A study of caching and prefetching and their interactions

Seminar Report
by

Sumon Nath
(21Q050007)

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

8 May 2022

Table of Contents

List of Figures

Introduction

Improvements in memory speeds has been staggering over the past few decades compared to rapid growth of CPU performance as shown in figure 1.1. This has led to a huge gap in cpu and memory speeds. Memory operations issued by the cpu which needs to go to the memory incurs approximately 100x penalty compared to a cpu cycle and significantly degrades the overall system performance. Numerous, latency hiding techniques has been used to minimize this gap like Out-of-order and speculative execution, multi-threading, caching and prefetching, to name a few. The infamous **3 level cache hierarchy** is another technique which reduces high memory latency by storing a subset of the memory in a smaller and faster memory, known as cache, with the expectation that the subset will contain the most frequently used data/instructions. A hit in the cache can give 10x to 50x less memory access time depending on the level at which the hit occurs. So a high cache rate leads to high overall performance improvement. Owing to the very high speed of these structures, they are small and significantly costlier compared to DRAMs.

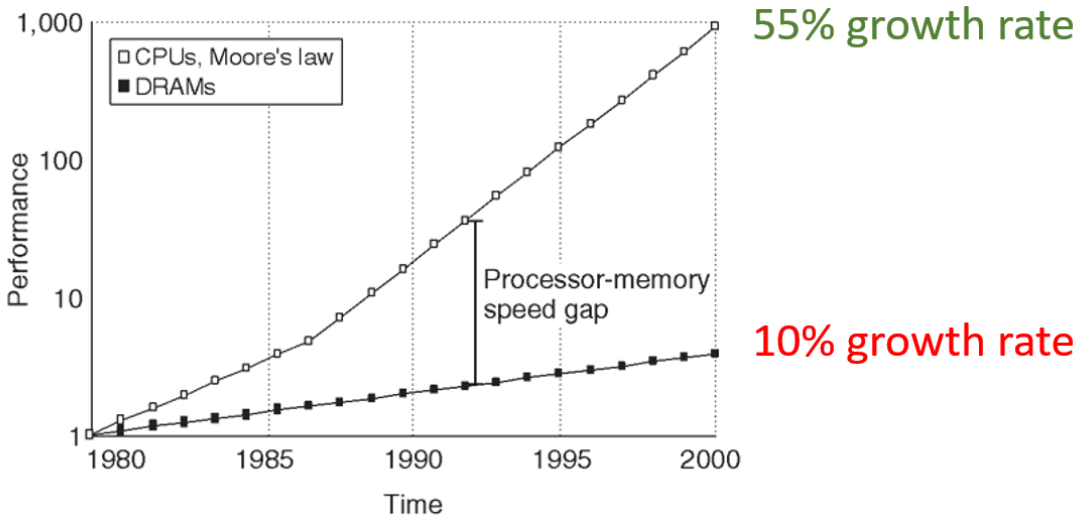

Figure 1.1: Comparison of CPU and memory performance improvement in last few decades

There are two key techniques to efficiently use the cache. First, **cache management techniques** which takes the responsibility of managing this scare resource through intelligent replacement of cache blocks, especially in the last-level cache(LLC). Second, **hardware prefetchers** which tries to push the limit of cache usabiltiy by predicting future accesses to the memory and fetching them before time into the cache. This hides the high memory latency that would have been incurred if the prefetch was not issued. Again these prefetchers mostly reside beside the level-2(L2) cache and prefetches are issued into the LLC. Cache management techniques and hardware prefetchers two orthogonal techniques used to mitigate the memory wall problem. However, as they work on the same structure which is the cache(in most cases the LLC), they can interfere with each other to impact system performance negatively. So, essentially they should work in harmony to minimize this negative interaction.

In this seminar we have studied the state-of-the-art cache management techniques and hardware prefetchers in details to understand their working, internal details and the intuition behind each method. Finally we also look at some of the recent techniques developed which tackles the problem of negative interactions between management techniques and hardware prefetchers.

