Lab 4 Part 2 Report

Ved Danait

October 2023

1 Part 1 : Replacement Policies

1.1 Speedup

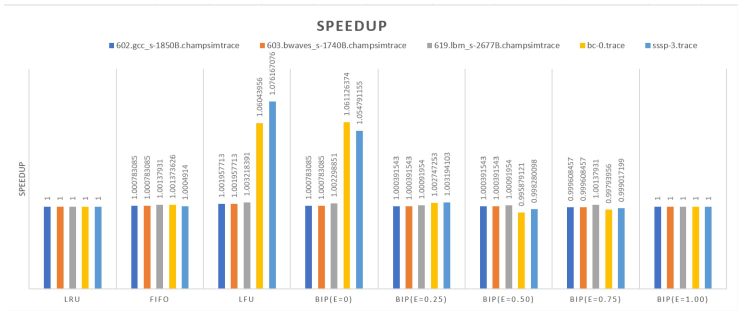

Figure 1: Speedup in IPC with respect to LRU Replacement

1.2 L2C Miss Rate

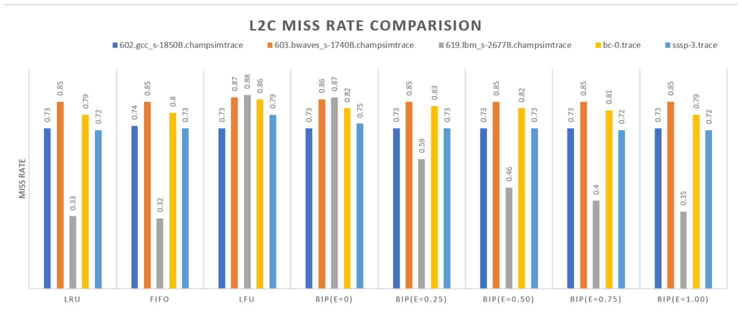

Figure 2: L2C Miss Rate

1.3 Explanation

LRU : This the default replacement policy for L2c which evicts the least recently used block, this is done by tracking clock cycle at which the cache line enters the L2C and the clock cycle at which it is accessed. At any point when we wish to evict a line, we find the minimum cache cycle, which denotes the least recently used line. Note that we store these in the array *last_used_cycles*

FIFO : This is a clever modification of the LRU which evicts lines as per *first-in-first-out policy.* We simply modify LRU to not update the clock cycle whenever we access a cache line. This can be done by dropping the second condition in !hit || access_typetype != access_type::WRITE. An alternate approach would be to use:

```
map<CACHE*, queue<uint64_t>> clock_cycle_tracker;
```

but that is needlessly complicated.

LFU : This is another modification of the base LRU code, where we maintain another

```
map<CACHE*, vector<uint64_t>> cache_line_frequencies;
```

corresponding to the CACHE* pointers. If there is a hit we increment the frequency and if there is a miss we find the minimum amongst the frequencies, evict it, replace it and set the new frequency to 0.

BIP : This is again a straightforward manipulation of the LRU policy, where if it is a hit we set the *last_used_cycle* as the *current_cycle*, i.e. basically promoting it to MRU position. Otherwise in case it is a miss, with probability ϵ we insert it into the MRU region i.e. set the *last_used_cycle* as the *current_cycle*, and with probability $1 - \epsilon$ set the *last_used_cycle* as 0 which effectively makes it LRU. This is a slightly different way of doing things, another way would be to maintain another array denoting the state of the cache line, i.e. whether it is in MRU or LRU and adjusting the logic accordingly. I tried both ways, the first approach seemed more consistent to I went with it.

The variation between traces is fairly easy to explain, since they belong to different programs which may have different types of memory accesses and traversal patterns leading to different program IPCs. We observe that different policies give different performances by looking at the Miss Rate graph, which is just the number of misses/number of accesses for each policy for a given trace. Variation between policies for a given trace are a result of pattern of memory accesses in the traces where different replacement policies are more effective than others. Regarding the speedup, BIP(e=1) and LRU have exactly the same IPCs and Miss Rates, this is because with an epsilon value of 1, we always insert as MRU and that means it functions exactly in the same way as LRU. And BIP policy improves drastically with respect to miss rate as epsilon increases for 602-trace, this is likely due to a characteristic of the trace, which has an access pattern favoring the more recently used elements, so the least recently used ones are evicted giving BIP(e=1) and LRU excellent miss rates.

