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

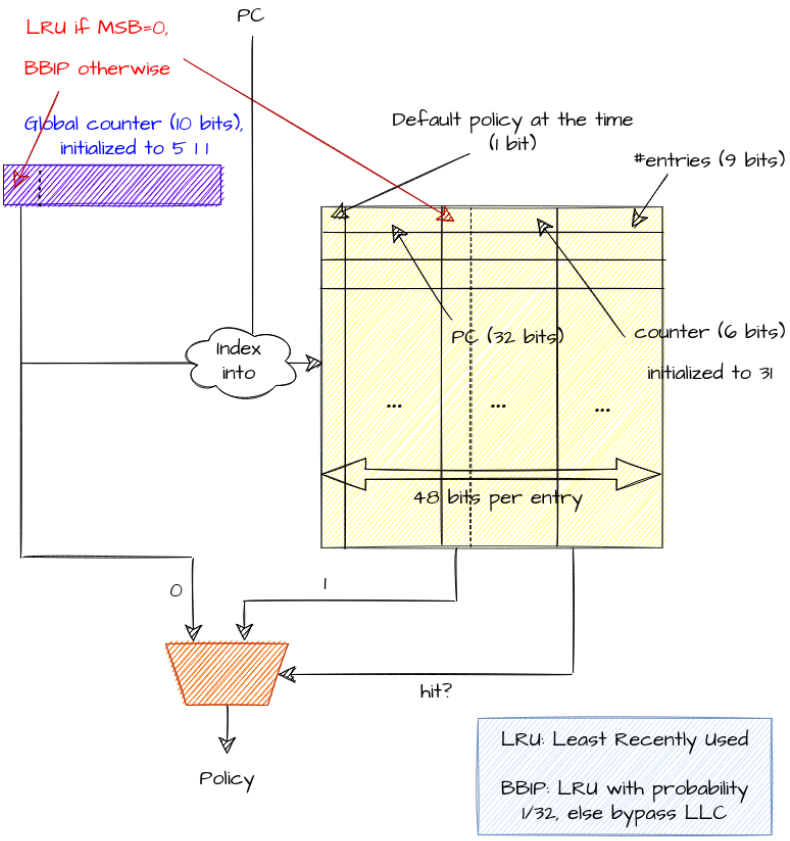
Abstract

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

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.