The rest of the report is organized as follows. Chapter 2 introduces cache management techniques and discusses the various techniques used in the state-of-the-art techniques. Similarly in chapter 3, we discuss the various techniques employed in modern hardware prefetchers. In Chapter 4 describes the problem of interaction between the two techniques and finally we conclude in the last Chapter.

Cache management

Caches are one of the most important hardware units in processors improving performance by storing most-frequently used data and delivering them fast to the cpu. The caching technique is built on the pillar of locality of data, namely of two types, temporal and spatial locality. Unfortunately due to the high speed of caches they are restricted in terms of space. After a certain space threshold it violates the cost as well as energy budget. Hence the cache is a scarce resource and needs to be managed well to maximize its utility.

Caches are divided into sets and each set has a fixed number of cache blocks also called lines. As the caches are small in size compared to the huge size of memory, they get filled fast. The problem arises here, once a set is filled one block has to be removed from the set to make space of the incoming block. This decision is critical to performance. For example, if a frequently used block is evicted from the cache instead of a block that will never be reused in the future, first the dead block will consume valuable cache space and the frequently used block will incur a miss and hence a high memory access penalty. This is where cache management techniques come into play by intelligently deciding which blocks to evict.

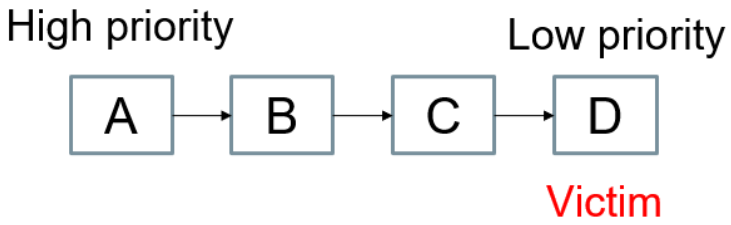

Figure 1.2: Priority chain of cache blocks for a particular cache set

Most cache management techniques creates a priority chain of cache blocks per cache set as shown in figure 1.2, the cache block at the head of the chain having the highest priority and the tail having the lowest priority. At the time of insertion when the set is full, the block with lowest priority is evicted. The key decisions that a management policy has to make is as follows:

1. At the time of **insertion** what will be the priority of the new cache block.

2. Whenever a cache block receives a hit, what will be the updated priority a cache block. That is what will be the **promotion policies**.

Next we will be looking at some of the techniques used in cache management techniques develop over the years.

