikci

1. Introduction

In this lab, you will extend Lab 3 to implement a memory hierarchy that includes an L2 cache and a DRAM-based main memory. Similar to Lab 3, 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 tightly integrated with the pipeline, the L2 cache is more decoupled from the pipeline. In this lab, we will maintain the semantics for the L1 caches that were described in Lab 3, 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 3, it instead issues a request to the memory hierarchy in this lab. 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 in the **next cycle**. We now specify exactly how the memory hierarchy is structured and how requests are handled in more detail.

2. Your Task: Extending the Simulator with L2 Cache and DRAM

2.1. Unified L2 Cache

The unified L2 cache is accessed whenever there is a 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

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

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 block is inserted into the L2 cache at the MRU position. For an 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 block that is requested by the fetch stage to the MRU position and the block that is requested by the memory stage 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 3, 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**: the LRU ordering of the cache blocks in the corresponding cache set 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**.

2.2. Main Memory (DRAM)

Organization. The DRAM system has a **single rank** on a **single channel**, where the 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 described 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 in the following order, which is similar to the FR-FCFS policy:

1. requests that are row buffer hits are prioritized over others
2. requests that arrived earlier are prioritized over others

3. requests coming from the memory stage are prioritized over others

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). Each command utilizes both the command and address buses for **4 cycles**. Once a DRAM bank receives a command, it becomes "busy" for **100 cycles** and cannot accept any other commands. 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 needs to access the 0th row in the 0th bank. At this point, there are three possible scenarios:

- Row buffer hit: when the 0th row is already in the row buffer
- Row buffer miss: when there is no row loaded in the corresponding bank's row buffer
- Row buffer conflict: when a row different from the 0th row is loaded in the corresponding bank's row buffer

The following table summarizes the sequence of commands that is required for each of the three scenarios.

Scenario	Commands (Latency)
Row-Buffer Hit	`READ/WRITE`
Row-Buffer Miss	`ACTIVATE`, `READ/WRITE`
Row-Buffer Conflict	`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, at the cycles when commands are to be issued and corresponding data is to be expected. 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:

1. The command and address buses are free at cycles *0, 1, 2, 3, 100, 101, 102,* and *103*
2. The data bus us free during cycles *200-249*
3. The bank is free during cycles *0-99* and *100-199*

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 Assumptions and Policies.

- Assume that the row buffers of all the banks are initially closed, i.e., there is no row in any row buffer.
- The memory controller follows the open row policy, i.e., once a row is opened it is *not* closed unless another scheduled request requires the closing of the row to access another row in the same bank.

3. Lab Resources

3.1. Source Code

The source code that we provide for this lab is the same as the source code we provided for Lab 3. *You are free to choose between starting from scratch using this bare version or continuing with your simulator that includes your modifications for Lab 3 (i.e., instruction/data caches and branch predictor).* If you decide to continue with your previous simulator, you can skip reading the rest of this section. If you decide to start from scratch, you will need to implement L1 data cache and L1 instruction cache, as described in Lab 3.

Do **NOT** modify any files or folders unless explicitly specified in the list below.

- `Makefile`
- `run`: Script that runs your simulator and compares it against the baseline simulator
- `src/`: Source code (**Modifiable; feel free to add more files**)
  - `pipe.c`: Your simulator (**Modifiable**)
  - `pipe.h`: Your simulator (**Modifiable**)
  - `mips.h`: MIPS related pound defines
  - `shell.c`: Interactive shell for your simulator
  - `shell.h`: Interactive shell for your simulator
- `inputs/`: Example test inputs for your simulator (**Modifiable; feel free to add more files**)

3.2. Makefile

We provide a `Makefile` that automates the compilation and verification of your simulator.

The first time you use the `Makefile` you should compile the baseline simulator:

```
$ make basesim
```

This will generate `basesim`, which is the baseline simulator corresponding to the code we provide. You can use it to verify the output of a program you run on your simulator. Note that the output of a program should always match the output obtained by running the program on the baseline simulator. However, the execution time of a program on the two simulators will not be same after the addition of the L2 Cache and the DRAM.

To compile your simulator:

```
$ make
```

To compile your simulator and check it against the baseline simulator using one or more test inputs:

```
$ make run INPUT=inputs/inst/addiu.x
$ make run INPUT=inputs/inst/*.x
$ make run
```

4. Getting Started & Tips

4.1. The Goal

You can skip this section if you are already familiar with the baseline simulator we provided in Lab 3.

We provide you with a skeleton of the timing simulator that models a five-stage MIPS pipeline: `pipe.c` and `pipe.h`. As it is, the simulator is already architecturally correct: it can correctly execute any arbitrary MIPS program that only uses the implemented instructions.[2] When the simulator detects data dependences, it correctly handles them by stalling and/or bypassing. When the simulator detects control dependences, it correctly handles them by stalling the pipeline as necessary.

