Cache Lab

Overview of this lab

In this lab, 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
- The same processor model we used in Lab 1 (rv5s)

After this lab, you will learn the following:

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

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

1. Cache Replacement Policies
2. Pseudo-LRU Policies
3. Adaptive Insertion Policy
4. Skewed-Associative Caches

Provided Infrastructure

In this lab, we provide the following files in the tree:

```
├── lab3
    ├── lab3-handout.pdf --- This document
    ├── testcases --- Two testcases for your performance
        ├── benchmark1.c
        ├── benchmark2.c
        ├── benchmark1_data.txt
    ├── dip.pdf --- Adaptive Insertion Policy paper
```

Goal

In this lab, your goal is to **design high-performance caches**. We have provided you two basic cache implementations with simple policies (Random, LRU). You should:

- Implement a Pseudo-LRU replacement policy
- Implement two more advanced replacement policies: LRU-LIP, DIP

- Implement a cache optimization method: skewed-associative cache
- Profile the effects of different cache configurations: associativity, cache write policy, write miss policy, etc. Design benchmarks that prefer different cache schemes.
- **[Bonus]** Write a short essay about *Algorithm Design vs. Assembly Optimization vs. Microarchitecture Design*

Tutorial of Cache Simulation in Ripes

The Cache Simulation Module

The cache module in Ripes is in `Ripes/src/cachesim`. It can be used to test the performance (hit rate, number of hits, etc.). All the code that you need to write in this lab is in `Ripes/src/cachesim`. We will leave out the directory path in the following content. For example, when we refer to `cachesim.cpp`, it means `Ripes/src/cachesim/cachesim.cpp`.

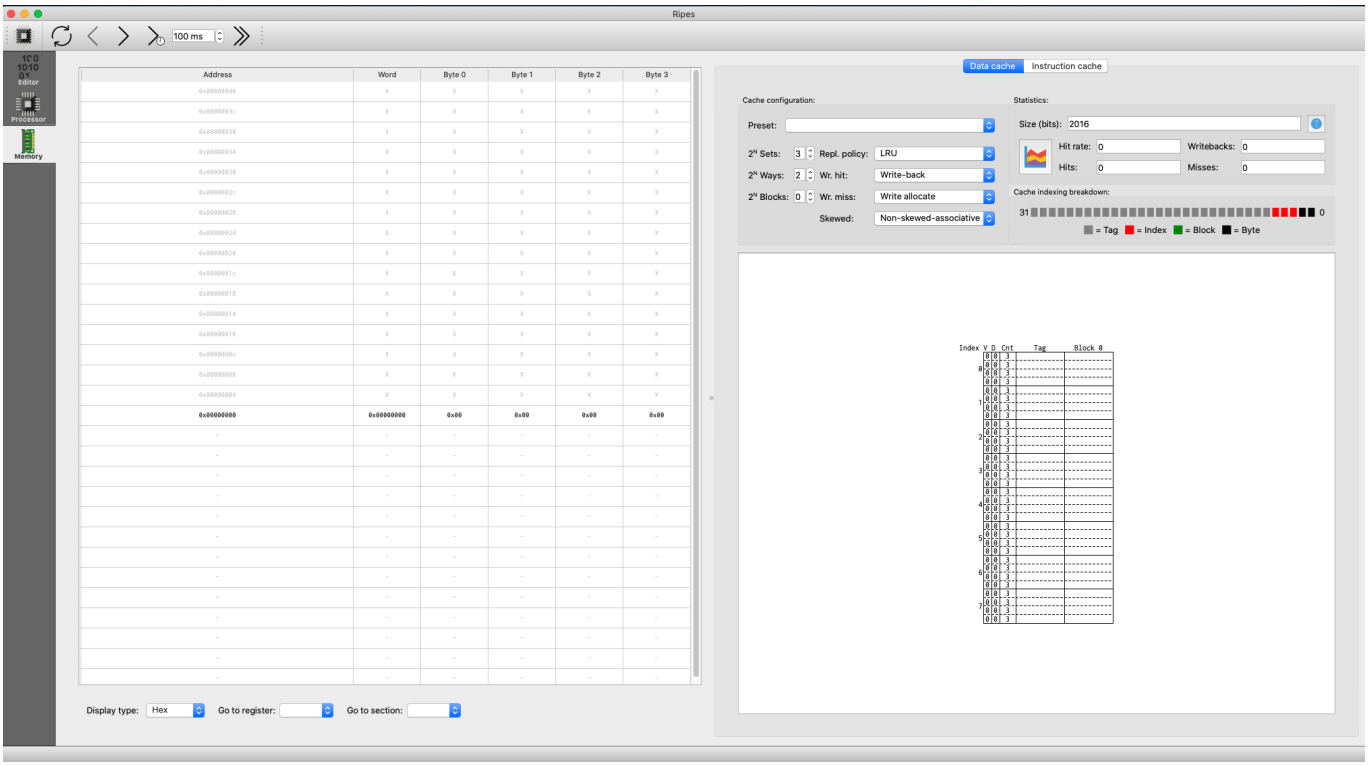
The major declaration of cache simulation is in `cachesim.h` and the implementation is in `cachesim.cpp`. Reading the code of these two files can help you finish the lab.

