CS341: Computer Architecture Lab

Lab 5: Hacking ChampSim
Report

Team ARCHimedes
Vibhav Aggarwal (190050128)
Adithya Bhaskar (190050005)
Devansh Jain (190100044)
Harshit Varma (190100055)
Harshit Gupta (190050048)

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2021-2022

Contents

Abstract

Summarize the objective of the lab, what experiments you have conducted, what were the results that you have obtained in a clear and concise manner. Numbers matter, not just words only, for ex. *very high*, *slow* etc.

1. Understanding the problem statement

1.0.1 Understanding the code

- The `handle_read` function is responsible for reading a query from the read queue and processing it.
  A read hit means that the line requested is present in the cache. So, the data is read from the cache itself. Following are the things that are done on a hit:
  - Get a read request from the read queue `RQ`. If no read request, return.
  - Get the set and the way corresponding to the request address. For a hit, there must be a way with same address as the requested one. This is done using the `get_set` and `check_hit` functions.
  - Read the data and update it in the packet that we got from the read queue.
  - If it is a load request, update the prefetcher using the prefetcher_operate functions
  - Update the replacement policy using the update_replacement_policy functions. aThis corresponds to bringing the accessed block to MRU position in the LRU scheme.
  - Mark the entry in the MSHR of higher level as completed and add the data to it. This is done using the `return_data` function for the correct upper level cache. Notice that for L1D, L1I, ITLB and DTLB do not have a higher level cache. Istead, they put the data in the PROCESSING queue.
  - Update the statistics like number of hits and number of accesses.
- A read miss means that the line requested is not present in the cache. In such a case, we do the following:(skipping over getting the query and detecting a miss as it is same as hit)
  - First we need to check if there is a read request for the same address already in MSHR or not. If there is we can merge the requests. This is done using the `check_mshr` function.
    - If the address is not in MSHR, we need to add a new entry in MSHR and read queue of the lower level. Adding to MSHR is done using `add_mshr` function. Adding to the lower level's read queue is done using `lower_level->add_rq` function. (This function also checks if the request can be served from the write queue or not and adds the request to read queue only if it can't be done)
      During the above operations, we need to ensure that we don't exceed the size of MSHR's and the queues. If this is the case, we must stall.

        - If the address is in MSHR, we need to merge the requests.
          There is a member set named `index_depend_on_me` (different sets for load, store and instruction) in teh `PACKET` class. This set stores the index of all the requests depending on it. Merging indices amounts to joining the correct sets from request to MSHR entry and insert the index of request to the MSHR entry.

    - Update statistics like number of misses, number of MSHR entries merged etc.

- The `handle_fill` function is responsible for processing the completed requests in MSHR and put the data in correct position in the cache. Any function serving a read request uses `return_data` which only adds the data to the MSHR entry for that address. But we need the data in cache and not in MSHR. The work of getting the data from MSHR and writing to cache is done by `handle_fill`. That is why it needs to read (completed) request from MSHR.

