CS341: Computer Architecture Lab

Lab #5: Assignment 5
Report

Team Baguette
Akash Cherukuri (190050009)
Battepati Karthikeya (190050026)
Mastansha Rajulapudi (190050098)
Sudhansh Peddabomma (190050118)
Thivesh Chandra (190050124)

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2021-2022

Contents

Abstract

This lab was focused on understanding LLC bypassing and its implementation in the MadCache [1] paper. After a firm understanding was gained, the code of ChampSim was modified to implement the paper. There are various types of LLC Bypassing methods, but in this lab, we have implemented an Instruction Pointer/Program Counter-based mechanism inspired by MadCache. The modified code has been commented appropriately for better readability.

The first two sections describe the implemented model and the improvements obtained upon using the said model. IPC improvement obtained is $\sim 71\%$.

Next, prefetch throttling has been implemented for an IP-stride prefetcher. IP-stride prefetcher is an extended version of the simple stride prefetcher which handles stride patterns based on instruction pointers. It maintains a table of previous addresses accessed by a list of instruction pointers. When the same instruction is executed again, a stride is calculated for the address accessed and a prefetch request is made based on it. The throttling mechanism changes the prefetch degree dynamically based on the performance of this IP in previous encounters.

The final two sections tackle this problem. The implemented model has been explained, and the obtained results have been displayed in both a tabular and a graphical manner. Upto 20% improvement in IPC was observed.

Note that `#define TASK1` line in `cache.cc` has to be changed for running code for Task3 and vice-versa. This is to enable a single champsim folder submission.

0. Understanding the problem statement

0.1 MadCache Summary

MadCache is an adaptive cache insertion policy which uses a PC's memory access history and behavior to determine the best insertion policy. This method assumes that the program is likely to continue its behavior, be it streaming or exhibiting high locality. The method makes a choice between BBIP (ideal for streaming, avoids thrashing) and LRU (ideal with locality) policies as explained below.

PC-Predictor is used to track the history of PCs which have accessed the tracker sets. It maintains a 6-bit counter, with the MSB deciding whether the PC entry will bypass. The threshold bit being set indicates streaming behavior (BBIP) and unset indicates locality (LRU). The current default policy and PC are used to index into the table as the behaviour of the current PC would depend on the current default policy as well.

Tracker sets are a subset of the cache used to model the behavior of the entire cache. A PC accessing the tracker set is stored in the PC-Predictor, and its index and reuse bit are stored along with the cache line for tracking behavior. **Set dueling** amongst the tracker sets is used to determine the current default policy. Do note that a PC in the PC-predictor table overrides the default policy, as the former is deemed to be more representative of the PC's behavior.

Upon a miss for the L3 tracker set, an entry for the PC is made in the PR-predictor table with counter barely below the bypassing threshold. Further behavior of the PC directs the policy chosen in the future. If no space is present in the table, the PC follows the default cache policy. Rows are evicted implicitly when the number of cache blocks referenced by the PC become zero. We also only credit the PCs which brought a line into the cache, and not the accesses to the L3 cache line by other PCs.

Follower Sets comprise of the L3 cache not part of the tracker sets, and do not update the PC-Predictor table. Note that follower sets are larger than tracker sets. The policy followed by default is the one dictated by set dueling amongst the tracker sets, but this can be overridden if an entry for the PC exists in the PC-Predictor table.

Single threaded implementation of MadCache reveals that it mostly performs as good as, if not better than, LRU. Additionally, the hardware overhead required for multithreaded MadCache is less than 475Kbits. The authors remarked that gcc was the only benchmark with worse performance, and many other benchmarks exhibited a substantial improvement. The geometric mean across all benchmarks revealed a 2.5% improvement in IPC.

0.2 Understanding ChampSim's code

The following section contains our understanding of `cache.cc`'s various functions, after going through `handle_read()` and `handle_fill()` in particular.

1. Incase of a **read hit**, the `handle_read()` function gets the data and adds the packet along with the data (for TLB's and L1I, L1D caches) to the processed queue. For L2C and LLC, the data is sent to the MSHR of upper level cache by `return_data()`. If the instruction is a **load**, the prefetch status of the cache line is updated and next data is prefetched. Finally, the current cache line is updated to MRU if the replacement policy used is LRU, i.e. the replacement policy's promotion rule is applied. Additionally some other stats are calculated. The entry is removed from the read queue (RQ) after handling.

