CS/ECE 752
Fall 2024
Homework 5 SOLUTION
Due 11 AM Central Time on Saturday, October 26th, 2024

NAME: ______________________________

You should do this assignment on your own (i.e., including separate from Chegg, CourseHero, ChatGPT, Gemmini, or any other similar tools), although you are encouraged to talk with classmates electronically or on Piazza about any issues you may have encountered. The standard late assignment policy applies: you may submit up to 1 day late with a 10% penalty.

What to Hand In

To submit your assignment:

1. Type up your answers to the following questions and submit **one PDF named HW5-<netID>.pdf** on Canvas. If you prefer writing your answers by hand, that is fine, but please scan your solutions and submit them on Canvas. **Please make sure to put your name in the above prompt of the first page of the PDF.** *If at all possible, please retain the format of the provided PDF/Word document. I am going to grade this homework using Gradescope, which relies on answers being in set places.*

2. Moreover, you should also create and turn in an archive (.zip, .gz., or .tgz) with the following files:

- For each of the 6 policies:
    - The output log that contains the hit, miss, and access information, with the appropriate replacement policy name appended to it:
        - `log-lfu.txt`
        - `log-lru.txt`
        - `log-plru.txt`
        - `log-sc.txt`
        - `log-srrip.txt`
        - `log-brrip.txt`
    - The stats file (normally stats.txt) for each replacement policy, with the appropriate replacement policy name appended to it:
        - `stats-lfu.txt`
        - `stats-lru.txt`
        - `stats-plru.txt`
        - `stats-sc.txt`
        - `stats-srrip.txt`
        - `stats-brrip.txt`

    - The trace you create for the access pattern for each replacement policy, with the appropriate replacement policy name appended to it:
        - `trace-lfu.py`
        - `trace-lru.py`
        - `trace-plru.py`
        - `trace-sc.py`
        - `trace-srrip.py`
        - `trace-brrip.py`
- The completed outer Python files needed to run the traces:
    - `run_replacement_policy.py`
    - `test_replacement_policy_hw4.py`
    - `cache_hierarchies.py`

Total Points: 45 (21 points for files (1 point each), 24 points for report – 6 points per policy)

In this assignment we will be building directly off of HW4 – where we examined how different cache replacement policies behave on relatively small, simple traces. However, this time we are going to learn how to use gem5 to test the exact same traces you did by hand last week!

Assumptions

The assumptions are the same as last week, although you will need to update the scripts (discussed below) to model the system accordingly.

- We have a single level of cache, which is 512B, has 64B blocks, and is 4-way set associative.
- You can assume the cache is PIPT and the translation has already been done for you (Note: we will not discuss virtual vs. physical addressing before this assignment is due – essentially PIPT means you can assume the provided addresses (below) are the addresses the cache uses for accesses).
    - In gem5 the method we will use for specifying addresses is going to be agnostic to this.
- All addresses are 64-bits.
- All of the requests are loads.
- There are no other cores, caches, etc. in the system, so these are the only memory accesses (i.e., no interleaved memory requests from another core).
- We are using standard hexadecimal/binary notation. So 0xABCD = $(1010\ 1011\ 1100\ 1101)_2$
- The data values you load are not important.

Access Pattern

For all cache replacement policies, you will create the following sequence of accesses (you can assume all bits that are not shown are 0):

```
    LD 0x0    // Access #0
    LD 0x81   // Access #1
    LD 0x100  // Access #2
    LD 0x188  // Access #3
    LD 0x4    // Access #4
    LD 0x200  // Access #5
    LD 0x18C  // Access #6
    LD 0x108  // Access #7
    LD 0x380  // Access #8
    LD 0x8    // Access #9
```

Running Access Pattern in gem5

Given the above information and access pattern, create a **trace** that will be passed into gem5's **traffic generator** to simulate the LFU, LRU, PseudoLRU (Tree-PLRU), SecondChance, SRRIP, and BRRIP replacement policies' behavior in gem5 for the above access pattern. To do this, we'll need several pieces:

- Traffic Generator
- Trace to Input to the Traffic Generator
- Configuring the Cache
- How to Run in gem5
- How to Update gem5 to get victim information
- Output Information (Log)

In the following paragraphs we explain how to use each of these pieces.

Traffic Generator

Modeling an entire system with a CPU, caches, main memory, network, etc. just to model the above minor cache access pattern is possible, but would be extremely complex and wasteful. Luckily, gem5 has support for a special type of configuration: a traffic generator. At a high level, the traffic generator works by modeling a system (essentially) with just a traffic generator, the cache(s), and a backing main memory. Essentially, this means our system is extremely simple and we can just pass accesses directly from the traffic generator into the caches to study their behavior – exactly what we want for this assignment! Some basic information about gem5's traffic generator is available here, if you want to read more about it.