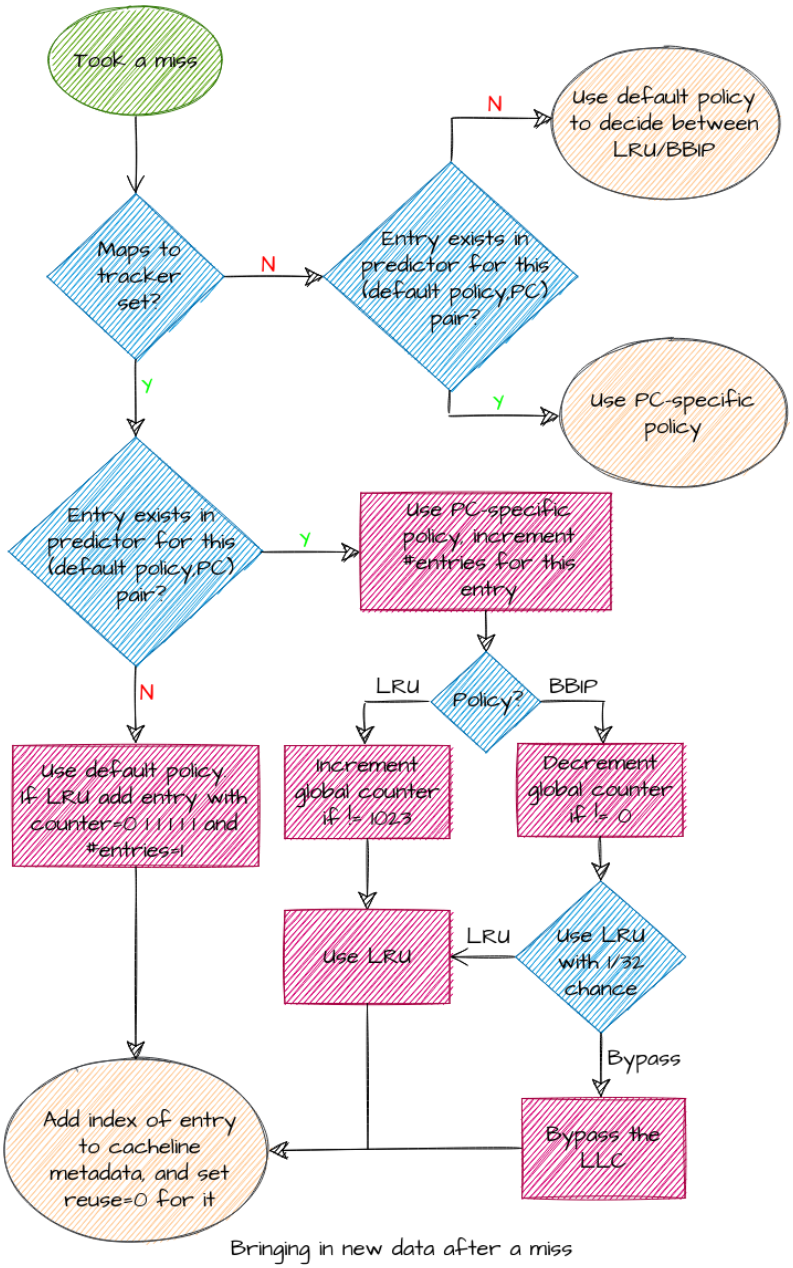


The structure of the predictor table

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:

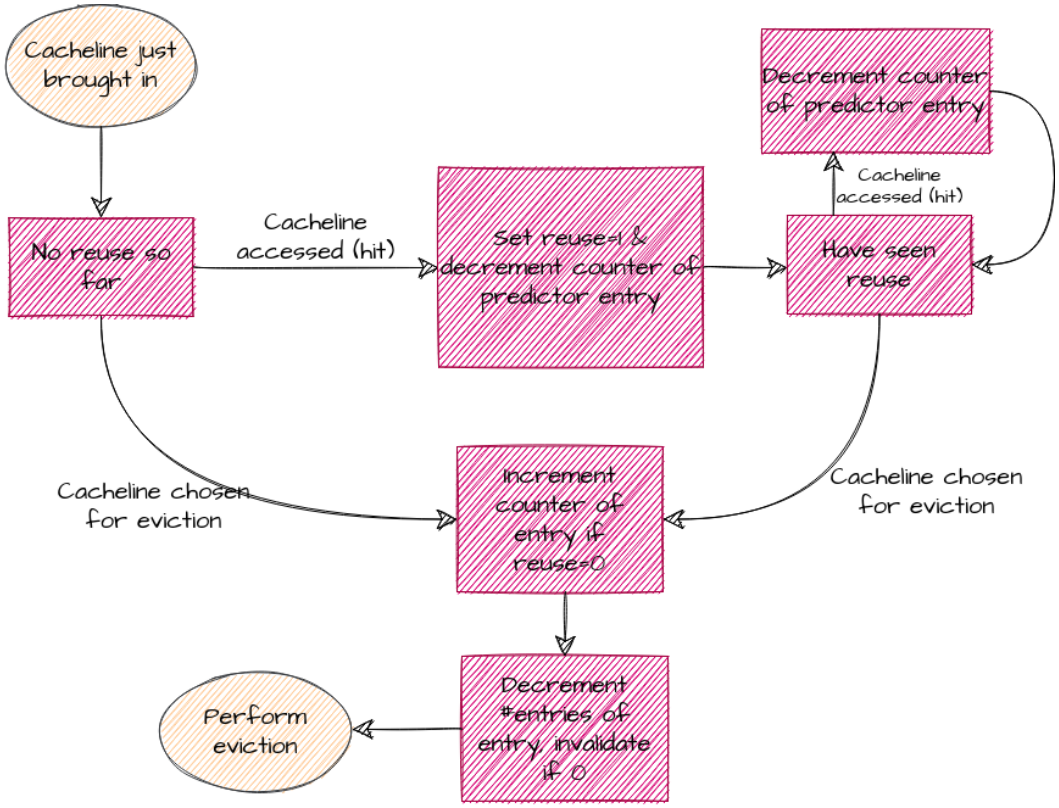
- **Default policy and PC:** This consists of the PC used when fetching the cacheline which led to the creation of this entry, and the default policy at the time.
- **Counter:** A 6-bit counter governing the PC-specific policy, of 6 bits.
- **Number of entries:** This field tells us how many cachelines 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.



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.



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 Normalized IPC improvement wrt thresholds

3.2 Prefetcher coverage wrt thresholds

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	