2. In case of a **read miss**, we first check whether the request is already present in the MSHR of the respective cache. Based on this, we have the following cases.

   - If the miss is not present in the MSHR, and the MSHR has space, then an entry is added to the MSHR. Also, the request is forwarded to the RQ of the next (lower) level cache.
   - If the miss is not present in the MSHR, and the MSHR is full, then a STALL is introduced.
   - If the miss is already present in the MSHR, it is *merged* with the corresponding miss in the list.

   We also add this packet to the read queue of the next level (if not last level). Finally, the prefetch status of that cache line is updated, and the following data is prefetched in case of a load instruction. The entry is removed from the RQ after handling.

3. If there is a cache miss for Read request in some level(say A), then it adds the request in it's MSHR and RQ of next level cache(say B), When the cache B gets the data it fills it in the MSHR of A, then A reads the data from the MSHR and fills it in cache A. In particular, it is convenient to read the data from the MSHR (after a miss) than from the cache. Otherwise, if we didn't have MSHR storing the missed data, we would need additional cycles for the caches to communicate with each other.

4. `Upper_level` caches are previous level caches, i.e L2 cache for LLC, and L1I/L1D for L2.

   `Lower_level` caches are next level caches, i.e, L2 for L1I/L1D, and LLC for L2.

   The values of these caches are assigned in the `main()` function of `src/main.cc` file. These variables are used in the code to access the neighbouring caches and to add requests to their queues and return data to the respective MSHR.

   In particular, the initialization is done in - lines 133, 134 for the upper level (`upper_level_icache[i] = NULL; upper_level_dcache[i] = NULL;`), and lines 148 (`lower_level = NULL`) for lower level in the original inc/cache.h file.

   The assignment is done in the `src/main.cc` file between the lines 692 and 785.

5. The `return_data()` function is used to check if a request is present in the MSHR. If the request is not found, an error is raised. Otherwise, the data of the corresponding cache line is added to that MSHR entry. In a nutshell, this function returns data to the MSHR of the higher cache as a part of miss handling.

1. Implement LLC Bypassing

The code has been well commented to guide the reader in understanding the logic. However, we are presenting our ideas below.

The crux of the implementation lies in the fact that data is written into the cache using the `fill_cache` function. Each of the functions in `cache.cc` is edited as follows.

- `handle_fill`
  We simply check if the PC of the MSHR packet is present in the PC Predictor table or not. If it is, we use the policy given by the entry. Otherwise, we use the global policy. If the policy recommends us to bypass the packet, we do so using the code mentioned inside `#ifdef LLC_BYPASS`.

- `handle_writeback`
  This function is used to write data into the caches as a part of the write back policy. The first action in this function would be to update the PC predictor table. We check whether the write queue (`WQ`) packet is present in the tracker sets using its PC. If it is, we update the counter based on cache hit/miss. We also need to index into the PC table using the global policy. That is, we track the histories of each policy separately for each PC.

  Another important action that we are supposed to do here is decrementing the reference counter (the no. of entries in the cache with a given PC). When we write back into the cache, if a block is being evicted, and is consequently added to the write queue, then we need to decrement its reference in the cache.

  Finally, in case of a miss and the bypass policy, we need to skip writing the packet into the cache. We simply do this by adding our entry into the write queue and leaving the entry present in the cache as it is.

- `handle_read`
  We do not have to make many changes in this function. We just have to update the PC predictor table of PCs present in the tracker sets based on a hit/miss.

The above design choice is rather a complicated one, and it could have been done more simply. However, we started writing the code before we completely understood the flow of the code, which led to more sophisticated changes in the code. That being said, the logic followed is simply based on the MadCache[1] paper, and is not too difficult to identify in our implementation.

We also describe a little bit more about the implementation of the PC predictor table and other variables here.

The PC Predictor table is implemented as a C `struct`. It contains the following variables.