Trace to Input to the Traffic Generator

In order to generate a sequence of memory accesses to send directly to the cache, the traffic generator requires some sort of input stimuli. Although there are multiple ways to design traffic generators, in this assignment we'll focus on one specific format where we specify a sequence of **linear** memory accesses. This input **trace** is written in Python and consists of a number of parts including lines that specify the addresses to access and when to access them, the trace must also specify some additional information such as what replacement policy it should be using, when the trace should exit/complete, a line in the trace to synchronize all accesses, and where the accesses should come from. In gem5, all trace accesses should be done inside a `python_generator` generator function.

Each line of cache accesses in this trace looks as follows:

```
      yield generator.createLinear(duration, startAddr, endAddr, accessSize,
minPeriod, maxPeriod, percentReads, dataLimit)
```

Where:

- `duration`: how long the memory access should take (*in ticks*). I recommend picking a consistent number (e.g., 60000) for all of your accesses.
- `startAddr`: the base address your request starts with
- `endAddr`: the end address your request ends with. If end – base spans multiple cache lines, this will generate multiple requests.
- `accessSize`: the size of the access in bytes
- `minPeriod`: if you want the request to repeat over a period of time, this specifies when that period should start. I recommend you set minPeriod = maxPeriod for this assignment, but make them a non-0 value.
- `maxPeriod`: if you want the request to repeat over a period of time, this specifies when that period should end. I recommend you set minPeriod = maxPeriod for this assignment, but make them a non-0 value.
- `percentReads`: an integer value [0,100], where 0 means all stores, and 100 means all reads.
- `dataLimit`: the data limit (*in bytes*). This can essentially be ignored for this assignment (i.e., set to 0).

An example of this (for the FIFO replacement policy) is:

```
      # tell gem5 what replacement policy this trace is using
      from m5.objects.ReplacementPolicies import FIFORP as rp

      # define the traffic generator pattern
      def python_generator(generator):
          # All generated linear memory accesses go here (each access)
          # happens after the previous one
          yield generator.createLinear(60000, 0, 63, 64, 30000, 30000, 100, 0)
          yield generator.createLinear(60000, 128, 191, 64, 30000, 30000, 100,
      0)
          …
          # After all memory accesses, synchronize all accesses with this access
          yield generator.createLinear(30000, 0, 0, 0, 30000, 30000, 100, 0)
```

```
        # Tell the traffic generator to exit
        yield generator.createExit(0)
```

Configuring the Cache

Now that we have written our trace to access the various memory locations in the specified pattern, we need to setup the surrounding scripts to configure gem5 in the desired manner to run our tests. For all of these, we'll take advantage of gem5's integration for the traffic generator and replacement policy tests with its test library (testlib) – this minimizes the amount of boilerplate code we need to develop. To start with, in your clone of gem5 you should access `cache_hierarchies.py` (specifically this line within that file: https://github.com/gem5/gem5/blob/stable/tests/gem5/replacement_policies/configs/cache_hierarchies.py#L41). What a coincidence, the size and associativity are already set to what you need! But if you wanted to model a system with a different cache size or associativity, this is where you'd change it. Also, note that the Ruby memory subsystem in gem5 (which we are using for this assignment) assumes 64B cache line sizes by default – this is relatively hardcoded into Ruby. Moreover, you may have noticed that is using the "MIExample" coherence protocol – for this assignment you can ignore this (we'll discuss coherence later in the semester). However, for future reference note that this assignment has only been tested with Ruby's MI_Example coherence protocol – even though Ruby has many other coherence protocols, none of them have been tested with this traffic generator assignment.

How to Run Traffic Generator in gem5

Finally, we need to run our traces in gem5. To do this, there are a few steps:

1. First we need to update the testlib to run your traces. Download *test_replacement_policies_hw4.py* (I have posted this file on Canvas, as well as at */u/s/i/sinclair/public/html-s/courses/cs752/fall2024/handouts/hw/hw4/test_replacement_policies_hw4.py* if you are logged into a CSL machine. You will need to **place this file in *$GEM5_HEAD/tests/gem5/replacement_policies/*** in order for things to work. **Leave everything in this file as is except for the "traces" array** (which starts on line 64). You will need to update the traces array to include the list of all traces you want the testlib to run with your traffic generator.
    a. Note that the traces array assumes **relative** paths, and the traces should be comma separated and use quotes (e.g., `"hw4/trace1.py", "hw4/trace2.py"`).
2. The `test_replacement_policies_hw4.py` script needs a wrapper script to run the tests, which already exists: https://github.com/gem5/gem5/blob/stable/tests/gem5/replacement_policies/run_replacement_policy.py. As the name implies, the script is responsible for running our traffic generator tests. It handles setting up the system with the traffic generator, the cache, and main memory. **You should not need to change it.**
3. Next we need to compile gem5. Specifically, we need to compile gem5 for the MI_Example protocol mentioned above. To do this:
    a. `cd $GEM5_HEAD`