The cache and the memory are in the MEM stage together. That is to say, the memory address accessed and the `pc` value are the same in the cache and in the MEM stage. Actually, the cache module gets the memory address from the MEM stage in the Ripes implementation.

The Memory Tab

We can play with the memory tab in Ripes. The configurations are quite self-explained. You can select the replacement policy (Random, LRU, etc.), the write hit policy (write-through/write-back), the write miss policy (write-allocate/non-write-allocate) and skewed/non-skewed-associative cache after your finish Task 3 below. Also, you can change the numbers of sets, ways in a cache set, and blocks in a cache way.

The results will be shown on the right panel, including hit rates, number of hits, etc..



Implement a New Replacement Policy in Ripes

To implement a new policy in Ripes, you need to declare the new policy in `cache_policy_object.h`. The new policy will be declared as a derived class based on the abstract class `CachePolicyBase`. There are 3 protected member variables in `CachePolicyBase`:

- `sets`: number of cache sets in the cache.
- `ways`: number of cache ways in one cache set.
- `blocks`: number of cache blocks (cachelines) in one cache way. In Lab 3 and in our lectures, there will always be one and only one block in one cache way, that is to say, `blocks ≡ 1`.

The `sets`, `ways` and `blocks` can be accessed inside all derived classes of `CachePolicyBase` since they are protected variables.

You need to do the **following work** to implement a new replacement policy in Ripes:

Step 1. Declare a new replacement policy (optional)

We have done this for PLRU, LRU-LIP, and DIP, so you do not need to do the declaration yourself. But if you want to implement a new replacement policy, you need to do so.

Before you implement a new replacement policy, you need to declare it first. You need to do the following:

In `cachesim.h`:

- Add a new value to the enum type `ReplPolicy`. The current values are `{Random, LRU, LRU-LIP, PLRU, DIP, NoCache}`.
- At the end of `cachesim.h`, add an entry to `s_cacheReplPolicyStrings` to make it visible on the Ripes GUI window.

In `cachesim.cpp`:

- In the implementation of the function `setReplacementPolicyObject`, call the corresponding constructor of the replacement policy according to the current chosen policy.

Step 2. Implement the following functions

This is what you need to do in [Task 2](#2. Implement Advanced Cache Replacement Policies) below.

