Computer Architecture 2020 Lab 3

Cache Lab

Table of Contents

Update: 200506, 200503, initial version

Update description: https://piazza.com/class/k6hx3dcek482q1?cid=116

Overview of this lab

It was midnight, 1am. The TA was struggling writing archlab 3 for Yao students. He had homework from four (!) of his own courses to finish, and also needed to grade the midterm exam papers that Yao students just submitted. It was impossible to release lab 3 on time! He thought to himself. He was so desperate until he found you, the smartest one he ever met, to help him finish the design of lab 3. Save the TA, please!

In lab 3, building upon your achievements in the previous labs, you are going to further optimize the processor microarchitecture for better performance. Specifically, you are designing a high-performance cache between your processor and the memory, to significantly reduce the memory access latency, which has created a lot of pipeline stalls in the previous labs.

Before starting this lab, you are supposed to be equipped with the following:

- The same environment as previous labs
- Basic C++ programming language knowledge
- A correct implementation of image processing (lab 1)
- A well-functioning timing processor (lab 2)

After this lab, you will obtain the following:

- Understanding various cache designs and replacement policies
- Better understanding about the difference between code optimization and microarchitecture design

During this lab, you are suggested to refer to the following materials:

1. How to resolve merge conflicts in Git

   https://stackoverflow.com/questions/161813/how-to-resolve-merge-conflicts-in-git

2. Cache Replacement Policies

   https://en.wikipedia.org/wiki/Cache_replacement_policies

3. ZCache paper

   https://people.csail.mit.edu/sanchez/papers/2010.zcache.micro.pdf

4. Hawkeye paper

   https://www.cs.utexas.edu/~lin/papers/isca16.pdf

During this lab, an infrastructure has been provided in the following tree form:

```
├── lab3
    ├── lab3-handout.pdf --- This document
    ├── run --- Our workspace folder
        ├── lab1-champion.S --- An image processing assembly that wins lab 1
contest
    ├── testcases --- Two testcases for your assembly
        ├── simplecase.in/out --- A 8*8 small input and correct output pixel
file
        ├── bigcase.in/out --- A 512*512 input and correct output pixel file
├── spim-timingmodel/CPU
    ├── lab3_def.h --- Constants for lab 3
    ├── timing_array.cpp/h --- A basic class for data array and tag array.
Latency is updated after each access
    ├── timing_cache.cpp/h --- Definition of TimingCache class
    ├── timing_mem.cpp/h --- Another memory simplified from original SPIM mem
    ├── timing_cache_custom.cpp/h --- Definition of a cache controller for
TimingCache, where an incomplete implementation has been provided
```

Task

In lab 3, you have only one goal: finish the image processing task as fast as possible, **by designing a high-performance timing cache**. You should:

- Integrate a basic cache into your existing timing core.
- Design a better cache.
- Write a design document.
- [Bonus] Write a short essay about *Algorithm Design vs. Assembly Optimization vs. Microarchitecture Design*

1. Integrate with TimingCache Framework

First make sure you have the identical directory structure as in "Provided Infrastructure". And be sure your timing core functions correctly under lab 2.

Now we take the example steps of merging into “lab2-lecturestyle” branch, as this is used by majority students. If you finished your lab2 in the original “lab2” branch, please try merge the code by manually checking the difference (git diff tool). Post the Git diff to Piazza if you don't know how to merge a specific conflict,

```
$ cd ~/yao-archlab-s20/spim-timingmodel/

$ git add CPU/*
```

```
$ git commit -m “My lab 2 final implementation!” # back up your work

$ export LAB3=$HOME/yao-archlab-s20/lab3

$ git fetch --tags

$ git checkout -b lab3 lab3-v1.1

$ git merge lab2
```

At first Git would try auto-merging your commits and the lab 3 commits. But because they are divergent in different commit paths and both modify the same files, inevitably you will see something similar to the following:

```
* branch lab3 -> FETCH_HEAD

* [new branch] lab3 -> origin/lab3

Auto-merging CPU/timing_lsu.cpp

CONFLICT (content): Merge conflict in CPU/timing_lsu.cpp

Automatic merge failed; fix conflicts and then commit the result.
```

You should manually edit the conflicted files after comparing the different parts. In some situations, simply use code from one commit does NOT work. You should carefully merge your existing TimingCore code and the lab 3 code to support TimingCache. There may be multiple failed merges. Go through the conflicting files one by one. And there are several visual tools helping with the merging process (refer to the extended material 1). If you don't know how to merge a specific conflict, post the Git diff to Piazza.