- `PC` - Stores the program counter for the entry.
- `policy_counter` - Stores the local policy counter specific to the PC. This is updated in the handle_writeback and handle_read functions based on hits and misses. In particular, this is incremented in `handle_read` miss and `handle_writeback` miss. It is decremented in `handle_read` hit and `handle_writeback` hit. If the MSB is 1, we use **BBIP**, otherwise we use **LRU**.
- `ref` - This is just the no. of entries for each PC. It is incremented in `handle_fill` and `handle_read` miss. Also, it is decremented in `handle_writeback` and `handle_fill` for the evicted packets.
- `policy_bit` - This stores the type of history tracked for the PC. It is initialized when a new PC is added to the Predictor table, and it used for indexing into the predictor array in the future.

For the global policy, we use a 10-bit counter **glbl_policy**. This counter is basically the default policy for the PCs in the follower set. Also, we added the variables **reuse** and **mypctable_ind** to each cache line using the class **BLOCK**. The former variable simply stores if the block has been reused or not. The latter is the PC Predictor table entry index associated with the particular block. This variable is helpful while changing the reference variable.

Finally, we added the variable **pol** in class **PACKET** to know the default policy when packet is added into the queue.

We have defined all the parameters relating to the MadCache algorithm in the `cache.h` file. The parameters set are -

- `PC_TB_SIZE` - This stores the size of the PC predictor table which is 1024.
- `POLICY_MAX_VAL` - This essentially stores the number of bits in the policy counter for each entry in the PC table. It is set to 6. That is, the maximum value of the counter can be $2^6 - 1$.
- `REF_MAX` - This variable stores the maximum number of entries in the cache for a PC. It is set to $2^9 - 1$.
- `MAX_GLOBAL_POLICY` - This variable essentially stores the maximum value of global policy counter, and is set to $2^{10} - 1$.
- `INITIAL_POLICY` - The initial value of the PC counter set.And is set to $2^5 - 1$
- `POLICY_THRESH` - A mask to check the local policy.
- `GLOBAL_THRESH` - A mask to check the global policy.

- `bbip_epsilon` - The probability percentage with which the BBIP policy does not bypass to protect from thrashing.

To complete the implementation aspect, here are the functions defined and their use.

- `defaultPolicy` - This function returns the current global policy using the global counter.
- `checkPCArray` - Given an address, the function returns the index of the PC if present in the PC Predictor array.
- `checkTrackerSet` - Given a set, the function checks whether the set is present in the tracker sets or not.
- `lowRefPC` - Returns an index from the PC predictor table whose number of entries are zero.
- `checkBypass` - Given an address, the function returns whether we have to bypass based on the policy.
- `updateTable` - This function is used to update the PC predictor table given the address and cache hit/miss status. The updates are done only for the PCs in the tracker sets.

2. LLC bypassing results

The files we have edited are `cache.cc`, `cache.h` and `block.h`. The places where we have edited the code are enclosed between multiple front slashes (`///`).
For running the code, copy these three files to original ChampSim folder into the respective directories. Then build the simulator, and run it using the trace given in the problem statement.

The conditions which are used to check whether we have to bypass or not are shown in the following flowchart (on the next page). Basically, when we bypass, we should prevent the data being written into the cache. Writing the data to the cache is done in `handle_fill` and `handle_writeback` as described previously.

In the former function, we simply use the `#ifdef LLC_BYPASS` to achieve the bypass functionality. This covers the bypassing part for reads.

In `handle_writeback`, we do the following. In case of a writeback hit, we cannot bypass the data. In case of a writeback miss, we implemented a clever manoeuvre to achieve bypassing.
Without any changes, the code adds the current data in the cache into the writeback queue for passing into the next cache level. The packet that arrives in the function is then written into the cache. However, in case of bypassing, we simply add the arriving packet into the writeback queue, instead of the block that is already present in the cache. We also prevent the `fill_cache` function from executing. This way, we achieve bypassing in writes.

The IPC improved from 0.142776 (in baseline implementation) to 0.244537 (in MadCache implementation) for 10M warmup and 10M simulation instructions. The IPC improvement in percentage is 71.3%. The baseline results are stored in the folder `baseline_no_bypass_result` and the MadCache ones are stored in `llc_bypass_result`

For 1M warmup and simulation instructions, the IPC for MadCache is 0.157448. On the other hand, the IPC for baseline is 0.139477. Therefore, we see 13% improvement here.