In `cache_policy_object.cpp`:

- `locateEvictionWay`: This function is called when a cache miss occurs and a new cache way needs to be chosen to store the accessed data. The eviction choice is up to the replacement policy. For example, when using LRU, if the cache set is not full, we can simply choose a cache way whose data is invalid. If the cache set is full, the LRU cache way, which has the highest `counter` value `ways - 1` (see below), will be chosen.
- `updateCacheSetReplFields`: This function is called to update the information used by the replacement policy. For example, in LRU, we maintain a variable `counter` for each cache way. When a cache block in a way is accessed or inserted, we set `counter = 0` for this way. And values of `counter` on other ways will be increased. When we need to evict a cache block, the block in the way with the highest `counter` value will be evicted. You can refer to the `updateCacheSetReplFields` implementation of `LruPolicy` for more details.
- `revertCacheSetReplFields` **(optional)**: This function is called to support the reverse operation 

in Ripes, i.e., to revert the impact of the last operation on the replacement policy information. If you do not need the reverse operation to debug, you can leave this function empty. The reverse operation will not be tested during grading.
- The constructor: The constructor of the cache policy object. You can leave the function empty if you do not do any work in the constructor, such as we did in LRU. But remember to implement it. Otherwise there will be a compilation error.
- The destructor: The destructor of the cache policy object. You can leave the function empty if you do not do any work in the destructor, such as we did in LRU. But remember to implement it. Otherwise there will be a compilation error.

If you find it hard to understand the grammar of the C++ code, you may want to search the following key words: *C++ inheritance*, *C++ separate header and implementation*. Also, feel free to ask questions.

Step 3. Compute the cache size (optional)

Implement the function `getCacheSize` in `cachesim.cpp`. You need to analyze the implementation cost of the replacement policies, including the cache size, in your **design document**. You are **not required** to implement this function, but otherwise you have to do the calculation manually.

Tasks

1. Setup Environment

```
$ cd ~/yao-archlab-s22/Ripes/
$ git pull
$ git checkout lab3-v1.0
$ mkdir -p build && cd build
$ cmake ../
$ make -j4
```

Note: You must checkout the `lab3-v1.0` branch and finish your lab on it.

In Lab 3, we are going to use the same processor as used in Lab 1, which is called **"5-Stage Processor"** in the Ripes window.

In Lab 3, a cache hit will incur a 2-cycle memory stall, while a cache miss will incur a 10-cycle memory stall.

2. Implement Pseudo-LRU Replacement Policy

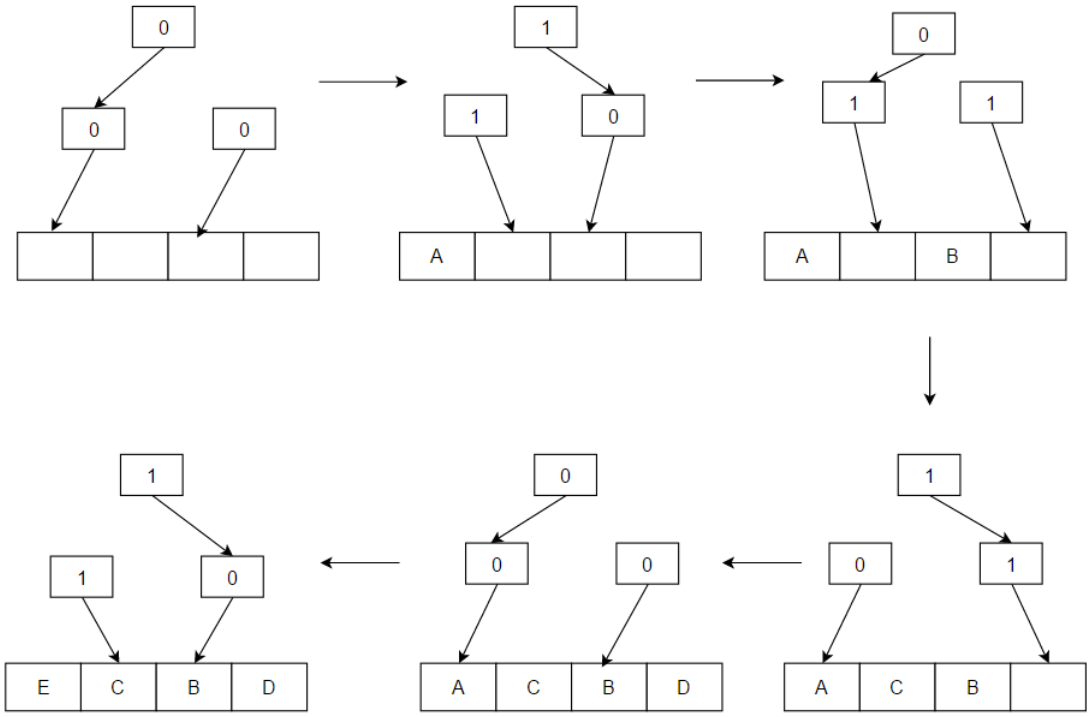
Although Least Recently Used (LRU) is a great idea for cache replacement, it requires relatively high hardware costs to maintain the exact least recency information. Specifically, for each set of an n-way set-associative cache, $n\lceil\log n\rceil$ bits are needed for the counters of LRU. It is natural to consider whether we can design an approximate LRU algorithm.