By executing the following command, you can see that your simulator (`sim`) does indeed have identical architectural outputs (e.g., register values) as the baseline simulator (`basesim`) for all the test inputs that we provide in `inputs/`.

```
$ make run
```

Your job is to model accurately the timing effects of the L2 Cache and the DRAM in your timing simulator.

4.2. Studying the Timing Simulator

Please study `pipe.c` and `pipe.h` in detail.

The simulator models each pipeline stage as a separate function – e.g., `pipe_stage_fetch()`. The simulator models the state of the pipeline as a collection of pointers to `Pipe_Op` structures (defined in `pipe.h`). Each `Pipe_Op` represents one instruction in the pipeline by storing all of the necessary information about the instruction that is needed by the simulator. A `Pipe_Op` structure is allocated when an instruction is fetched. It then flows through the pipeline and eventually arrives at the last stage (writeback), where it is deallocated once the instruction completes. To elaborate, each stage receives a `Pipe_Op` from the previous stage, processes the `Pipe_Op`, and passes it down to the next stage. The simulator models pipeline stalls by stopping the flow of `Pipe_Op` structures and pipeline flushes by deallocating the `Pipe_Op` structures at different stages.

4.3. Tips

- **Please do not distribute the provided program files. These are for exclusive individual use of each student of the Computer Architecture course. Distribution and sharing violates the copyright of the software provided to you.**
- **Read this handout in detail.**
- **If needed, please ask questions to the TAs using the online Q&A forum in Moodle.**
- When you encounter a technical problem, please first read the error messages. A search on the web can usually solve many debugging issues, and error messages.

5. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/`). You should submit all the files needed to compile and simulate your code in a single tarball (with the name `lab4_`*YourSurname*`_`*YourName*`.tar.gz`). Please include comments to explain what you have done in the simulator code.

6. Extra Credit

We will offer up to 100% extra credit *for this lab* for implementing 1) *prefetching* from main memory into the L2 cache and 2) an even more realistic system considering writebacks from the cache and refreshes in the memory controller. A fully correct implementation of Lab 4 is a prerequisite for extra credit.

[2] This is not entirely true since we pose the usual restrictions on system calls, exceptions, etc., as we described in Lab 3.

6.1. Prefetching

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] or locality-based prefetched [2], both of which were discussed in lectures. However, you can also consider implementing more sophisticated prefetchers such as content directed prefetching [3, 4] or Markov prefetchers [5]. If you are really ambitious, you can try implementing runahead execution [6].

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 one cycle (or even less time!). Real hardware, clearly, 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 instructor will be the final judge 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).

Please write a report (`report_prefetching.pdf`) that briefly summarizes *(i)* the prefetching mechanism(s) that you implemented, *(ii)* your observations on the performance and usefulness of the prefetched data, and *(iii)* any other optimizations you implement. Your report does not need to be more than four pages, but feel free to use more pages to present schematics, data, and graphs. Please also submit the version of your simulator (`src/`) that implements prefetching. This version of the simulator and the report should be submitted as a separate folder (called *extra_credit_prefetching*) within the same tarball as the regular submission. You can get 50% additional credit for this extra-credit assignment.

6.2. Writebacks and Refreshes

Alternatively, or in conjunction with prefetching, you can add support for writebacks and refreshes in your simulator. The simulator with support for writebacks should model the writeback of dirty blocks from the L1 cache to the L2 cache, and from the L2 cache to the main memory. You need to ensure these requests occupy resources and take time, similarly to other requests.

The simulator with support for refreshes should model periodic refresh of every single DRAM row. These refreshes should be issued by the memory controller. The refresh rate should be configurable, e.g., a refresh to a row is issued every N cycles. You need to accurately model the contention caused by refreshes to other requests and ensure refresh requests occupy request queue entries in the memory controller, and take time. For this extra-credit assignment, we refer you to the following works if you want to experiment with even fancier state-of-the-art mechanisms [7, 8].

Please write a report (`report_writeback_refresh.pdf`) that briefly summarizes *(i)* the writeback policy and refresh mechanism that you implemented, *(ii)* your observations on the performance and cache hit/miss rate compared to the simulator without realistic writeback and refresh, and *(iii)* any other optimizations you implement. Your report does not need to be more than four pages, but feel free to use more pages to present schematics, data, and graphs. Please also submit the version of your simulator (`src/`) that implements writebacks and refreshes. This version of the simulator and the report should be submitted as a separate folder (called *extra_credit_writeback_refresh*) within the same tarball as the regular submission. You can get 50% additional credit for this extra-credit assignment.

References

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

[2] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[3] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.

[4] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.

[5] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[6] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.

[7] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical report, UT Austin, 2010.

[8] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *HPCA*, 2014.