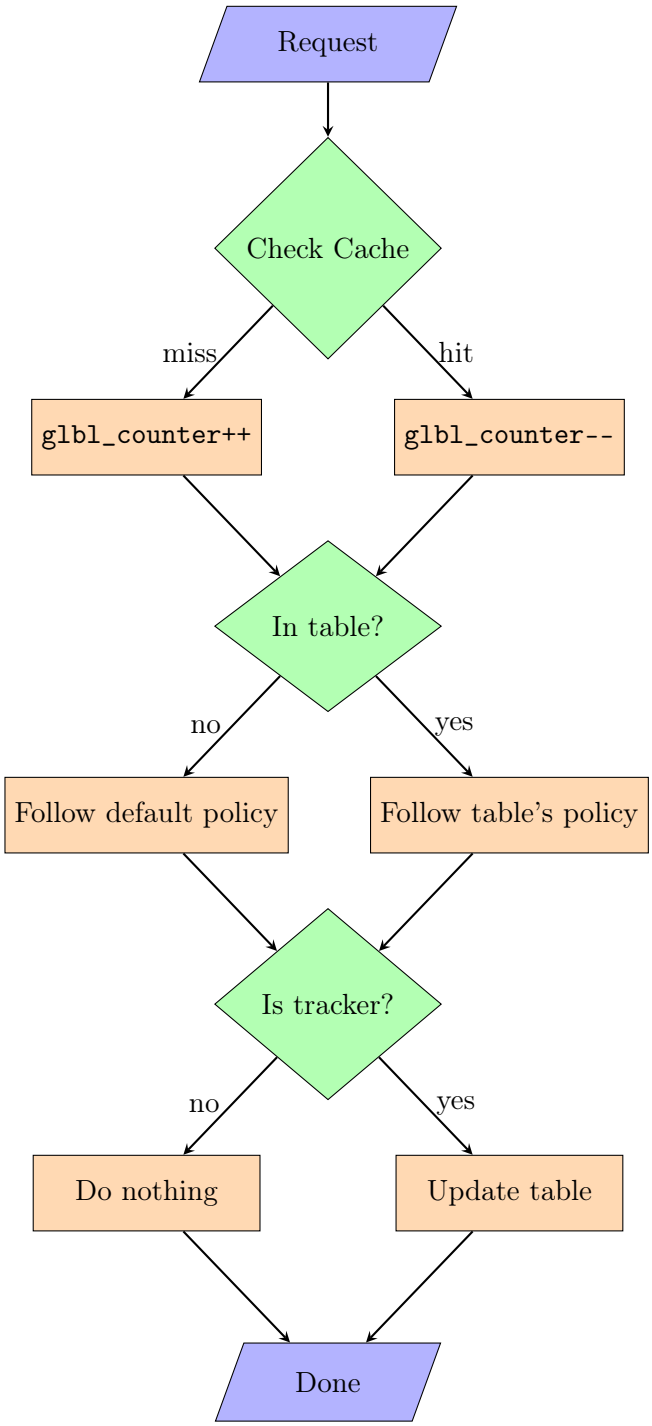

Figure 2.1: Highlevel flowchart of the implemented LLC Bypass

3. Prefetch Throttling Implementation

The following chapters concern the implementation of IP-based prefetch throttling, and the improvements obtained. The inspiration for our design was drawn from previous works [2] on implementing the same for shared caches. The implemented design has been explained in the current section below. The code has appropriate comments added as well.

The implementation consists of two parameters, called `MAX_DEGREE` and `MIN_DEGREE`. The baseline implementation uses a static prefetch degree of 3. For the sake of comparison, we vary `MAX_DEGREE` and have fixed `MIN_DEGREE` to be 3 (as stated in this piazza post).