Pseudo-LRU or **PLRU** is a family of replacement algorithms which simplity LRU by using approximate measures of age rather than maintaining the exact age in the cache. One of the most common PLRU policies is Tree-PLRU, and you are required to implement it.

To implement the policies, you need to follow [Step 2](#Step 2. Implement the following functions) of the tutorial above. We have provided you the implementation of the Random policy and the LRU policy as references.

2.1 Tree-PLRU

The algorithm works as follows: consider a binary search tree for the items in question. Each node of the tree has a one-bit flag denoting "go left to find a pseudo-LRU element" or "go right to find a pseudo-LRU element". To find a pseudo-LRU element, traverse the tree according to the values of the flags. To update the tree with an access to a specific item, traverse the tree to find this item and, during the traversal, set the node flags to denote the direction that is opposite to the direction taken.



3. Implement Advanced Cache Replacement Policies

In this task, you are required to implement two advanced cache replacement policies, following [Step 2] (#Step 2. Implement the following functions) of the tutorial above.

3.1. LRU-LIP

For LRU, in addition to the replacement policy, we can also use different **insertion policies**. The insertion policy indicates how to allocate the newly accessed block in the rank list. There are typically two choices. LRU-MIP, which we usually simply call LRU, inserts the new block as the most recently accessed, i.e., the highest rank ("I believe this new block would be accessed again soon"). LRU-LIP, on the other hand, inserts the new block as the least recently accessed, i.e., the lowest rank ("I believe this new block is not important, unless there is a second access soon, which puts it to the most recently accessed"). Think of when this LRU-LIP policy will be better than LRU. 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 the basic LRU.

You can refer to the Adaptive Insertion Policy paper, especially Section 4, for more information about cache replacement policies with adaptive insertion policies including LRU-LIP.

3.2. DIP (Dynamic Insertion Policy)

For some applications, LRU-LIP performs better than LRU(-MIP), while for others, LRU is better than LRU-LIP. Therefore we want a mechanism that can dynamically and adaptively choose the insertion policy that has the fewest misses for the application. There is one such mechanism called DIP that dynamically estimates the numbers of misses incurred by the two insertion policies and selects the one with fewer misses.

A straightforward method of implementing DIP is to implement both LRU and LRU-LIP in two extra tag directories (a.k.a., tag arrays) which only keep track of hit/miss statistics but not the actual data, and

determine which of the two is better. But this implementation requires substantial hardware overheads of two extra tag directories.

A better way with low hardware overheads is called **set dueling**. Set dueling uses the tags of a few existing sets (so no additional directories) to monitor the performance of the two policies. More specifically, we use two small groups of sets in the cache, each fixed to use one of the two policies, and let them *duel*. The remaining sets will use the winner policy.