After fixing the conflicts, stage and commit again:

```
$ git add CPU/*

$ git commit -m “Merged conflicts”
```

After compiling and soft linking (refer to previous lab handouts), your timing core should work now with a timing cache, though this cache does not yet buffer anything at this time.

```
$ ./spim -lab3 file simple_add.S
```

Apart from correct output, if you see an extra line prompting `Timing Cache is enabled now.`, your merge has been successful.

Before starting your design, you should decide which assembly you are working on. After requesting the consent from a warm-hearted classmate, there is a public `lab1-champion.S` program in `run/`. He has applied many optimization methods to keep the cycles to minimum. Note that he might use different instruction types that your timing core does not support. If you are using his program, extend your timing core

first. Or you can stick on your own lab 2 assembly. Please specify the assemble filename in your final document. From now on we call your choice `process.S`.

Regardless of which version you are using, you can **NOT** modify the program anymore from now on.

Now record your performance (total cycles) on lab 2:

```
$ ./spim -lab2 file process.S -redirect ../testcases/bigcase.in
myoutput.out
```

Remember to make sure your output is correct for each performance recording (ignoring end-of-line spaces):

```
$ diff myoutput.out ../testcases/bigcase.out
```

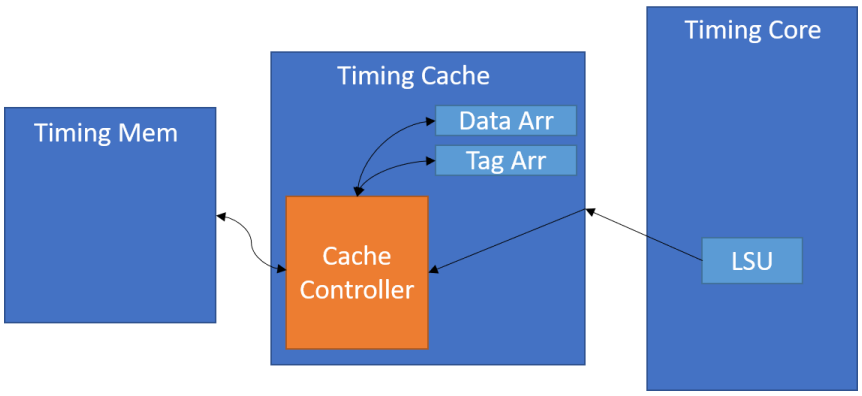
Then record your performance on lab 3 (incomplete cache), including the total cycles on TimingCore and TimingCache, and the cache miss rate:

```
$ ./spim -lab3 file process.S -redirect ../testcases/bigcase.in
myoutput.out
```

You will notice a 100.00% miss rate, which you need to improve in the next task.

2. Complete a Two-Way Set-Associative Basic TimingCache

The following figure illustrates the relation between different components in the timing cache, our memory, and your timing load/store unit (LSU). From this figure we know that our fetcher still uses 1 cycle to fetch instructions (as a perfect I-Cache would do), but we will have an independent timing memory and data cache, which handles access requests from load store unit (LSU).



In lab 3, you are only allowed to modify `timing_cache_custom.h/cpp`, which is the main controller (shown in orange in the figure) in our cache. For each memory access requested by the timing LSU, our cache would forward the request to the controller. The controller would (normally) do the following steps: hit checking, fetching memory block under misses, and returning correct cache line. Note that

now memory access would spend variable cycles for different requests, but if you implement your lab 2 correctly, your core could handle it correctly.

Besides the controller, the cache also contains a data array of data blocks, and a tag array of tags, valid and dirty bits, and other metadata if you need. During request handling, the controller may read and write the two arrays for one or several times, including checking valid bits and tags, accessing data blocks, filling in new data blocks from memory, and other actions. Each access will incrementally apply an SRAM latency onto the request. The two arrays are the **ONLY** places that can store data blocks and metadata in the cache. Therefore, **you are NOT allowed to define any global or static local variable or to add any class member**. Similar to lab 2, you can **NOT** directly use SPIM's memory and register. You can still create local functions and use local variables within each method that does not live across method calls.