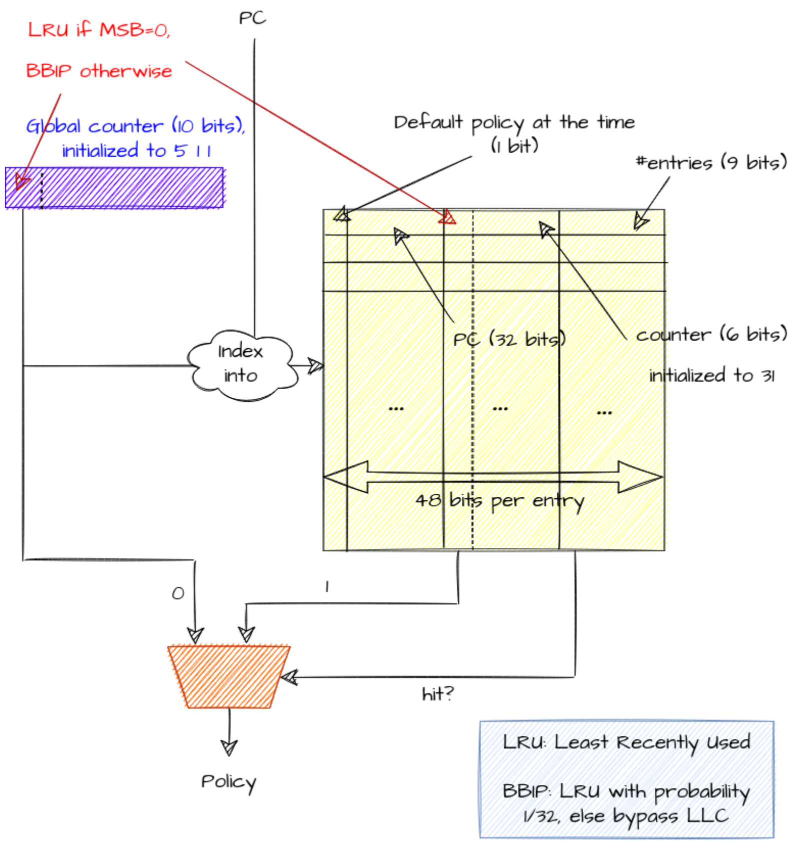
- The lower_level, upper_level data members are declared in teh MEMORY class. Since CACHE class inherits from MEMORY, it also has these data members. They are assigned in `main.cc` in the `main` function for all the caches.
  These are arrays which store pointer to the next lower or next higher cache in the memory hierarchy (if they do not exist, then set to NULL). Since we have different instruction and data cache at L1, we need 2 different upper_level cache array, one for data and one for instruction. So, for example for L2 cache, the values are:
  ooo_cpu[i].upper_level_icache = ooo_cpu[i].L1I
  ooo_cpu[i].upper_level_dcache = ooo_cpu[i].L1D
  ooo_cpu[i].lower_level = uncore.LLC
  Notice that L1D, L1I, L2 and the TLBs are CPU specific while the LLC and DRAM are shared.

- The `return_data` function is responsible for adding the data in an MSHR entry and marking it as completed. It takes a PACKET object as argument which stores the address and data. The basic workflow is:

    - Find the MSHR entry with the same address as the one mentioned in the given packet. This is done using `check_mshr` function. If it does not exist, return error
    - Mark the MSHR entry as completed(setting returned to COMPLETED) and add the data and metadata to the MSHR entry.
    - Update the metrics like number of requests returned and latency
    - Update `next_fill_index` using `update_fill_cycle` function. This index is the index from which `read_fill` reads the request to be written in the cache.

2. Implement LLC Bypassing

The relevant output files are in their respective directories, as required by the problem statement. We proceed to describe our implementation of MadCache. We begin with the description of the *predictor* structure. Refer to the illustration below.