   b. `python3 `which scons` build/X86_MI_example/gem5.opt -j9`
4. Next we need to build and run the tests in the testlib:
   a. `cd $GEM5_HEAD/tests/`
   b. `./main.py run -vvv -j9 --length=very-long -t 240 gem5/replacement-policies`
      i. **Note**: this might take a long time the first time
      ii. **After the first time**: to save time (avoid rebuilding gem5) you should run this instead: `./main.py run -j9 -vvv --length=very-long --skip-build -t 240 gem5/replacement-policies`

How to Update gem5 to get victim information (optional, but highly useful)

When trying to understand which entries are getting evicted from a given replacement policy in gem5, it is often useful to instrument gem5 to print additional information about the options and what is being evicted. Fortunately, all gem5 replacement policies use the same format – their `getVictim`() function is responsible for deciding which way to evict. Unfortunately, by default these policies do not print anything out by default.

Thus, below I'm including an example of how to update the FIFO replacement policies' getVictim function to print out information about which victim's it is considering ("Candidate …") and which victim it ultimately picks. Both of these added prints are highlighted in blue. While optional, I encourage you to add similar prints to gem5 for other replacement policies to get more information about what is being considered/evicted.

```
ReplaceableEntry*
FIFO::getVictim(const ReplacementCandidates& candidates) const
{
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);

    // Visit all candidates to find victim
    ReplaceableEntry* victim = candidates[0];
    for (const auto& candidate : candidates) {
        // Update victim entry if necessary
        if (std::static_pointer_cast<FIFOReplData>(
                    candidate->replacementData)->tickInserted <
                std::static_pointer_cast<FIFOReplData>(
                    victim->replacementData)->tickInserted) {
            printf("Candidate TickInsert: %ld, Victim TickInsert: %ld\n",
std::static_pointer_cast<FIFOReplData>(candidate->replacementData)-
>tickInserted,             std::static_pointer_cast<FIFOReplData>(victim-
>replacementData)->tickInserted);
            victim = candidate;
        }
    }
    printf("replacement: %s\n", victim->print().c_str());
    return victim;
}
```

All replacement policies you'll use are located in this folder: https://github.com/gem5/gem5/tree/stable/src/mem/cache/replacement_policies. Note that the changes you might need to make to add these prints to a different replacement policy may be slightly different from the changes to the FIFO replacement policy.

Output Information (Log)

If you run with the above setup, you will get output (e.g., to stdout) that looks like this:

```
…
Test:  test-replacement-policy-traces/<traceName>-NULL-x86_64-opt-
MI_example Passed
…
```

(**if it says your trace fails, please post on Piazza**)

All of these your trace tests will write themselves to `$GEM5_HEAD/tests/gem5/testing-results/SuiteUID\:test-replacement-policy-traces/`.

Note: your path for the output files may differ slightly depending on the name of the folder you use for the traces.

Follow the path specified by the output and you'll see the following contents in the log file (`simout`):

```
…
Global frequency set at 1000000000000 ticks per second
Beginning simulation!
  74000:      system.cache_hierarchy.ruby_system.controllers.sequencer:
Cache miss at [0x0, line 0x0]
 134000:      system.cache_hierarchy.ruby_system.controllers.sequencer:
Cache miss at [0x80, line 0x80]
…
```

You should capture this log file as specified above in the hand-in information. Note how the addresses here match with the above sample for the FIFO trace. For the address in the brackets, the format is: [*wordAddress*, line *lineAddress*]. Moreover, note that the prints here tell you if an access hits or misses. Finally, if you check the resultant `stats.txt` file, you should also see the cache hits and misses there match the prints you see in the above output in `simout`.

What You Need to Do in gem5

Given all of this, you should write the trace(s) to pass the aforementioned memory access pattern into the above ecosystem for each replacement policy (LFU, LRU, TreePLRU, SecondChance, SRRIP, and BRRIP), then compare the results of gem5 (e.g., hits and misses) to those from HW4. For each policy answer the following questions in your report:

- How many hits and misses did the replacement policy get for this access pattern in HW4?
- How many hits and misses did the replacement policy get for this access pattern in gem5?

- If they don't match, where do they diverge?