2 Part 2 : Prefetcher

2.1 Graphs

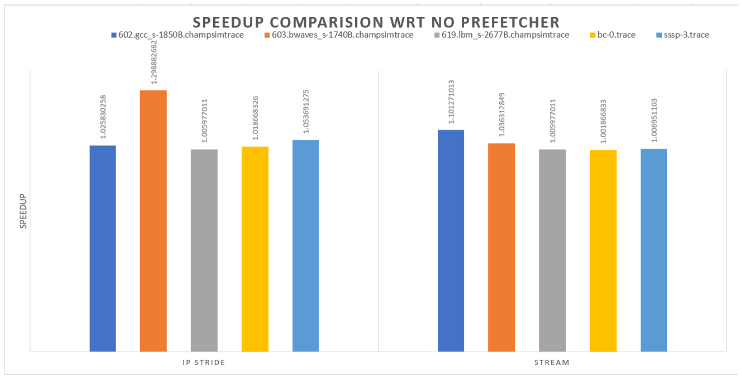

Figure 3: Speedup in IPC with respect to "No Prefetcher"

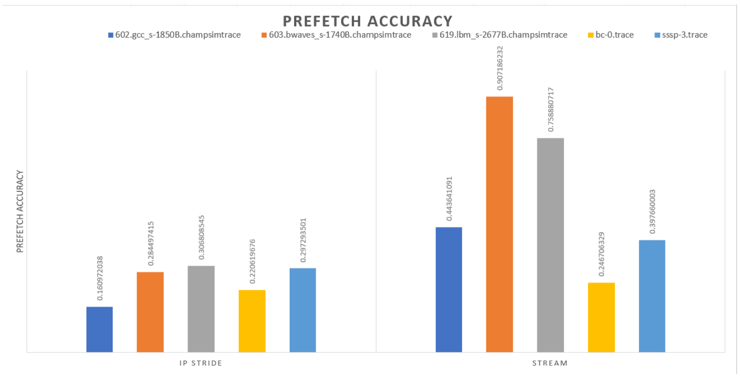

Figure 4: Prefetch Accuracy = Useful Prefetches/Issued Prefetches

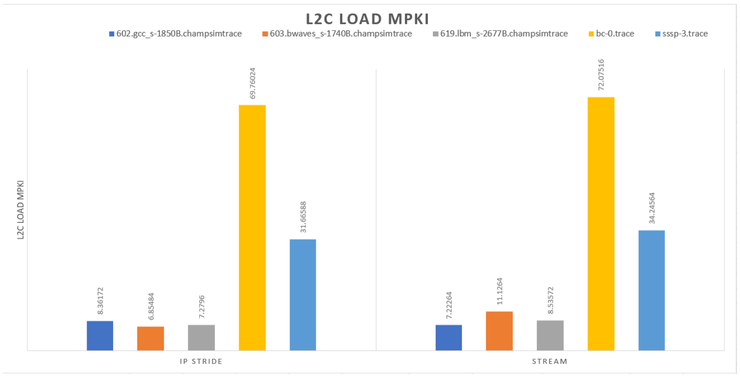

Figure 5: L2C Misses per Kilo Instructions

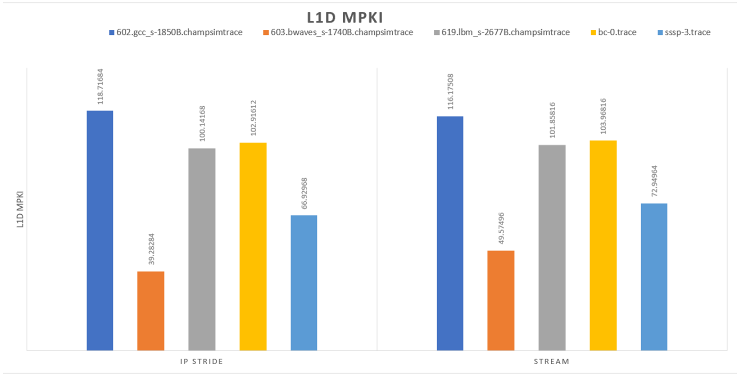

Figure 6: L1D Misses per Kilo Instruction

2.2 Optimization

I tried several things to optimize the prefetcher :

1. I deleted the forming streams in case there was any miss in them for any incoming address while traversing through the monitoring region.

2. I traversed through the monitoring table in reverse order so that I check the MRU hits first, before checking the forming streams, this way I can issue more prefetches and get more useful prefetches.

Once I was satisfied with the overall functioning of my Prefetcher, I played around a little bit with degrees and distances. I observed that on decreasing the distances I got better speedup for the 602 and 603 traces(used these for testing since they are faster to run). I observed the sweet spot to be around 400 for the *PREFETCH_DISTANCE*. I didn't end up changing the degree since increasing it impacted my prefetch accuracy, since it lead to a drop in Accuracy with no apprarent increase in speedup with respect to "no prefetcher". Ultimately I ended up choosing *PREFETCH_DISTANCE* as 400 and *PREFETCH_DEGREE* as 1 which seemed to provide the best balance betweeen prefetcher accuracy and speedup. The graphs for the same are provided above.