CS341: Computer Architecture Lab

Lab 5: LLC Bypassing and Prefetching
Report

Team - Hazard Eliminators

Adarsh Raj (190050004)
Aditya Badola (190050006)
Gudipaty Aniket (190050041)
Sasmit Vaidya Swapnil (190040129)
Soham Mistri (190050116)

October 2021

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2021-2022

Contents

1. Paper review

1.1 Part 0

In this paper, the authors have proposed MadCache, a dynamic cache insertion policy which seeks to learn the pattern of L3 cache accesses based on the Program Counter(PC). The two main policies which the L3 cache uses are the default **LRU policy** which is good if a PC exploits memory locality, and **bypass,** which is useful for streaming services because locality is not used there.

Further, this policy allows the PC to override the default behaviour if there is "enough" evidence to support the same.

Coming into the details, the data structures maintained by this policy are:

- **PC Predictor table**: This table is basically a mapping from a given PC to a 6 bit counter, which indicates the policy the sets using the PC will use.
- **Tracker sets**: These are a subset all the sets present in the cache. The PC behaviour is learnt from these tracker sets, and whatever behaviour is learnt from these sets will be applicable for all the sets in cache for a particular PC. Those sets are called **follower sets**.

So, the counter value initially is set to some intermediate value, so that it is just below the threshold and follows LRU policy. However, if the block is evicted without use, the counter increases, and once above threshold(indicated by the MSB), the policy can be overridden for that PC and set access and we shift to bypass policy.

While a PC may exhibit streaming behavior for one portion of the program, it is possible it changes its behavior later on. For such a case, thrashing protection must be enabled to allow a PC to get out of bypass mode. In thrashing protection, a few lines are inserted to the MRU position.

The paper makes an important assumption that a given PC is likely to follow a similar memory access pattern over a majority of time, based on which it can do the learning and avoid cache misses.

The downside of this policy is the added hardware overhead it uses for learning these patterns based on the PC, so in case we get a lot of memory accesses, maintaining a huge PC predictor table can be difficult. Also the hardware overload substantially increases to deal with multi threaded systems.

2. Experiments

2.1 Part 1

For building ChampSim, use parameters:

- Branch Predictor - `bimodal`
- L1I Prefetcher - `no`
- L1D Prefetcher - `no`
- L2C Prefetcher - `no`
- LLC Prefetcher - `no`
- Replacement Policy - `lru`
- Number of CPU cores - `1`

The number of warm-up instructions are $10M$ and simulation instructions are $10M$.

Running Code Instructions

- Comment both the lines `#define T1_BYPASS` and `#define T3_THROTTLE` in the file `inc\cache.h` to run for Baseline Results.

- To run for LLC Bypassing, uncomment `#define T1_BYPASS`.

2.2 Part 3

For building ChampSim, use parameters:

- Branch Predictor - `bimodal`
- L1I Prefetcher - `no`
- L1D Prefetcher - `ip-stride`
- L2C Prefetcher - `ip-stride`
- LLC Prefetcher - `no`
- Replacement Policy - `lru`
- Number of CPU cores - `1`

The number of warm-up instructions are $10M$ and simulation instructions are $10M$.

Running Code Instructions

- Comment both the lines `#define T1_BYPASS` and `#define T3_THROTTLE` in the file `inc\cache.h` and build & run the ChampSim with the given parameters for Baseline Results.

- To run for Prefetcher Throttling, uncomment `#define T3_THROTTLE`. Update the `THRESHOLD` in files `ip_stride.l1d_pref` and `ip_stride.l2c_pref` (in the `prefetcher\` subdirectory), to run for different threshold values.

3. Code and Observations

3.1 Part 0

Answers pertaining to `cache.cc` are as follows:

1. In `handle_read()`, upon a cache hit, the following steps are taken:

    - The `PROCESSED` queue is updated, adding a pointer to the current entry in the read queue (provided the queue is not full). For L1D, this is done only if the request wasn't a prefetch request. In case of a TLB hit, the physical address for the read queue entry is provided.
    - In case of a LOAD instruction hitting an instruction or a data cache, the prefetcher is appropriately updated.
    - The cache replacement policy is updated, accounting for the current access.
    - Checks the cache level at which the line has to be filled, and if it is to be filled at a lower level, the cache line is returned to a lower level.
    - Prefetch bit is set to 0, and stats are updated.
    - Finally, the entry is removed from the read queue, and the variable `reads_available_this_cycle` is decremented.