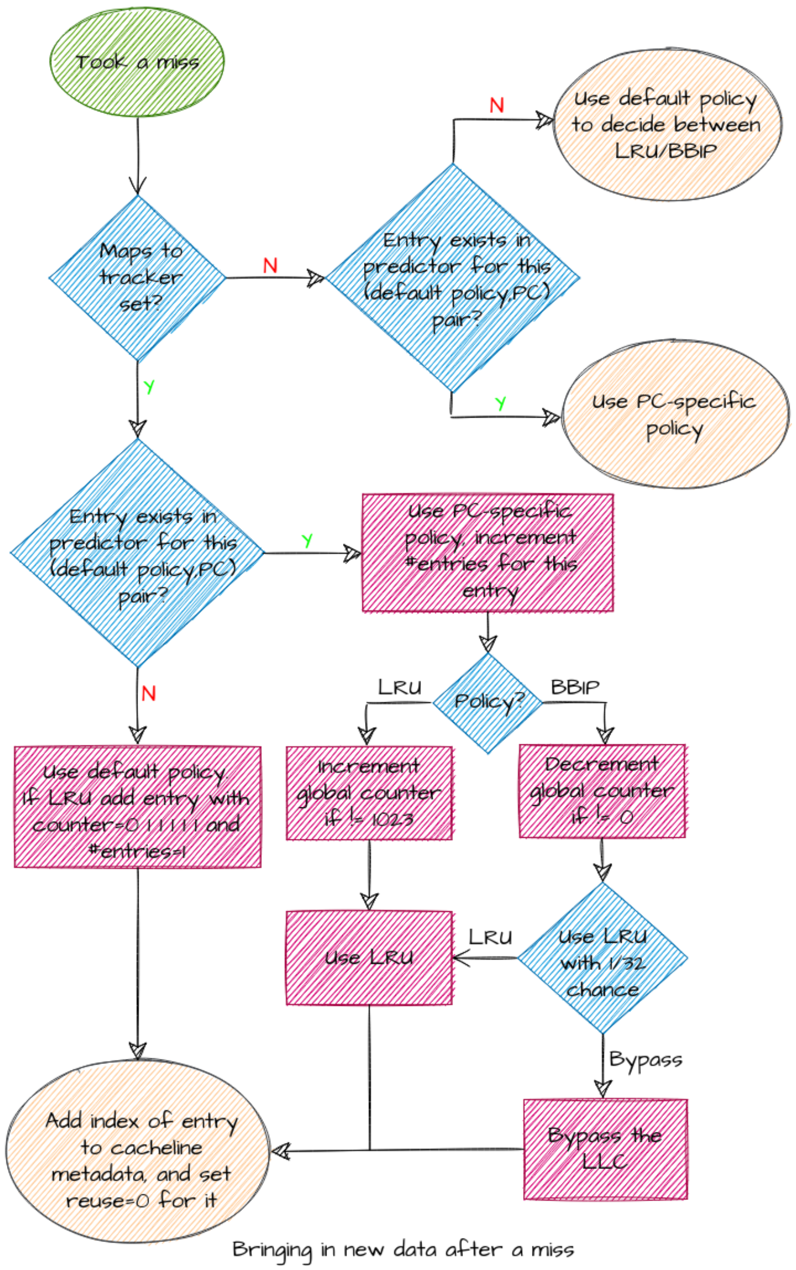


The structure of the predictor table

The predictor consists of a global 10-bit counter (which determines the global default policy, initialized to 0111111111 or 511) and a set of entries each with their own counters which determines PC-specific policies. In particular, if the MSB of the global counter is 0, the default policy is LRU; else it is to use BBIP. Here, BBIP uses LRU 1/32-th of the time, and bypasses otherwise. Each entry contains:

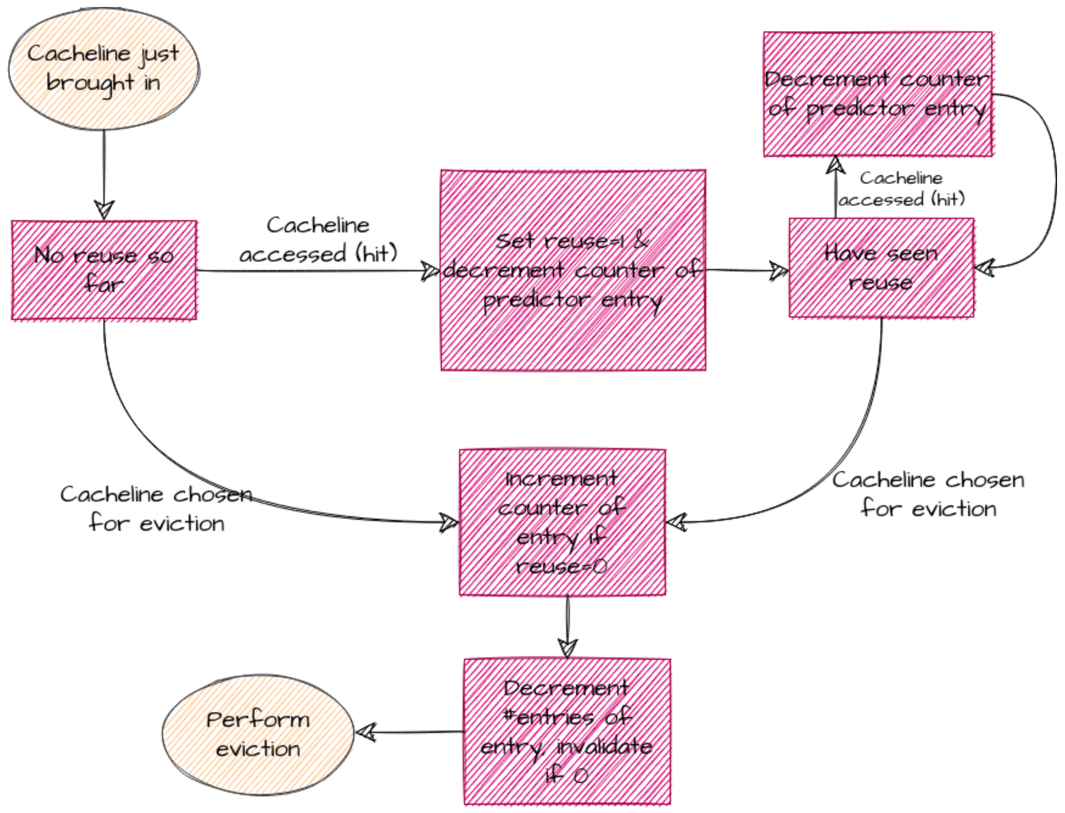
- **Default policy and PC:** This consists of the PC used when fetching the cacheline which led to the creation of this entry, and the default policy at the time.
- **Counter:** A 6-bit counter governing the PC-specific policy, of 6 bits.
- **Number of entries:** This field tells us how many cachelines currently use this entry (0 if none).

Next we describe how we use this predictor to decide what to do with newly fetched cachelines from DRAM to the LLC. Refer to the flowchart below.


Took a miss
Maps to tracker set?
N
Entry exists in predictor for this (default policy,PC) pair?
N
Use default policy to decide between LRU/BBIP
Y
Use PC-specific policy
Y
Entry exists in predictor for this (default policy,PC) pair?
Y
Use PC-specific policy, increment #entries for this entry
N
Policy?
LRU
BBIP
Use default policy if LRU add entry with counter=0 11111 and #entries=1
Increment global counter if != 1023
Decrement global counter if != 0
Use LRU
LRU
Use LRU with 1/32 chance
Bypass
Bypass the LLC
Add index of entry to cacheline metadata, and set reuse=0 for it

Bringing in new data after a miss

If the `(Policy,PC)` pair exists in the predictor, we use the custom policy to decide what to do. This occurs even for lines not in tracker sets (we map those cachelines whose index bits have 0101 as the 4 LSBs as tracker sets; i.e. 1/16 of all cachelines are tracker sets). However, for these non-tracker lines we do **not** create an entry in the predictor. However, if the cacheline *is* part of a tracker set, we check to see if an entry exists in the predictor. If so, we use the corresponding custom policy and increment its `#entries` field if it happens to be LRU (we shouldn't and don't do this for Bypass since there is no cacheline allocated in that case). Otherwise, we fall back to using the default policy, and try to allocate a free predictor entry for this cacheline. If this succeeds, we initialize it with a counter of 011111 ("just" LRU) and a `#entries` field of 1. Note that if the allocation fails, we treat this cacheline just as if it were a non-tracker line henceforth (until new data is brought into it again).
If the policy used is LRU, we initialize the cacheline by setting `reuse=0` and the `pc_pred_index` to point to the allocated predictor entry.
Further, whenever we see a miss that maps to a tracker set, we check if we have a PC-specific policy for this PC. If so, we correspondingly change the global counter to penalize that specific type of behaviour (since it caused us a miss). This implements set dueling (or "PC dueling", rather) between the LRU and Bypass sets. We next look at the lifetime of a tracker cacheline in the LLC. Refer to the flowchart below.