The good news is that we have provided an incomplete cache controller framework for you with the example format of tag entry and usage of the two arrays. Check the code comments for detailed information. However, the TA does not finish the framework due to limited time. Several code lines are left to be corrected. Now in this subtask, you need to first get familiar with the existing controller workflow and then fill in the remaining code blanks. Each blank is commented with a `FIXME` tag.

The TA would like to design a cache with writeback, write-allocate and fetch-on-miss policy (refer to lec10, page 11). Moreover, it should be a 2-way set-associative cache, with a FIFO replacement policy, i.e., evicting the cache line which is allocated the earliest in a set.

Before you believe you have understood the workflow and start coding, ask yourself about the following technical details:

- How large is the cache data array and tag array? How many cycles are there for each SRAM access?
- What are the differences of the concept *cache line*, *word* and *byte* (in terms of address interval)? Can you explain the address translation scheme in `timing_cache_custom.h`?
- How to get the "index" part of an address?
- How to mask out the "offset" part of an address to get the start address of a cache line?
- How to check the tag of each way in one set? How many tag array accesses do you need to check a hit?
- How should we decode an address into set-associative locations, i.e., the mapping function of `addr => (idx, way)`, where `idx` for the set id and `way` for the way id if the data actually resides in the cache?

- We use `ptr` to identify the start address of a specific block in either SRAM array. What is the mapping function of `(idx, way) => ptr` in data array? What about tag array?
- How to get and set one cache line in the data array?
- How to get and set one tag entry in the tag array?
- What is the critical path for one cache access? Which operations can be done off the critical path? By how? How does SPIM-T measure your off-critical operation time?
- What are the input and output formats of `lw/lh/lb`?
- What are the input and output formats of `sw/sh/sb`?

After you complete all blanks, record your performance (basic cache):

```
$ ./spim -lab3 file process.S -redirect ../testcases/bigcase.in
myoutput.out
```

3. Design a High-Performance TimingCache

The TA did not prepare any optimization framework for Yao students. Please help him select and implement two optimization methods, to release lab 3 on time.

In this task, you are required to implement two of the many optimization methods to design a high-performance cache.

Cache Replacement Policies

When designing a set-associative cache, the cache replacement policy is important when several hot blocks compete for limited locations. There are a number of rank-based policies: FIFO, RANDOM, LFU, and LRU, which adjust the rank list in different ways after each access. In one set, the block with the lowest rank will be evicted to make room for the new block. Here is a simple introduction of various replacement policies (refer to extended material 2).

Specifically for LRU, there are two choices for insertion policy, i.e., how to allocate a block, in addition to the replacement policy. LRU-MIP inserts the cache line with the most recency, i.e., highest rank ("I believe this new block must be accessed very soon"), while LRU-LIP inserts with the least recency, i.e., lowest rank ("I believe this new block must be evicted very soon unless there is a second access." Think of when this policy will be helpful). Note that insertion policy only determines the block rank at allocation. If a later access hits in the block again, it will be promoted to the highest rank as in basic LRU.

Implement the above 5 policies, FIFO, RANDOM, LFU, LRU-MIP, LRU-LIP. You may need to redesign the format of the tag array to record the necessary metadata for each policy.

Here is the original paper about insertion policy. You are encouraged to find more interesting replacement policies to implement here (not required).

Display the hit rates with all 5 policies in a bar chart, and discuss about your plot.

Off-Critical Path Exploiting

Generally, cache writebacks can be executed in the background, because stores are not so urgent comparing to loads. Realizing a writeback queue to write memory in the background seems a good choice. Because a temporary buffer is not allowed in this lab, you should redesign your data array, reserving a small space for buffering data to be written back. During `ProcessOffCritical`, try draining the queue by writing to memory at the proper time. Note that this does not mean our cache can handle multiple tasks in one cycle. We just try exploting idle cycles for the cache. Even if you put writebacks off the critical path, when there are lots of normal memory accesses, your cache may drain the queue and cannot respond to them in time and your processor performance can still be influenced. You should also be careful about correctness: the most up-to-date values might not be committed to memory immediately but still in the write queue.

Based on a simple write queue, you are encouraged to apply more optimization methods, e.g., merge writes of the same cache line (not required).

Record your performance before and after your design.