2. In `handle_read()`, upon a cache miss, the following steps are taken:

    - First, the `mshr_index` is checked, to ascertain whether the cache miss collides with an old miss, or if it is a new miss. If it is a new miss, then we also have to check if the MSHRs are all occupied.
    - In case of a new miss with space in the mshr, if the cache is LLC, we check if the DRAM has enough space in the read queue for this request, and then add it to our MSHR and the DRAMs read queue. In case it isn't LLC, then we simply add it to our MSHR and the next level's read queue.
    - In case an MSHR is not available, we stall this request until an MSHR is available.
    - In case the current miss is an already in-flight miss, the requested miss is merged with the in-flight miss, and if the fill level of this read request is lower than that in the MSHR, then the fill level in the MSHR is also updated appropriately.
    - We have a variable `miss_handled`, which tells us whether this miss has been handled or not. `miss_handled` is set to 0 if the miss collides with another miss, or if the miss is from the LLC and the read queue of DRAM is full.

- If the miss has been handled, then the prefetcher is updated, the stats are updated (misses and accesses). The request is removed from the read queue of the cache.

3. The function `handle_fill()` reads request from the cache's MSHR, because the MSHR entry has the information about which cache level to fill, and some other information which is used in the function.

4. Upper level cache for a given cache is the cache which is just closer to the core, and the lower level cache is the cache which is just further from the core. For example, for an L2C cache, the upper level data cache is L1D, and the lower level data cache is the LLC cache. The upper level and lower level cache are initialised by default to `NULL` in `cache.h`, and are assigned to different caches in the code file `main.cc`.

5. The `return_data()` function returns the cache data into the MSHR, marking the completion of the request. If puts the data and meta-data in the appropriate MSHR entry, and appropriately updates the `event_cycle` field of the entry, accounting for the latency. Finally, it calls the function `update_fill_cycle()`, which finds the next filled MSHR index which was filled with data the earliest.

3.2 Part 2

Code Design Description

We have implemented an adaptive insertion policy for L3 cache that uses the PC's memory access history and behavior to determine the currently used insertion policy.

Following data structures are used to implement MadCache:

- `PC_PREDICTOR_TABLE`: A lookup table which essentially maps the program counter to counter and number of tracker set cache entries associated with this PC. `64 bits` PC is used to index in this table with `1024` entries. Every table entry comprises of a `6 bit` counter and a `9 bit` entries. The MSB of the counter indicates bypassing.
- `TRACKER_SET`: A tracker set is a $\frac{1}{16}^{th}$-sized subset of the L3 cache. The sets are included in the tracker set based on dynamic set sampling (DSS). That is, the sets are chosen uniformly at random to be part of the tracker sets. Each tracker entry stores a `64 bit` PC to index into the PC predictor table.
- `DEFAULT_POLICY`: A `10 bit` counter whose MSB indicated the global policy.

Initially, there are no entries in the PC predictor table. Also, the tracker set entries map to garbage value. When the first miss occurs, if the corresponding PC accesses a tracker set cache line, then add an entry in the PC predictor table and set the PC as the PC that brought the line into the cache. In case of a hit, the global counter is decremented. The counter for the PC is also decreased if it belongs in the PC predictor table.

In case when a new PC accesses an already-accessed cacheline from tracker set, the tracker set entry will now have the updated PC. If the older PC is in the PC prediction table, increment the counter and decrease the number of entries to implicitly evict it when number of entries become zero. Increase the number of entries for the current PC if already present in the table, else add a new table entry.

Note that the updates are only made for tracker set cachelines. However, both follower and tracker sets have the capacity to override the default policy.

To resolve the issue of thrashing, even if the MSB of the counter is set, with a small finite probability (1/128 in the implementation), the policy followed can be LRU.

List of source files added or edited to implement LLC Bypassing

- Edited `src\cache.cc`
- Edited `inc\cache.h`
- Added `inc\madcache.h`

IPC Improvement Result

The results text files can be found in the `baseline_no_bypass_result\` and `llc_bypass_result\` folder in the submission directory.