Cacheline just brought in
No reuse so far
Cacheline accessed (hit)
Set reuse=1 & decrement counter of predictor entry
Have seen reuse
Cacheline accessed (hit)
Decrement counter of predictor entry
Cacheline chosen for eviction
Increment counter of entry if reuse=0
Cacheline chosen for eviction
Decrement #entries of entry, invalidate if 0
Perform eviction

Life-cycle of a cacheline in a tracker set

A tracker cacheline starts out with a `reuse` bit of 0, and a counter initialized to 31 (if it was the first one for that entry). Whenever this cacheline sees a hit, it means that we were right to use LRU; thus we decrement the corresponding predictor entry's counter to make the behaviour more "LRU-like" for this PC. We also set `reuse=1` for this cacheline on a hit. On the other hand, when the cacheline is being evicted, if we see a reuse bit of 0, this means that the cacheline was never used again; we should have bypassed the LLC. Thus we increment the corresponding counter. In either case, we decrement the `#entries` field of the counter on eviction, and if it falls to 0, de-allocate the predictor entry (so that it can be used by future misses).

Modified files. We modified the following files to implement the above scheme:

- The files `inc/block.h` and `src/block.cc` contain the implementation of the predictor class.
- The file `inc/cache.h` has the definitions of a few helper functions and the modifications of the `CACHE` class to include the predictor.
- The file `src/cache.cc` contains the bulk of the changes, and implements the core logic of the above scheme. In particular, the functions that handle reads, writebacks and fills see extensive changes. Some other functions also see minor modifications.

Impact on IPC. The baseline LRU scheme gave an IPC of 0.142776 on average. On the other hand, our scheme gave an IPC of 0.171424 on average.
This corresponds to an increase (improvement) in IPC by roughly **20.065%**.

3. Prefetching

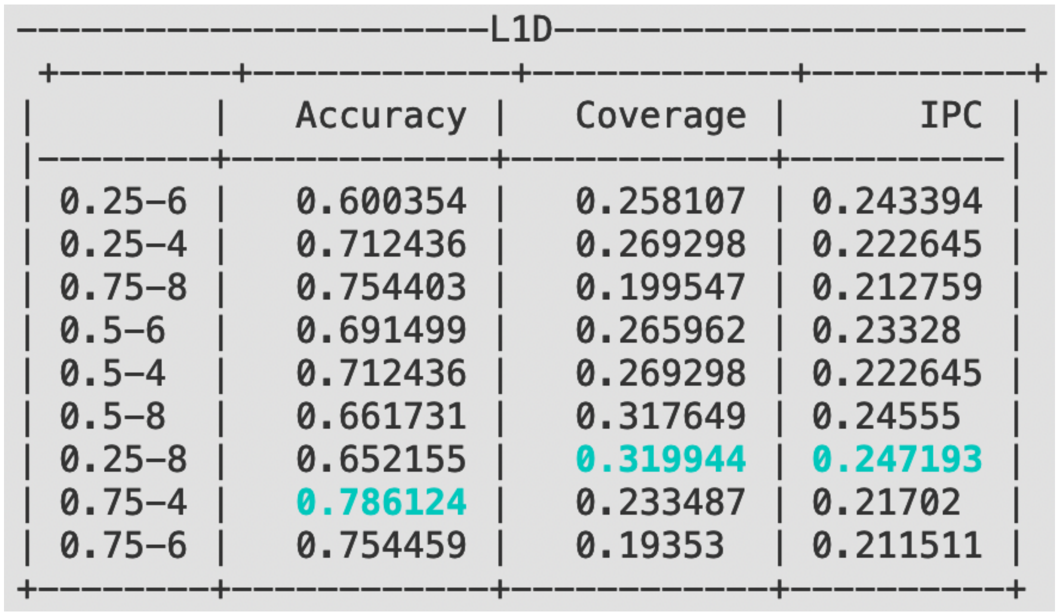
3.1 The Throttle Policy

<briefly describe the policy we used and the way we calculated accuracy and coverage>