Cache Bypass

When we have a large work set and a relative small cache capacity, we may encounter cache thrashing. Therefore we need to bypass some of the requests, in order to make our cache at least hit for the other requests. It seems nice if we can discard unimportant access requests, and only allocate cache lines for those really hot blocks. But how do we know the hotness before we cache them? We can make the classification by PC. Generally, if blocks are accessed many times, i.e., are “hot”, and they are caused by the same PC, we can predict new accesses by the same PC would also be hot. You can reserve some space from the tag array to design a hash table that is indexed by PC and provides a boolean result about whether to cache or not. Update the hashtable if your cached blocks are not actually hot.

Record your performance before and after your design.

Research: Hawkeye

Similar to cache bypass technique, we can use more complicated ways to learn the hashtable. In algorithm course, you may have learned the best cache replacement policy called the Belady's MIN algorithm, which gives the theoretical best replacement choice if knowing all future accesses. In a real cache, although we cannot learn the future accesses in advance, we can learn the past by storing access history, and applying Belady's algorithm to the history. Then by analyzing the relation of "should-have-cached-blocks" and their PC, we predict whether to cache new accesses according to their PC, or dedicate a proper insertion rank to different blocks. Refer to Hawkeye paper for more information. You are allowed to unlock `timing_cache_custom.h` and add additional class members related to Hawkeye implementation.

Record your performance before and after your design.

Skew-associative Cache

You may notice that our tag array has a limited capacity for each set. Therefore we cannot increase the associativity for better performance. However, if the program shows different utilizations on different sets (for example, a pathological example always accesses high addresses in stack and low addresses in heap, with few accesses to the sets of middle addresses), we can increase the data array utilization by allowing blocks being in different sets.

Normally our basic cache uses the same set for an address, and only chooses between different ways, i.e., `(idx, way1)` and `(idx, way2)`. In skew-associative caches, we apply different hash functions on each way `H1, ..., Hp`, where `p` is the number of ways in a set. Now the possible locations of an address become `(H1(tag, idx) % Nset, way1), ..., (Hp(tag, idx) % Nset, wayp)`, where `Nset` is the number of sets. A good hash function can decrease the conflicts of different addresses and increase cache utilization. Think about what hash function combination would fit for higher cache utilization.

Record your performance before and after your design. Also note down the change of utilization percent (valid blocks/all blocks). For example, you can plot a chart, where the x-axis is the current_cycle and y-axis is the utilization rate of your cache. Note down your best hash function combination.

Research: ZCache

In skew-associative cache design, we still have only `p` candidates for evicting blocks. How about we find a better placement plan for all blocks in the cache, to make room

for the incoming blocks? The placement transformation can be represented by a swapping chain. And the swapping chain can be applied off the critical to further save your time. Refer to the Figure 1 of the ZCache paper for more information. You are allowed to unlock `timing_cache_custom.h` and add additional class members related to ZCache implementation.

Record your performance before and after your design.

Others

If you have more interesting ideas, please post your (public or private) ideas to Piazza with "possible cache idea:" prefix. After the TA or the professor agree on your idea, this design counts one optimization.

Write-up

You should write a design document for Lab 3. There is no specific format required, but you should demonstrate how your cache works, in a clear way. You should at least include the following parts:

Methodology

- How did you finish the lab? Show your debugging process if there is anything interesting.
- Briefly describe the assembly you choose and optimization methods you apply. What are their pros and cons if applied to real hardware design?
- Record the access latency cost of all possible access paths in the following table. You can append new access paths to the table according to your design.

Access path	Latency cost
Read hit req.	TAG_READ + DATA_READ + TAG_WRITE
Read miss req. without writeback	
Read miss req. with writeback	
Write hit req.	
Write miss req. without writeback	
Write miss req. with writeback	
New case (if any)	

Results

Record your performance under this configuration:

- User program: `process.S`
- Input to user program: `testcases/bigcase.in`
- Correct output of emulation: `testcases/bigcase.out`
- Cache data array and tag array: default capacity

Compare the performance of your designs in the following table, and briefly discuss the results. Which optimization is more effective?

Design	Miss rate	AMAT (in cycle)	Total cycles on TimingCache	Total cycles on TimingCore
Lab 2 (no cache)	100.00%	150	0	
Lab 3 (incomplete cache)	100.00%			
Lab 3 (basic cache)				
Lab 3 (opt 1: ...)				
Lab 3 (opt 2: ...)				