Observed IPC

- *for baseline*: 0.142776
- *with bypassing*: 0.217305

Calculation of IPC Improvement:

$$\begin{aligned} IPC_{improve} &= \frac{IPC_{new} - IPC_{base}}{IPC_{base}} \\ &= \frac{0.217305 - 0.142776}{0.142776} \\ IPC_{improve} &= 0.52199946 \end{aligned}$$

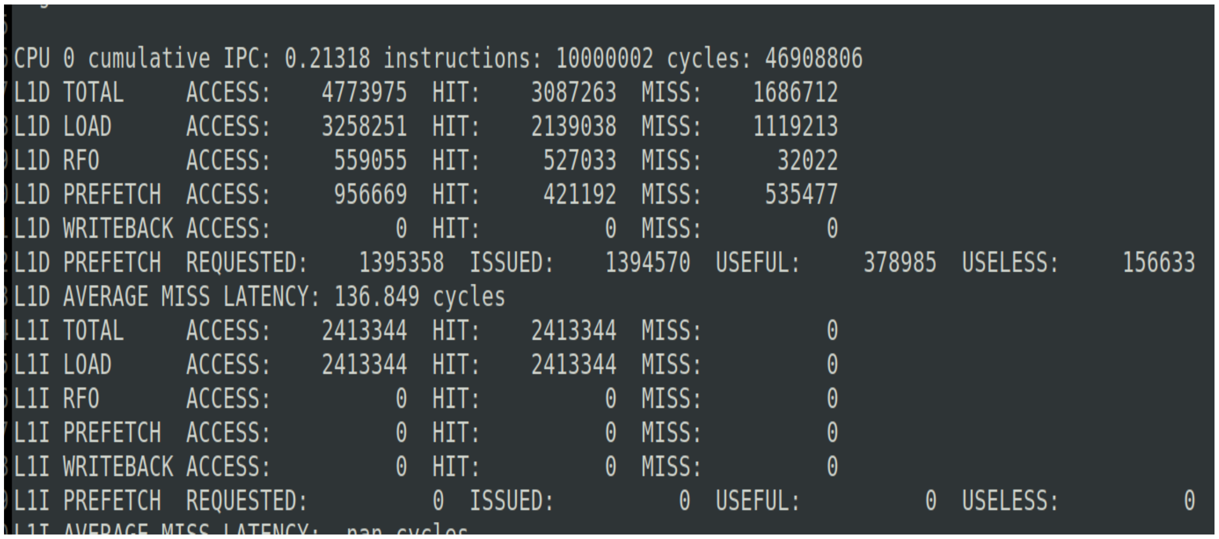
So, we achieved an IPC Improvement of around 52.2% using LLC Bypassing.

3.3 Part 4

Baseline Results

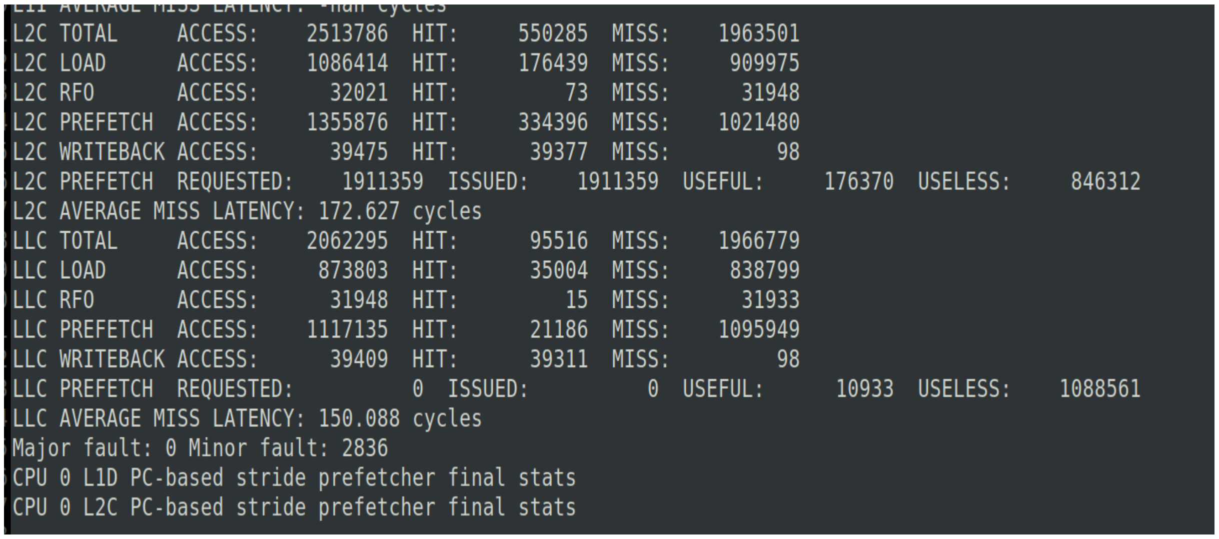
The results text file can be found in the `baseline_no_throttling_result\` folder in the submission folder.