Hit fraction (f) varying between 0 and 1 is computed. The degree is then computed by linear approximation, with 0 corresponding to `MIN_DEGREE` and 1 to `MAX_DEGREE`. That is, for f the degree is given by the following formula.

$$\text{Degree} = \lfloor \texttt{MIN\_DEGREE} + (\texttt{MAX\_DEGREE} - \texttt{MIN\_DEGREE}) \cdot f \rfloor$$

`IP_TRACKER` table has been slightly modified to track f. Two new variables called `tot` and `cnt` have been added, whose division yields f. Do note that the values are updated only when the if condition `stride == trackers[index].last_stride` is satisfied, so we don't have pre fetching with degree 3 even if the value of f is 0.

The normalized IPC, L1D coverage and L2C coverage are tabulated as below.

`MAX_DEGREE`	**IPC**	**L1D Coverage**	**L2C Coverage**	**Normalized IPC**
Baseline	0.21318	0.25294	0.3926	1
max_5	0.220275	0.26087	0.41211	1.033281734
max_7	0.229926	0.30535	0.46194	1.078553335
max_9	0.237587	0.31134	0.48089	1.114490102
max_11	0.242698	0.31829	0.49930	1.138465147
max_13	0.24534	0.32081	0.5129	1.150858429
max_15	0.249777	0.32174	0.53478	1.171671827
max_17	0.252201	0.32764	0.5489	1.183042499
max_19	0.253137	0.33511	0.55641	1.187433155
max_21	0.253352	0.34140	0.55934	1.188441692

4. Prefetch Throttling Results

The resulting baseline files have been stored in `baseline_no_throttling_result` in the submitted tarball file. Similarly, the submitted prefetch throttler has `MAX_DEGREE` of 11, and the files have been stored in `prefetch_throttling_result`.

The table given in the previous section contains the data required for gauging the performance of the prefetch throttler. However, for ease of interpretation, graphical representations have also been provided below.

4.1 Normalized IPC improvement with respect to thresholds used

The normalized IPC increases with the threshold. Although it is found that the rate of improvement is not monotonous, it gradually decreases for higher thresholds. It is quite likely that the IPC would fall after a further increment due to large number of prefetch requests. The following plot represents the observed normalized IPC and threshold relation.

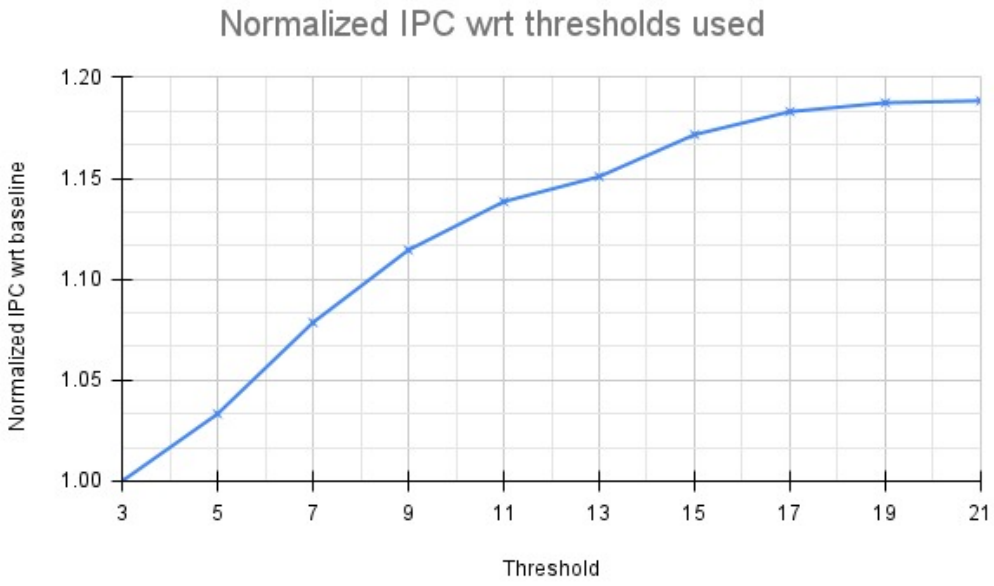

Figure 4.1: Normalized IPC wrt thresholds used

4.2 Prefetcher coverage vs thresholds used

As expected, the coverage for both the caches increases at first but then the increment starts to decline slowly owing to important entries being pushed out to make room for the prefetched data. A similar trend can be seen for L2C as well, with the increment declining slowly.