Plot a graph about miss rate on different cache line sizes, associativities, and cache capacities. You can modify `timing_cache.h/cpp` and `lab3_def.h` to make your config right. Briefly discuss your result if you have interesting observations.

Typical design space:

Cache line size: 16B, 32B

Associativity: direct mapping, 2 way, 4 way

Cache Capacity: 512B, 1KB, 2KB

Question Answering

- There is one more optimization called non-blocking cache, which can continue to handle requests even a prior miss request is on the fly, e.g., the cache is

fetching data from memory and handling hit requests in the same cycle. This design combines ideas of pipelining and exploiting off-critical path. Do you think it is useful in your processor? In your opinion how much is the cost and benefit?

- There is one more optimization called compiler prefetching. The MIPS ISA can be extended to add a new "prefetch offset($rs)" instruction by the compiler. This instruction fetches the target block to the cache, but not fills in any register yet. Therefore the actual load instructions can be cache hits with small delays. Obviously it should be added several instructions ahead of actual load instructions, in order to prefetch the target block. Do you think it is useful in your processor? In your opinion how much is the cost and benefit?
- The TA would like to build a multi-level cache framework next year. Is it possible? Can you point out the parts that should be modified in our current framework?

Create a folder with the name of `lab3-<student ID>-<first name>-<last name>`, e.g., `lab3-2018011222-xiaoming-wang`. Put the following files in it.

Upload the package to `learn.tsinghua.edu.cn` (网络学堂) by **May 22**. You may submit for multiple times, and we will grade based on the latest submission.

```
+-- lab3-designdocument.pdf --- Your Lab 3 design document
+-- lab3.S --- The assembly you used for emulating with your own processor
+-- CPU/* --- Your implementation of timing core and timing cache. Remember
that we will overwrite files unrelated in this handout and files marked with
`Unmodified`.
```

Example usage: `$ zip lab3-2018011222-xiaoming-wang.zip -qr lab3.S CPU/ lab3-designdocument.pdf`

Bonus Essay (not required)

When we retrospect our yao-archlab, it is a long journey of blood, sweat, tears and joy (maybe). Eventually we have built a MIPS emulator with timing statistics, which has its own processor, cache and memory, and it works well even without SPIM. Hopefully you now understand the main ideas of microarchitecture, flows of system design and tips for building and debugging a project step by step. After the TA finished the last line of code of our archlab, he was considering this question: We learned basic techniques about **code optimization** (compiling techniques) in lab 1, **microarchitecture optimization** in lab 2&3 (high-performance cache design and full forwarding, if anyone realized it). Although we did not cover algorithms in the labs, OIers must have played with **various algorithms** since long ago. Which topic is the most *important*, where “important” means better performance in terms of execution cycle reduction, more promising research directions, and higher likelihood to make better lives of human beings. How do you like each of these three? This is an open question with no standard answers. Please write over 300 words to get points.

Also, if you like, you can write down your feelings and thoughts about our archlab (alpha version). Which part looks interesting and which part not? Which part should be improved next year? Which part is unfair? These thoughts would not have any positive or negative impact on your grades.

Grading policy

Plagiarism is **strictly** forbidden in this lab. Peer discussion is **not** suggested. Contact the TA if you are in trouble.

- 30% - Successfully merged TimingCache and TimingCore, and implemented basic timing cache.
- 40% - Applied two optimization methods on the basic timing cache.
- 10% - Design document: Methodology and result
- 10% - Design document: Question answering
- 10% - Performance contest
- 5% - Bonus essay.

It **is** possible that your lab 3 score will be over 100 points with the bonus.

Correctness

We will emulate your program to generate output pixel files with two test cases. One is an 8 * 8 small image and the other is a 512 * 512 big image. Then we compare them with the pre-processed ground-truth. More matches mean a higher correctness score. Note that we will use different image pairs for grading!

In this lab we will check your detailed cache implementation. Please have enough comments in your code.

If you did not finish previous lab completely, contact TA.

Performance

As for basic cache design, you should reach **30% hit rate** to acquire full points.

The number of cycles of memory accessing in your **cache** should be as small as possible. Our contest will rank everyone's performance on a 512 * 512 images.