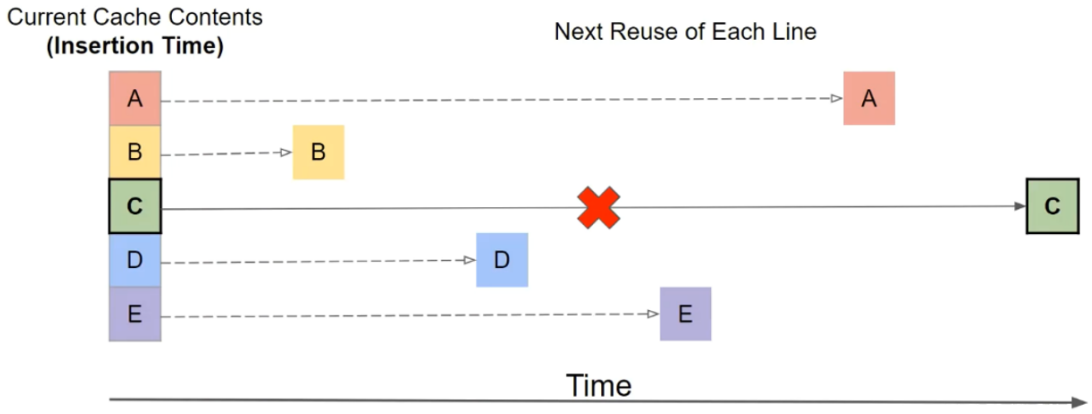

Figure 1.3: Belady's optimal policy

1.1 Textbook techniques

Belady: Most cache management techniques tries to emulate the Belady's optimal replacement policy. The Belady's optimal replacement policy says to evict the block that will be used furthest in future as shown in figure 1.3. This simple technique maximizes the utility of cache by retaining useful cache blocks in the cache and evicting blocks that will used furthest in the future. But this policy is not implementable in practice as it knowledge about the future.

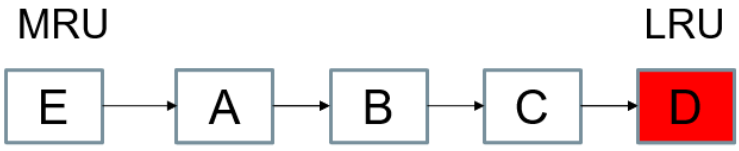

Figure 1.4: Least recently used policy

LRU: LRU is the most basic cache management technique that is widely used still now in modern systems. It uses a simple LRU priority chain, where the least recently used cache block is evicted at the time of insertion as show in figure 1.4. The key assumption made by the LRU policy is new blocks will be re-referenced soon in the future, hence it

insert an incoming cache block with the highest priority that is at the MRU position. And on a cache hit, LRU promotes a cache block to the MRU position. But the problem with LRU is, it only works for workloads that have a recency friendly access pattern as shown in figure 1.5 as well as the workload needs to fit into the cache. To tackle such scenarios Jaleel et al. proposed the Re-reference interval prediction [1] method to handle thrashing workloads as well as workloads with scans.

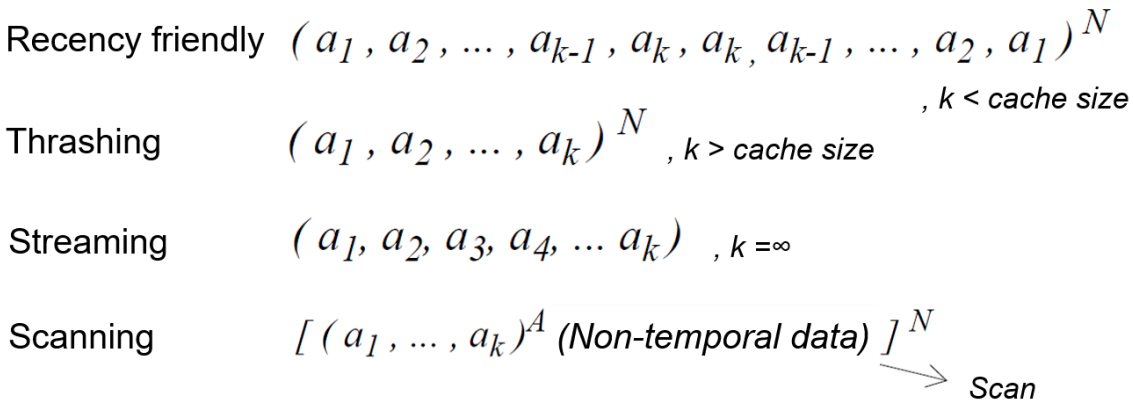

Figure 1.5: Various cache access patterns

1.2 Re-reference interval prediction

Let us understand the access patterns shown in figure 1.5 before diving into the solution. A workload with a **thrashing** access pattern has a cache friendly access pattern but cannot fit into the cache and hence thrashes the cache,i.e., evicts the cache friendly blocks. A **streaming** access pattern has no locality, as it is a stream of independent accesses without having any reuse. Workloads with a scanning access pattern as shown in figure 1.5 has a sequence of cache friendly accesses which are discarded from the cache by a sequence of non-temporal accesses which is called a scan.