We have calculated coverage for both the cases as (cache misses eliminated by prefetching with throttling)/(cache misses without any prefetching). We have only considered the load cache misses for both L1D and L2C.

Graphs for both have been given below.

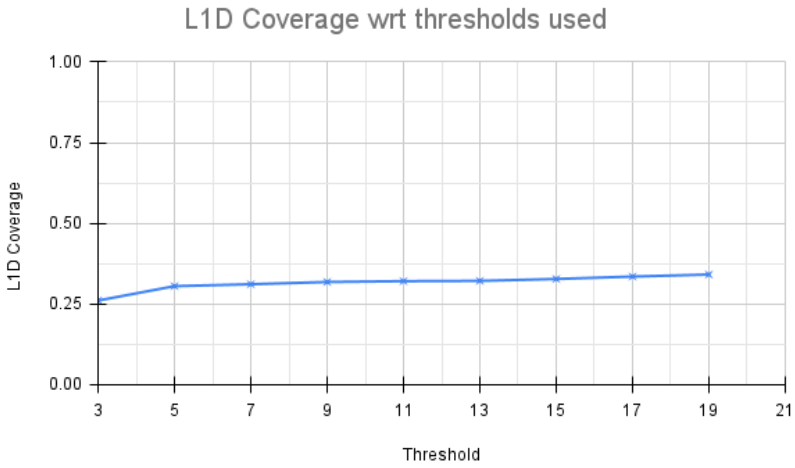

Figure 4.2: L1D Coverage wrt thresholds used

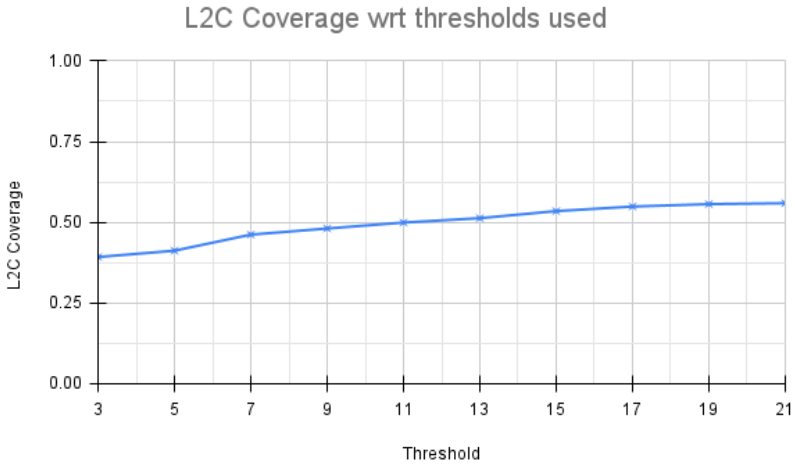

Figure 4.3: L2C Coverage wrt thresholds used

4.3 Running the code

The provided `ip_stride.l2c_pref` was modified with the above mentioned changes to implement throttling, and was renamed to `ip-stride.l2c_pref` to align with the problem statement. The same was then copied over to `ip-stride.l1d_pref` with appropriate changes.

Run the following command to build champsim, and run using any trace as you normally would.

```
./build_champsim.sh bimodal no ip-stride ip-stride no lru 1
```

Note the `#define TASK3` line in `cache.cc`.

5. Contributions

Table 5.1: Contributions of each team member

Member	**Total Contribution**	**Work Done**
Thivesh Chandra (Leader)	20%	Part 0-Q2 and Part 1 code
Sudhansh	20%	Part 0-Q2 and Part 1 code + report.
Akash Cherukuri	20%	Part 0-Q1, Part 3 code, Report
Battepati Karthikeya	20%	Part 3 code, Report
Mastansha	20%	Parts 3 and 4 Report

Bibliography

[1] Hayenga, Mitch & Nere, Andrew & Lipasti, Mikko. (2010). MadCache: A PC-aware Cache Insertion Policy. JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship.

[2] O. Ozturk, Seung Woo Son, M. Kandemir and M. Karakoy, "Prefetch throttling and data pinning for improving performance of shared caches," SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1-12, doi: 10.1109/SC.2008.5213128.