Author: Jason Lowe-Power

ECS 154B Lab 4, Winter 2018

Due by 9:00 AM on Mar 12, 2018

Via Canvas

Objectives

- Build a controller for a set-associative and non-blocking cache.
- Analyze the performance of different cache designs with a simulator.

Description

In this lab, you will build a cache model of a set-associative non-blocking caches in C++. Computer architects often use high-level languages like C++ to *model* hardware in early research and development of new designs. Using high-level languages gives architects more freedom and flexibility to quickly iterate with designs than using hardware description languages or actually making hardware.

You will be given a simulation framework and an example direct-mapped cache implementation. You will build your new cache models in this framework and evaluate them. You will be graded in two ways:

1. Correctly implementing your cache models (e.g., the correct number of tag bits).
2. Understanding the tradeoffs between different designs.

DO NOT MODIFY THE GIVEN CODE IN ANY WAY. DOING SO WILL LEAD TO AN AUTOMATIC ZERO. (Unless expressly stated below.) Other rules:

- Do not use any libraries except the C++ STL.
- Your code must compile on the lab machines with no changes.
- To test your code, we will extract your set_assoc.hh, set_assoc.cc, non_blocking.hh, and non_blocking.cc files from your submission and building it with the most up-to- date library files. Any code you write should go into these files, and make sure that you test against the latest versions of code on github.

EXTRA CREDIT: This is my first time giving this assignment. I have written this code over the last couple of days, so it's likely there are bugs. If you're unsure if you've found a bug, post on Piazza. If you find a bug, please open a pull request on Github. **You will receive 5 points of extra credit for each pull request I merge**.

I will accept pull requests with style, grammar, wording updates, but they must be non-trivial to get the 5 points of extra credit.

Details

Given Code

You will find the code in the `lab4/provided/` directory. The code is heavily documented. Please see the code for all details.

There is a Makefile included that will build the `cache_simulator`. The cache simulator currently has a very simple test implemented. In `main.cc`, a processor, a cache, and a memory are instantiated. Then, the simulation is executed.

Processor

The processor drives the simulation. It sends memory requests to the cache by calling `receiveRequest` on the cache object. The processor will continually send requests to the cache until the cache responds that it is blocked (`receiveRequest` returns `false`). There requests may be sent multiple in a single *tick* or cycle.

The test that is run is initialized in `Processor::createRecords()`. This function populates a queue which is the "trace" of memory accesses. See `processor.cc` for details.

Feel free to add your own tests and/or modify the tests in `createRecords`. When we test your code, we will use a different implementation of `Processor::createRecords`.

Cache

You will be inheriting from the `Cache` object. See the code in `cache.hh` for details of the interface you will implement. You will need to implement the following functions: - `receiveRequest`: Called by the Processor when sending a new request. - `receiveMemResponse`: Called by memory when it has finished reading the data previously requested by the cache.

DO NOT MODIFY THIS INTERFACE OR THE IMPLEMENTATION OF THE NON-VIRTUAL FUNCTIONS.

There are two functions that are implemented in the base `Cache` class that you will need to use.

- `sendResponse`**: Call this function when you want to send a response to the processor. It takes two arguments.**
    - `int request_id`: When the processor sends a request it includes a request ID. This is needed for the processor to track which request it is receiving a response for. You must save this request ID from when the processor sent the request and use the *same* ID when responding.
    - `const uint8_t* data`: This is the data you are returning to the processor on loads. On stores, the data must be `nullptr`. The processor will copy the data out of this pointer.
- `sendMemRequest`**: Send a request to memory to either read or write an address. You will call this when writing back data from the cache or to fetch data on a miss. It takes four arguments:**
    - `uint64_t address`: The address that you want to read or write.
    - `int size`: The size of the request (this should always be the memory line size).
    - `const uint8_t* data`: If writing back this pointer should hold the data to write. The memory will copy the data from this pointer.
    - `int request_id`: Like when the processor sends a request, when the cache sends a request to memory it needs a way to know which request it is getting a response for. Thus, you can choose any `request_id` you would like when sending the request to memory. When memory responds to the request, it will use that ID. I suggest using the MSHR table index.

Memory

Memory receives requests from the cache. After some delay, it will respond to the request. The delay is guaranteed to be more than 1 tick. This object also has an interface for you to determine the line size. The memory line size is used as the cache line size.

TickedObject

This object implements the discrete event simulation. Feel free to ignore it :).

DirectMappedCache

This is an example cache implementation. Feel free to base you implementation of the set-associative cache on the direct mapped cache that is included. You can use any code from the class in your set-associative cache.

TagArray and SRAMArray

DO NOT MODIFY THIS CODE IN ANY WAY.

