Lab 1: Simulating and Exploring Cache Behavior

Assigned: Sun., 22.09; Due: **Sun., 13.10** (Midnight)

Instructor: Onur Mutlu

TAs: Mohammed Alser, Rahul Bera, Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gomez Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim, Nika Mansouri Ghiasi, Lois Orosa Nogueira, Jisung Park, Minesh Hamenbhai Patel, Abdullah Giray Yaglikci

1. Introduction

In this lab, you will extend a *timing simulator* (written in C) to model instruction/data caches. Unlike the RTL that you worked with in the Design of Digital Circuits course, a *timing simulator* is *not* a direct or synthesizable implementation of a processor. Rather, it is a higher-level, abstract model designed to allow quick architectural exploration. Describing and simulating hardware at a higher level of abstraction allows the designer to quickly see how different design choices would impact performance.

We will give you the base simulator, which models a simple MIPS processor. We will also fully specify the behavior of the caches. Your job is to extend the simulator so that it implements the caches as specified.

2. Timing Simulator

We are not constrained to logic-level implementation in a C-based timing simulator. Our main goal is to compute the *number of cycles* that a program requires to execute on the simulated processor. Because of this, many simplifications are possible:

- We do not actually need to model control logic and datapath details in each block of the processor.
- We only need to write code for each stage that performs the relatively high-level function of that pipeline stage (read the register file, access memory, etc.).
- In general, the simulator's algorithms and structures *do not* need to *exactly* match the processor's algorithms and structures, as long as the *result* is the same.

While we no longer have a low-level implementation, and thus cannot determine the critical path (or other cost metrics that we care about, such as area taken on a silicon chip, or power consumed during operation), we can know how many cycles a program would take (assuming we model the cycle-level solution accurately).

3. Task 1/2: Additions to the Baseline Timing Simulator

Your goal is to *implement* the timing simulator so that it models a MIPS machine with instruction/data caches accurate to the specifications we provide (below). In the following, we will fully specify the microarchitecture of the MIPS machine that you will simulate.

3.1. Instruction Cache

The *instruction cache* is accessed every cycle by the fetch stage.

Organization. It is a **four-way** set-associative cache that is **8 KB** in size with **32-byte** blocks (this implies that the cache has **64** sets). When accessing the cache, the set index is calculated using bits `[10:5]` of the PC.

Miss Timing. When the fetch stage *misses* in the instruction cache, the block must be retrieved from main memory. An access to main memory takes **50** cycles. On the 50th cycle, the new block is inserted into the cache. In total, an instruction cache miss stalls the pipeline for 50 cycles.

Replacement. When a new block is retrieved from main memory, it is inserted into the appropriate set within the instruction cache. If any way within the set is empty (i.e., invalid), the new block is simply inserted into the invalid way. However, if none of the ways in the set are empty, the new block *replaces* the *least-recently-used* block in the set. For both cases, the new block becomes the *most-recently-used* block.

3.2. Data Cache

The *data cache* is accessed whenever a load or store instruction is in the memory stage.

Organization. It is an **eight-way** set-associative cache that is **64 KB** in size with **32 byte** blocks (this implies that the cache has **256** sets). When accessing the cache, the set index is calculated using bits `[12:5]` of the data address that is being loaded/stored.

Miss Timing & Replacement. Miss timing and replacement of the data cache are identical to those of the instruction cache.

Handling Stores. Both load and store misses stall the pipeline for 50 cycles. They both retrieve a new block from main memory and insert it into the cache.

Dirty Evictions. When a "dirty" block is replaced by a new block from main memory, it must be written back into main memory. For the purpose of this lab, we will assume that such dirty evictions are handled *instantaneously* – i.e., they are written immediately into main memory in the same cycle as when the new block is inserted into the cache.

3.3. Assumptions about Instruction & Data Caches

- Assume that both caches are initially empty (i.e., all blocks are invalid).
- In both caches, every block has a separate tag that stores information about the block: e.g., address, valid, recency, etc. Tags are initialized to 0.
- Assume that the program that runs on the processor *never* modifies its own code (referred to as self-modifying code): a given block *cannot* reside in both the caches.

4. Task 2/2: Cache Exploration

Your second task is to evaluate the performance characteristics of the caches that you implemented as part of Task 1. You must evaluate the following design characteristics:

1. **Cache size, block size, associativity:** a sweep of *cache parameters*. You should write a set of benchmarks that use significant amounts of memory (for example, accessing a large array in streaming or random patterns), and run your simulator to measure IPC for various cache parameters. Show how changing the associativity, block size, and cache size affect performance.
2. **Replacement and insertion policies:** an exploration of cache replacement and/or insertion policies. The *cache replacement policy* specifies which cache block in a set is replaced when a new block is inserted into the cache. The *cache insertion policy* specifies where in the list of blocks the new block is placed. Up to now, we have used a replacement policy that evicts (replaces) the

least-recently-used block, and an insertion policy that places new blocks at the most-recently-used position. However, other replacement and insertion policies have been studied, and some have been shown to achieve significantly better performance (fewer cache misses) for certain access patterns [1, 2]. You should experiment with a variety of test programs and optimize the cache replacement/insertion policy.

3. **Other (extra credit):** Optionally, you may also choose to experiment with other aspects of the cache. For example, using more sophisticated hashing functions to map cache blocks to cache sets and/or using more than one hashing function [3]. Implementing a victim cache is another possibility [4]. Since this part is open-ended, the instructor reserves the amount of extra credit that can be obtained, but up to 25% extra credit is possible depending on the difficulty of the optimization and the goodness of the resulting design and implementation.

Please write a report (`report_cache.pdf`) that summarizes *(i)* your observations on the effect of each cache parameter, *(ii)* your findings on cache replacement/insertion policies, 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 the best performing cache design(s). This version of the simulator and the report should be submitted under their own folder within the same tarball that you submit (see Section 7 for a detailed list of what to submit).

5. Lab Resources

5.1. Source Code

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**)

5.2. 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 your changes on the caches.

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
```

6. Getting Started & Tips

6.1. The Goal

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.[1] 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 caches and the main memory in your timing simulator.

6.2. 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.

6.3. 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.

[1] This is not entirely true since we pose the usual restrictions on system calls, exceptions, etc.

- One way to approach this lab is to first write a generic implementation of a set-associative cache, and then plug it into both the fetch stage (instruction cache) and the memory stage (data cache).

7. 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 `lab1_`*YourSurname*`_`*YourName*`.tar.gz`). Please include comments to explain what you have done in the simulator code.

The structure of the submitted tarball should look like:

- `lab1/`
  - `README`*: (optional) any special instructions you have for us*
  - `task_1/`*: Source code for your cache implementations*
    - `Makefile`
    - `run`
    - `src/`
    - `inputs/`
  - `task_2/`*: Source code and report for your cache exploration*
    - `report_cache.pdf`
    - `most_performant_design/`
      - `Makefile`
      - `run`
      - `src/`
      - `inputs/`*: Also containing your set of benchmarks for memory performance*

Please feel free to include a brief `README` to describe any nuances of building/running your project that we should be aware of when grading your submission. You may also submit additional source code alongside `most_performant_design/` in `task_2/` if you want to show off some aspect of your design that is discussed in your report.

References

[1] M.K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *ISCA*, 2007.

[2] V. Seshadri et al. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *PACT*, 2012.

[3] A. Seznec. A case for two-way skewed-associative cache. In *ISCA*, 1993.

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