Observed IPC: 0.21318



```
CPU 0 cumulative IPC: 0.21318 instructions: 10000002 cycles: 46908806
L1D TOTAL     ACCESS:    4773975  HIT:    3087263  MISS:    1686712
L1D LOAD      ACCESS:    3258251  HIT:    2139038  MISS:    1119213
L1D RFO       ACCESS:     559055  HIT:     527033  MISS:      32022
L1D PREFETCH  ACCESS:     956669  HIT:     421192  MISS:     535477
L1D WRITEBACK ACCESS:          0  HIT:          0  MISS:          0
L1D PREFETCH  REQUESTED:    1395358  ISSUED:    1394570  USEFUL:     378985  USELESS:     156633
L1D AVERAGE MISS LATENCY: 136.849 cycles
L1I TOTAL     ACCESS:    2413344  HIT:    2413344  MISS:          0
L1I LOAD      ACCESS:    2413344  HIT:    2413344  MISS:          0
L1I RFO       ACCESS:          0  HIT:          0  MISS:          0
L1I PREFETCH  ACCESS:          0  HIT:          0  MISS:          0
L1I WRITEBACK ACCESS:          0  HIT:          0  MISS:          0
L1I PREFETCH  REQUESTED:          0  ISSUED:          0  USEFUL:          0  USELESS:          0
L1I AVERAGE MISS LATENCY: -nan cycles
```

Figure 3.1: Basline Results Snap 1



```
L1I AVERAGE MISS LATENCY: -nan cycles
L2C TOTAL     ACCESS:    2513786  HIT:     550285  MISS:    1963501
L2C LOAD      ACCESS:    1086414  HIT:     176439  MISS:     909975
L2C RFO       ACCESS:      32021  HIT:         73  MISS:      31948
L2C PREFETCH  ACCESS:    1355876  HIT:     334396  MISS:    1021480
L2C WRITEBACK ACCESS:      39475  HIT:      39377  MISS:         98
L2C PREFETCH  REQUESTED:    1911359  ISSUED:    1911359  USEFUL:     176370  USELESS:     846312
L2C AVERAGE MISS LATENCY: 172.627 cycles
LLC TOTAL     ACCESS:    2062295  HIT:      95516  MISS:    1966779
LLC LOAD      ACCESS:     873803  HIT:      35004  MISS:     838799
LLC RFO       ACCESS:      31948  HIT:         15  MISS:      31933
LLC PREFETCH  ACCESS:    1117135  HIT:      21186  MISS:    1095949
LLC WRITEBACK ACCESS:      39409  HIT:      39311  MISS:         98
LLC PREFETCH  REQUESTED:          0  ISSUED:          0  USEFUL:      10933  USELESS:    1088561
LLC AVERAGE MISS LATENCY: 150.088 cycles
Major fault: 0 Minor fault: 2836
CPU 0 L1D PC-based stride prefetcher final stats
CPU 0 L2C PC-based stride prefetcher final stats
```

Figure 3.2: Basline Results Snap 2

Prefetch Throttling Results

For the prefetch throttling, the initial `PREFETCH_DEGREE` was set to **3**, and it was varied depending on the accuracy of tracker for an IP, and the range was set to **(3,6)** (both inclusive). We increment the `PREFETCH_DEGREE` by 3 if accuracy is greater than the given threshold, else set to the default value, in the `prefetch_throttle()` function.