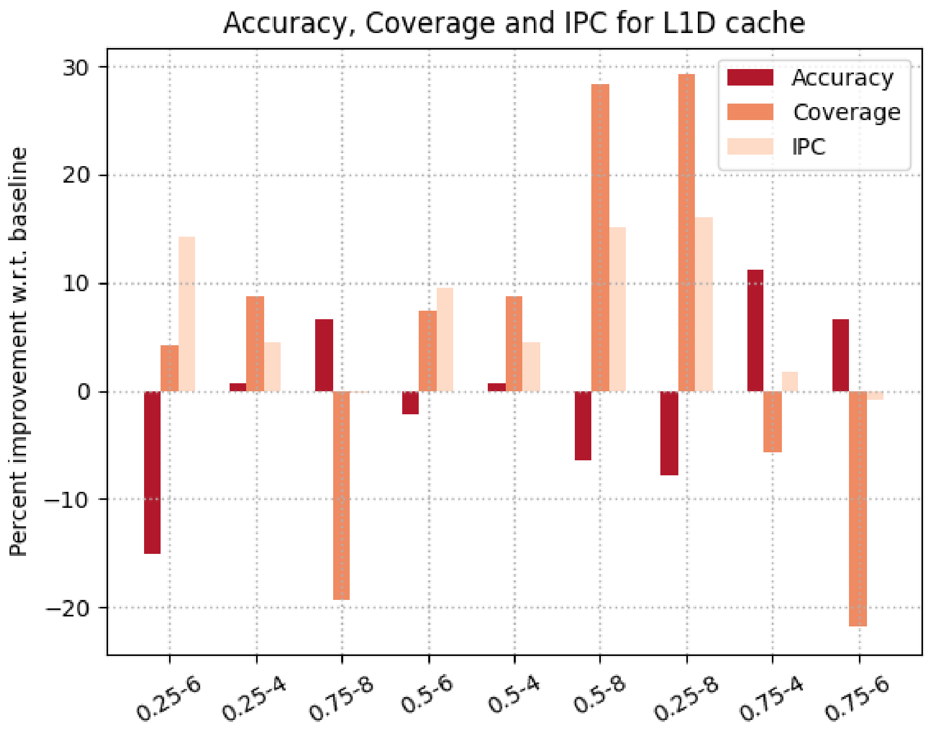
3.2 Results on different thresholds

3.2.1 L1D Prefetcher

Here, we used the baseline parameters for L2C prefetcher and varied the threshold and upper bound for L1D. In the following figures, `x-y` represents that the threshold was set to `x` and the upper bound was set to `y` for that partiular run.



```
----------------------L1D----------------------
 +--------+------------+------------+----------+
|         |   Accuracy |   Coverage |      IPC |
|---------+------------+------------+----------|
| 0.25-6 |   0.600354 |   0.258107 | 0.243394 |
| 0.25-4 |   0.712436 |   0.269298 | 0.222645 |
| 0.75-8 |   0.754403 |   0.199547 | 0.212759 |
| 0.5-6  |   0.691499 |   0.265962 | 0.23328  |
| 0.5-4  |   0.712436 |   0.269298 | 0.222645 |
| 0.5-8  |   0.661731 |   0.317649 | 0.24555  |
| 0.25-8 |   0.652155 |   0.319944 | 0.247193 |
| 0.75-4 |   0.786124 |   0.233487 | 0.21702  |
| 0.75-6 |   0.754459 |   0.19353  | 0.211511 |
+---------+------------+------------+----------+
```



3.2.2 L2C Prefetcher

Simialar to previous section, we varied the threshold and upper bound for L2C while keeping the baseline parameters for L1D.

3.3 Source files edited or added

Added: `prefetchers/ip_stride.l1d_pref`, `inc/ip_stride.h`

4. Contributions

Table 4.1: Contributions of each team member

Member	**Work Done**
Vibhav Aggarwal	
Adithya Bhaskar	
Devansh Jain	
Harshit Varma	
Harshit Gupta	