In this task, you are required to **implement the DIP policy with set dueling**. You need to keep 2 dedicated cache sets using LRU and LRU-LIP, respectively, and make all other sets use the policy that has the better performance. Specifically, dedicate the set with set ID 0 and the set with set ID 1 to LRU and LRU-LIP, respectively. Monitor the miss rates within the dedicated sets, and choose the policy with the lower miss rate to be the policy for the remaining sets. Reset the monitoring states every 100K memory accesses. It is even better to tune the policy with your own parameters, for performance reasons, e.g., use 32 or 64 dedicated sets for monitoring, or storing the number of misses as metric, as the original paper says. No matter what your final policy parameters are, please write them down in your write-up, and show the corresponding hardware overhead, i.e., the number of bits you use for monitoring. You may want to maintain the performance of the two sets in the function `updateCacheSetReplFields`, since this function has full visibility of hits/misses and set IDs.

You can refer to the Adaptive Insertion Policy paper , especially Section 5.2, for more details of set dueling. Notice that in the paper, the set dueling is designed for choosing between LRU-LIP and BIP. In our lab, only choosing between LRU and LRU-LIP is enough.

3. Implement Skewed-Associative Cache

As the size of one set increases, we need to do more tag comparisons when searching for a block inside a set, and also use more metadata spaces (e.g., the length of the LRU counters). Therefore we cannot increase the associativity too much for better performance. However, if the program shows different utilizations on different sets (for example, a pathological example always accesses high addresses in the stack and low addresses in the heap, with few accesses to the sets of middle addresses), we can increase the cache utilization by allowing associative blocks being in different sets.

Normally our set-associative cache uses the same set for an address, and only chooses between different ways, i.e., `(idx, way1)` and `(idx, way2)`. In skewed-associative caches, we apply different hash functions `H1, ..., Hp` on each way, 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.

For example, suppose we are using a 2-way skewed-associative cache and the number of sets is `Nset`. Then we need 2 hash functions, e.g., `address % Nset` and `(address + 4) % Nset`. The possible locations of the block with `address` are the way 1 of set `address % Nset`, and the way 2 of set `(address + 4) % Nset`. Using the LRU policy, we will choose the one to evict from these two locations using their LRU values. If we hit in a location, e.g., the way 1 of set `address % Nset`, the LRU counter of this way will be set to 0, and the LRU counters of all other ways in this set will be updated. The other sets are not impacted under hits to this set.

The key to skewed-associative caches is a good design of hash functions. A set of good hash functions can decrease the conflicts of different addresses and increase cache utilization. Think about what hash function

combination would lead to higher cache utilization.

In this task, you are required to implement **skewed-associative cache using LRU replacement policy**. Select **skewed-associative** in the memory tab of Ripes to test the performance of skewed-associativity cache.

Implementation

In `cachesim.cpp`:

Implement the function `analyzeCacheAccessSkewedCache` to realize the skewed-associative cache. This function is used to compute the set ID and way ID of a cache access. You need to extract the correct sets using the hash functions you choose and the corresponding ways. Then, you need to check whether the block is in these locations. If so, it is a cache hit and does not need eviction. If not, you need to decide one cache way to evict. You only need to implement **LRU policy** for comparing ways from different sets.

The function `analyzeCacheAccess` does the same thing only without support of skewed-associative caches. You can refer to it to help understand what to do.

Refer to the skewed-associative cache paper for more information about skewed-associative cache, including the choice of hash functions.

4. Learn the Performance Gap

Ripes config

To run the benmarks we've provided, please remove `--nostdlib` from linker arugments, and set the optimization level to `-O0` in the toolchain setting.

4.1 Benchmark Introduction

We have provided you two benchmarks to test the performance of your cache scheme.

- Benchmark 1 implements the Page Rank algorithm with 10 vertices and 50 edges. Notice that benchmark 1 will read values from `benchmark1_data.txt` in the `testcases/` folder. **Remember to modify the path of the input file to your own in `benchmark1.c` in ABSOLUTE PATH.**
- Benchmark 2 is an iterative program that computes the value of π. It is also the ranpi benchmark in Lab 2. It takes serveral minutes to execute in Ripes.