You are required to use the `TagArray` and `SRAMArray` to hold tags/state and data, respectively. Constraining yourself to this interface will help ensure your cache controller design is realistic.

The `TagArray` holds both tag and the state for each line. There is only room for a single tag and state for each line, so you will have to consider how to implement a set-associative cache. The `TagArray` takes the number of tag bits and data bits as parameters. We will check to make sure you use the correct number of tag and data bits.

The `SRAMArray` is a simple data storage array that only takes the size of the data (cache block) and the number of elements to store as parameters. You will instantiate multiple tag and SRAM arrays for your set-associative cache implementation.

Implementing a Set-Associative Cache

You are given a file `set_assoc.cc` which has empty functions for each function you are required to implement. You will be modifying and turning in this file and `set_assoc.hh`. Feel free to add new private/protected functions to the `SetAssociativeCache`, but **DO NOT MODIFY THE PUBLIC INTERFACE**.

You implementation of the set-associative cache should behave *exactly the same* as the direct-mapped cache when there is only one way. It should also be able to be any associative up to fully-associative (unless you explain why this is not possible as described below).

You may choose to implement any replacement policy. **However, if your replacement policy requires any state, you must store this state in the tag array!** Note: The tag-array state is limited to 32 bits per line. If this constrains your associativity, be sure to include assertions in your code and explain this in the README file (see below).

You **must** use the given TagArray and DataArray objects to store your tags and data.

It may help testing your implementation to think about the following questions: (You will be answering these questions in lab 5.)

- How does increasing the set-associativity from direct-mapped to 8-way set-associative affect the hit ratio?
- How does increasing the set-associativity from direct-mapped to 8-way set-associative affect the performance of the system?

Implementing a Non-Blocking Cache

Your non-blocking cache should inherit from your set-associative implementation as it must be both non-blocking and set-associative. Much of the code will be shared between the set-associative and the non-blocking cache.

We discussed non-blocking caches in class. To implement a non-blocking cache, you must track outstanding requests from the cache. This is usually done with a table of miss status handling registers (MSHRs). You may choose any software implementation of this you would like. However, you will be expected to explain how your software implementation would translate to hardware in interactive grading.

It may help testing your implementation to think about the following questions: (You will be answering these questions in lab 5.)

- How does increasing the number of MSHRs affect the performance of the system?

Testing and Grading

You should expect that we will test the following things: - Changing the size, associativity, and number of MSHRs of the cache. - Changing the line size of the memory. - Sending requests of any power-of-two size to the cache from the processor that are less than or equal to the line size. - Changing the address width.

Note, we may use different implementations of `TagArray`, `SRAMArray`, `Memory`, and `Processor` when testing. The only thing you can rely on is the interfaces in the header files. The underlying implementation may change.

Think about the following questions for interactive grading:

- How does my software implementation compare to a real hardware implementation?
- What parts of the hardware does your code model explicitly and which parts of the hardware design are not modeled?

Name	Percentage of Lab Grade	Description
set_assoc.cc	25%	Implementation of the set-associative cache. This includes the correct number of tag/index bits, etc.
non_blocking.cc	25%	Implementation of the non-blocking cache. This includes the correct implementation of MSHRs and per-line state.
Interactive Grading	50%	Your answers to the questions will be evaluated during interactive grading. 25% of your overall grade will come from the answers to these questions and the other 25% from your answers to the interactive grading questions.

Submission

Warning: read the submission instructions carefully. Failure to adhere to the instructions will result in a loss of points.

- Upload to Canvas the zip/tar/tgz of your `set_assoc.cc`, `set_assoc.hh`, `non_blocking.cc` *and* `non_blocking.hh` file along with a README file that contains:
  - The names of you and your partner.
  - If there are any constraints on the associativity of your cache, explain what they are and why.
  - Any difficulties you had.
  - Anything that doesn't work correctly and why.
  - Anything you feel that the graders should know.

- **Copy and paste the README into the comment box when you are submitting your assignment**, as well.
- Only one partner should submit the assignment.
- You may submit your assignment as many times as you want.

Hints

- This is a new assignment. There may be bugs. I will update the git repository if/when bugs are found. It will be best for you to clone the repo so you can pull new changes as they happen.
- We discussed the state machines for the caches in class. Use these ideas when implementing your cache models.
- Use the DPRINT macro in `util.hh` for debugging. This allows you to insert extra debugging print statements that are easily turned off (see the Makefile).
- Using Git (or any other version control system) may prove useful for keeping history in case you need go back. Make sure you commit at reasonable times with reasonable messages :).
- If you find a bug, submit a pull request on Github! You'll get 5 points of extra credit on the assignment per PR that I accept!