Calculation of Normalised IPC Improvement:

$$IPC_{improve} = \frac{IPC_{new} - IPC_{base}}{IPC_{base}}$$

$$IPC_{improve} = \frac{IPC_{new}}{0.21318} - 1$$

For calculation of Prefetcher Coverage, for a cache level, we will consider all misses except `prefetch` misses. Hence, considering writeback, rfo and load misses we have:

$$\texttt{Cov} = \frac{\texttt{Useful}}{\texttt{Useful} + \texttt{Misses}_w + \texttt{Misses}_l + \texttt{Misses}_{rfo}}$$

Where, `Useful` is the number of useful prefetches and $\texttt{Misses}_w$ is number of writeback misses, similarly $\texttt{Misses}_l$ for Load Misses, and $\texttt{Misses}_{rfo}$ for RFO misses. For the below, we will denote `Misses` as a triplet of load, RFO and writeback misses.

Threshold = 0.25

Result data File in `prefetch_throttling_result\t=0.25\`:

- IPC - 0.22814
- Useful Prefetches for L1D - 450411
- Useful Prefetches for L2C - 200518
- Misses for L1D (LOAD, RFO, WB) - 1047791, 32020, 0
- Misses for L2C (LOAD, RFO, WB) - 817400, 31940, 96
- Normalized IPC Improvement $= (0.22814/0.21318) - 1 = 0.070175$
- Prefetcher Coverage for L1D $= (450411/(1047791 + 32020 + 0 + 450411)) = 0.294343$
- Prefetcher Coverage for L2C $= (200518/(817400 + 31940 + 96 + 200518)) = 0.190977$

Threshold = 0.3

Result data File in `prefetch_throttling_result\t=0.3\`:

- IPC - 0.228395
- Useful Prefetches for L1D - 450976
- Useful Prefetches for L2C - 200526
- Misses for L1D (LOAD, RFO, WB) - 1047224, 32018, 0
- Misses for L2C (LOAD, RFO, WB) - 816938, 31940, 96
- Normalized IPC Improvement $= (0.228395/0.21318) - 1 = 0.0713716$
- Prefetcher Coverage for L1D $= (450976/(1047224 + 32018 + 0 + 450976)) = 0.2947135$
- Prefetcher Coverage for L2C $= (213172/(816938 + 31940 + 96 + 200526)) = 0.2031176$

Threshold = 0.4

Result data File in `prefetch_throttling_result\t=0.4\`:

- IPC - 0.222859
- Useful Prefetches for L1D - 443032
- Useful Prefetches for L2C - 189388
- Misses for L1D (LOAD, RFO, WB) - 1055163, 32020, 0
- Misses for L2C (LOAD, RFO, WB) - 840241, 31940, 98
- Normalized IPC Improvement $= (0.222859/0.21318) - 1 = 0.0454029$
- Prefetcher Coverage for L1D $= (443032/(1055163 + 32020 + 0 + 443032)) = 0.2895227$
- Prefetcher Coverage for L2C $= (189388/(840241 + 31940 + 98 + 189388)) = 0.1783874$

Threshold = 0.5

Result data File in `prefetch_throttling_result\t=0.5\`:

- IPC - 0.217768
- Useful Prefetches for L1D - 410838
- Useful Prefetches for L2C - 182206
- Misses for L1D (LOAD, RFO, WB) - 1087354, 32017, 0
- Misses for L2C (LOAD, RFO, WB) - 880143, 31940, 96
- Normalized IPC Improvement = $(0.217768/0.21318) - 1 = 0.02152$
- Prefetcher Coverage for L1D = $(410838/(1087354 + 32017 + 0 + 410838)) = 0.26848$
- Prefetcher Coverage for L2C = $(182206/(880143 + 31940 + 96 + 182206)) = 0.16649$

Threshold = 0.6

Result data File in `prefetch_throttling_result\t=0.6\`:

- IPC - 0.215733
- Useful Prefetches for L1D - 392527
- Useful Prefetches for L2C - 174900
- Misses for L1D (LOAD, RFO, WB) - 1105668, 32016, 0
- Misses for L2C (LOAD, RFO, WB) - 906074, 31940, 95
- Normalized IPC Improvement = $(0.215733/0.21318) - 1 = 0.0119758$
- Prefetcher Coverage for L1D = $(392527/(1105668 + 0 + 392527 + 32016)) = 0.2565182$
- Prefetcher Coverage for L2C = $(174900/(906074 + 95 + 174900 + 31940)) = 0.1571416$