4.2 Evaluation of Different Cache Schemes

Replacement Policies

Use a 16-entry, 4-way associative cache, write-back, write-allocate, non-skewed for the rest configurations.

Design	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
No Cache	100.00%	100.00%		

Design	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
Random Replacement				
LRU Replacement				
PLRU Replacement				
LRU-LIP Replacement				
DIP Replacement				

Cache Associativity and Capacity

Use LRU, write-back, write-allocate, non-skewed for the rest configurations.

Associativity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
32-entry direct-mapped				
32-entry 8-way associative				
32-entry fully-associative				

Capacity	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way associative				
32-entry 8-way associative				
64-entry 8-way associative				

Skewed Associative

Use LRU, write-back, write-allocate for the remaining configuration options.

Capacity and Mapping	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles

Capacity and Mapping	Benchmark 1 Miss Rate	Benchmark 2 Miss Rate	Benchmark 1 Total Cycles	Benchmark 2 Total Cycles
16-entry 8-way associative				
16-entry 8-way skewed-associative				
32-entry 8-way associative				
32-entry 8-way skewed-associative				

4.3 Design Benchmarks

As you can see in the memory tab, Ripes supports us to use different cache configurations. As we have learned in the lectures, different configurations fit different programs and scenarios. In this part, you are required to write benchmarks in C to demonstrate the effects of the following configurations. For each category, implement one benchmark for each choice that makes the benchmark have the best performance over the others. Performance is measured in the number of cycles.

We define the default configuration as: 32-entry, LRU, 4-way set associative, write-through, write-allocate, non-skewed-associative. When comparing one configuration choice, other configurations should be set to default .The configurations are set in the memory tab of Ripes.

- Replacement policy: LRU vs. LRU-LIP (You only need to compare these two in this part. But you can also test other policies if you like). The benchmark file names should be `bench_lru.c`, `bench_lrulip.c`. Also, test the result of DIP on the two testcases and analyze how good is the dynamic reconfiguration.

Replacement Policy	bench_lru Miss Rate	bench_lrulip Miss Rate	bench_lru Total Cycles	bench_lrulip Total Cycles
LRU				
LRU-LIP				
DIP				

- Write hit policy: write-through vs. write-back. The benchmark file names should be `bench_writehit_through.c`, `bench_writehit_back.c`.
    - We use *bench_through* to represent `bench_writehit_through`, *bench_back* to represent `bench_writehit_back` in the table below.

Write Hit Policy	bench_through Miss Rate	bench_back Miss Rate	bench_through Total Cycles	bench_back Total Cycles
Write-back				

Write Hit Policy	bench_through Miss Rate	bench_back Miss Rate	bench_through Total Cycles	bench_back Total Cycles
Write-through				

- Write miss policy: write-allocate vs. no-write-allocate. The benchmark file names should be `bench_writemiss_allocate.c`, `bench_writemiss_noallocate.c`.
    - We use *bench_allocate* to represent `bench_writemiss_allocate`, *bench_noallocate* to represent `bench_writemiss_noallocate` in the table below.

Write Miss Policy	bench_allocate Miss Rate	bench_noallocate Miss Rate	bench_allocate Total Cycles	bench_noallocate Total Cycles
Write-allocate				
No Write-allocate				

If you think one of the choices **would always win**, describe why and discuss about what advantages other than performance that other choices may have. Or maybe you think the choice **does not affect miss rate and total cycles**, but will make a difference on other aspects. Write down your thoughts and explain the reasons. Here are some other aspects you can take into consideration: cache size, implementation cost, energy, number of writebacks, etc.

Record your results in the tables above. Also, You need to explain how you design the benchmarks. Discuss generally what kinds of programs prefer each choice and explain the reasons.