Hint: for RRIP, gem5 uses a single implementation of SRRIP and BRRIP. You will need to change the threshold that is passed into this class to properly model BRRIP per the HW4 specifications.

Solution:

For hits/misses from HW4, see HW4 solution/explanation

LFU:

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  4 hits, 6 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  4 hits, 6 misses

- If they don't match, where do they diverge and why?

  The hits and misses in gem5 line up exactly with the hits and misses in HW4 (i.e., each hit and miss matches in the same order)!

LRU:

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  3 hits, 7 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  3 hits, 7 misses

- If they don't match, where do they diverge and why?

  The hits and misses in gem5 line up exactly with the hits and misses in HW4 (i.e., each hit and miss matches in the same order)!

Pseudo (Tree) LRU (PLRU):

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  4 hits, 6 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  3 hits, 7 misses!

- If they don't match, where do they diverge and why?

The 8th access (0x108) misses in gem5 while it hits in HW4. This is because ultimately the algorithm in HW4 about which way to insert entries into when there is one or more invalid entries to pick from. HW4 specifies that we should pick these entries based on the Tree values. However, after adding debug prints to gem5's Tree PLRU implementation, we can see that it is updating the tree values, but picks the next monotonically increasing way to put an entry into (i.e., like all of the other replacement policies).

To get this information, I instrumented PLRU's touch() function (which is called by its reset() function every time a new entry is inserted – index 0 = root, index 1 = left subtree, index 2 = right subtree):

```
        Tree Reset
    Tree Updated: index: 1: 1
    Tree Updated: index: 0: 1
 74000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x0, line 0x0]
        Tree Reset
    Tree Updated: index: 1: 0
    Tree Updated: index: 0: 1
 134000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x81, line 0x80]
        Tree Reset
    Tree Updated: index: 2: 1
    Tree Updated: index: 0: 0
 194000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x100, line 0x100]
        Tree Reset
    Tree Updated: index: 2: 0
    Tree Updated: index: 0: 0
 254000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x188, line 0x180]
```

By inserting entries this way despite updating the tree, it has a knock-on effect with subsequent accesses, which ultimately leads to 0x100 being evicted sooner (during the 5th access to 0x200), causing an additional miss.

<u>**Second Chance (SC)**</u>:

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  3 hits, 7 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  3 hits, 7 misses

- If they don't match, where do they diverge and why?

  The hits and misses in gem5 line up exactly with the hits and misses in HW4 (i.e., each hit and miss matches in the same order)!

SRRIP:

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  3 hits, 7 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  3 hits, 7 misses

- If they don't match, where do they diverge and why?

  The hits and misses in gem5 line up exactly with the hits and misses in HW4 (i.e., each hit and miss matches in the same order)!

BRRIP:

Note: for this solution I set btp=67. If you set btp=66 instead, you may have seen slightly different results.

- How many hits and misses did the replacement policy get for this access pattern in HW4?

  2 hits, 8 misses

- How many hits and misses did the replacement policy get for this access pattern in gem5?

  3 hits, 7 misses!

- If they don't match, where do they diverge and why?

  Access 7 (0x18C) hits in gem5 but does not in HW4. The reason for this is that our algorithm in HW4 to approximate how often BRRIP should choose a long (2) vs. distant (3) re-reference interval upon insertion is not exactly the same as gem5. In gem5, BRRIP does this by picking a random number (https://github.com/gem5/gem5/blob/stable/src/mem/cache/replacement_policies/brrip_rp.cc#L87). This is not guaranteed to precisely match the behavior of our BRRIP algorithm in HW4 – where we knew which insertions were going to exceed the threshold and which were not. As a result, after adding additional debug prints into the functions in BRRIP, we can see that the insertions do not match. For example, the first 4 insertions (when there is at least 1 invalid entry) for me were:

  (to get these prints, I instrumented the reset() function in brrip_rp.cc)

```
        inserted RRPV: 3
 74000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x0, line 0x0]
        inserted RRPV: 2
134000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x81, line 0x80]
        inserted RRPV: 2
194000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x100, line 0x100]
```

```
            inserted RRPV: 2
 254000: system.cache_hierarchy.ruby_system.controllers.sequencer: Cache miss at [0x188, line 0x180]
```

In particular, the issue here is the 4th insertion (into way 3) – our HW3 algorithm assumed that this would be inserted with an RRPV of 3, but the random number generator in gem5 chose an RRPV of 2. This affects the subsequent hits and misses – ultimately causing access 7 to eventually hit because the better RRPV value means it is not evicted earlier during the access of 0x200 (way 0 is instead).

Note: because gem5 uses a random number generator instead, it is possible you got a different answer than this. As long as your answer explains something like the above, you will get credit.