Threshold = 0.75

Result data File in `prefetch_throttling_result\t=0.75\`:

- IPC - 0.215733
- Useful Prefetches for L1D - 392527
- Useful Prefetches for L2C - 174900
- Misses for L1D (LOAD, RFO, WB) - 1105668, 32016, 0
- Misses for L2C (LOAD, RFO, WB) - 906074, 31940, 95
- Normalized IPC Improvement $= (0.215733/0.21318) - 1 = 0.0119758$
- Prefetcher Coverage for L1D $= (392527/(1105668 + 32016 + 0 + 392527)) = 0.2565182$
- Prefetcher Coverage for L2C $= (174900/(906074 + 31940 + 95 + 174900)) = 0.1571416$

Threshold = 0.8

Result data File in `prefetch_throttling_result\t=0.8\`:

- IPC - 0.215725
- Useful Prefetches for L1D - 392518
- Useful Prefetches for L2C - 174666
- Misses for L1D (LOAD, RFO, WB) - 1105676, 32016, 0
- Misses for L2C (LOAD, RFO, WB) - 906387, 31940, 93
- Normalized IPC Improvement $= (0.215725/0.21318) - 1 = 0.0119383$
- Prefetcher Coverage for L1D $= (392518/(1105676 + 32016 + 0 + 392518)) = 0.2565125$
- Prefetcher Coverage for L2C $= (174666/(906387 + 93 + 31940 + 174666)) = 0.1569205$

Prefetch Throttling Results Plots

Normalised IPC Improvement:

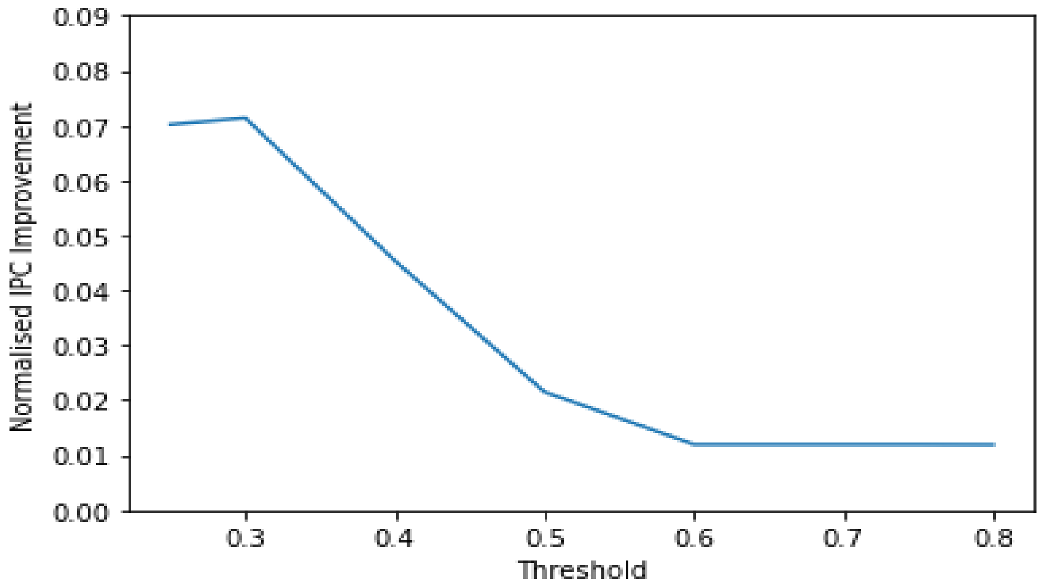


Figure 3.3: Normalized IPC Improvement w.r.to Thresholds

Prefetcher Coverage:

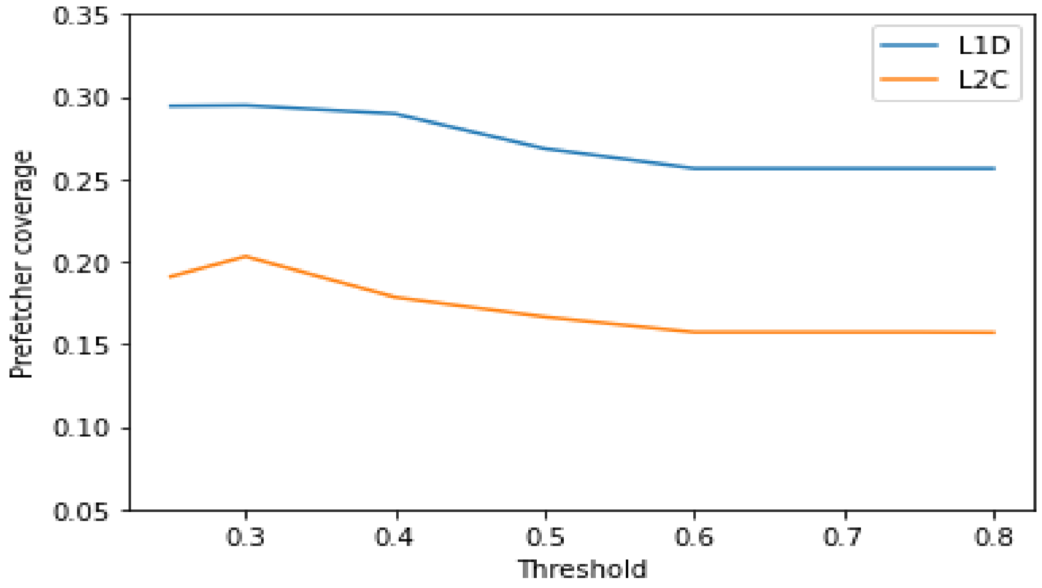


Figure 3.4: Prefetcher Coverage for Different Thresholds

List of source files added or edited to implement Prefetch Throttling

- Edited `src\cache.cc`
- Edited `inc\cache.h`
- Edited `prefetcher\ip_stride.l2c_pref`
- Added `prefetcher\ip_stride.l1d_pref`

Logic for the changes made during implementation

- Firstly, the prefetch accuracy for each IP is the ratio of the number of useful prefetches to the sum of useful and useless prefetches. So, we need to keep re-calculating the accuracy for each IP in the `trackers` array for L2C prefetcher, and `l1_trackers` array for L1D prefetcher.
- A useful prefetch is identified in the `l1d_prefetcher_operate()` and `l2c_prefetcher_operate()`, when a `cache_hit` is observed and the `type` argument equals `PREFETCH`.
- A useless prefetch is detected in the `fill_cache()` function, wherein we check if the evicted block was a prefetched block and wasn't used.
- Functions `l1d_prefetch_throttle()` and `l2c_prefetch_throttle()` were implemented, and have similar implementations and signatures. They take 2 arguments - ip and `is_useful`, which tells us if we have detected a useful or useless prefetch. These functions have been implemented as member functions of the `CACHE` class, and are defined in the prefetcher files. These functions update the state of the relevant ip tracker and calculate it's accuracy, and update the prefetch degree for that ip tracker if needed, based on the threshold.
- Member variables `prefetch_degree`, `useful`, `useless` and `accuracy` have been added to the `IP_TRACKER` class, in order to dynamically update the state per-ip.
- **NOTE** : The accuracy threshold has been defined as a macro `THRESHOLD` in both the L1D and L2C prefetcher files. In order to try out different threshold values or to check the default threshold in our submission, refer to that macro and modify it if needed.

4. Contributions

Adarsh Raj - Tested the code for Part 3, with different value of `PREFETCH_DEGREE` and with different ways of degree changes, i.e gradual change ($\pm$ 1) and fixed change (changing between min and max values). Collected Baseline Result and Prefetch Throttling Results for thresholds - (0.3, 0.4, 0.6, 0.8).

Aditya Badola - Implemented Part 1, i.e LLC Bypassing in code, added comments to the code, tested and debugged code for the same, added the corresponding observations in report.

Gudipaty Aniket - Implemented Part 3, i.e Prefetcher Throttling in code, added comments for the same, modified the code for different inital `PREFETCH_DEGREE` (example - 3,4,6), and modified the code for accommodating different ways of change of prefetch degree for a tracker.

Sasmit Vaidya Swapnil - Tested the code for Part 3, with different value of `PREFETCH_DEGREE`. Collected Prefetch Throttling Results for thresholds - (0.25, 0.5, 0.75). Made plots for Normalised IPC Improvement from the collected data, Prefetcher Coverage for L1D and L2C, and added them in report.

Soham Mistri - Added summary of the MadCache Paper in the report.