5. (Optional) Bonus Essay

When we retrospect our three labs, it was a long journey of blood, sweat, tears, and joy (maybe). Eventually we have built a relatively complete RISC-V processor simulator with its own pipeline and cache modules. Hopefully you now understand the main ideas of microarchitecture, flows of system design, and tips for building and debugging a software project step by step. At this point, we think it is beneficial to consider the following question: We learned basic techniques about **code optimizations** (compiling techniques) in Lab 1, and **microarchitecture optimizations** in Labs 2 and 3 (branch prediction, high-performance cache designs, etc.). Although we did not cover algorithms in the labs, you must have been playing with **various algorithms** in other CS courses. How do you like each of these three classes of performance-improving approaches? Which approach do you think works the best and is the most effective one? This is an open question with no standard answers. We encourage you to have and show your own opinions.

Also, if you like, you can write down your feelings and thoughts about our labs. Which part looks interesting and which not? Which part should be improved in the next year? Which part is unfair? These thoughts would not have any positive or negative impact on your grades, but the next-year students would greatly benefit from your important feedback.

Please write over 300 words to get points for this part.

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 implement the PLRU, LRU-LIP, and DIP policies? What are the respecitve hardware cost of the policies?
- How did you implement the skewed-associative cache? How did you choose the hash function combination in your skewed-associative cache implementation?
- Show your debugging process if there is anything interesting during the lab.

Results

- Analyze and explain the results in Section [4.2](#4.2 Evaluation of Different Cache Schemes). What conclusions can you make?
- How did you design the benchmarks in Section [4.3](#4.3 Design Benchmarks)? Analyze and explain the performance of the benchmarks.

Question Answering

- By extracting useful patterns from large amounts of data, deep learning has produced dramatic breakthroughs in many areas. It is therefore natural to wonder if deep learning could help design better cache replacement policies since the replacement is also a prediction problem. Do you think deep learning, or other machine learning techniques, can be applied to cache replacement? What are the advantages and possible difficulties? If you are interested in this problem, you can further refer to this paper.

- In the lectures we discussed another optimization called non-blocking cache. Do you think it is useful in your processor? In your opinions how much is the cost and benefit?

- In the lectures we also discussed an optimization called software prefetching. For example, the RISC-V ISA can be extended to add a new `prefetch offset(rs1)` instruction. This instruction fetches the target block to the cache, but does not fill in any register yet. Therefore the actual load instruction executed later can be a cache hit. Obviously the compiler should insert this instruction at proper locations in the program, e.g., several instructions ahead of the actual loads to have enough time to prefetch. Do you think it is useful in your processor? In your opinions how much is the cost and benefit?

Submission

Create a folder with the name of `lab3-<student ID>-<first name>-<last name>`, e.g., `lab3-2020123456-xiaoming-wang`. Put the following files in it. Compress the folder as a `.zip` package and upload it to `learn.tsinghua.edu.cn` (网络学堂) by **Dec 23 (Friday)**. You may submit multiple times, and we will grade based on the latest submission.

```
+-- lab3-design-document.pdf --- Your Lab 3 design document
+-- cachesim/ --- Your implementation
+-- designed_benchmarks/ --- Your benchmarks
        +-- replacement_benchmarks/
```

```
        +-- writehit_benchmarks/
        +-- writemiss_benchmarks/
+-- bonus_essay.pdf --- Your bonus essay (if your write one)
```

Grading policy

Plagiarism is **strictly** forbidden in this lab. Contact the TAs if you are in trouble.

- 15% - Correct Implementation of PLRU policy
- 10% - Correct Implementation of LRU-LIP policy
- 15% - Correct Implementation of DIP policy
- 10% - Correct Implementation of skewed-associative cache
- 20% - Correct implementation of benchmarks to show effect of different cache configurations
- 10% - Design document: methodology and result
- 10% - Design document: design considerations of benchmarks.
- 10% - Design document: question answering
- extra 5% - Bonus essay.

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

Correctness

As for your implementations of replacement policies, you should reach at least **30% hit rates** on the provided testcases to get full scores.