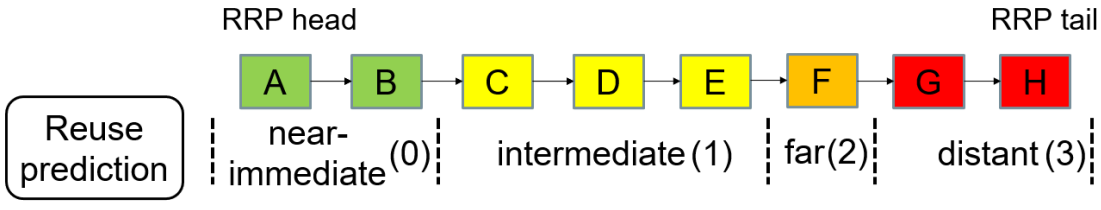

Figure 1.6: Re-reference interval chain used by RRIP technique

RRIP: Instead of using the basic LRU chain, RRIP uses a re-reference interval chain, which prioritizes cache blocks according to their predicted reuse as shown in figure 1.6. A block with a near-immediate re-use interval is predicted to be reused in the

recent future while a block with re-reference prediction value(RRPV) of 3 is predicted to be reused farthest in the future and hence it will be the one to get evicted at the time of eviction. When a block receives a hit it is promoted to an RRPV of 0. RRIP has two different variations to deal with insertions as follows:

Static-RRIP: SRRIP always inserts a block a "far" re-reference interval. Using this approach, it preserves the cache friendly blocks and discards the scans quickly. Hence SRRIP is resistant to thrashing access patterns.

Bimodal-RRIP: BRRIP insert a block with an RRPV of 2 or 3 randomly to preserve some of the blocks in case of a thrashing access pattern, so that all blocks doesn't age similarly. BRRIP protects from thrashing access patterns.

A **Set-dueling mechanism** is used to identify which insertion policy among SRRIP and BRRIP performs better for the active workload as shown in figure 1.7. This is done by dedicating a few cache sets which are tested with both the policies and selects the policy incurring fewer misses. The entire technique is termed as Dynamic-RRIP(DRRIP).

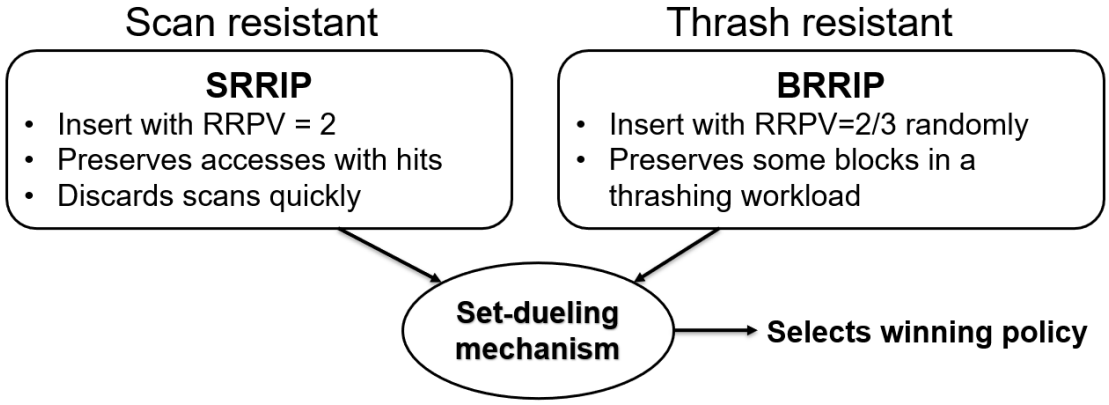

Figure 1.7: Set dueling mechanism used by DRRIP to select the better performing technique

1.3 Improving RRIP accuracy

RRIP, due to its simlistic technique is not capable of handling more complex access patterns. Wu et al. proposed SHiP[2] a policy based on RRIP which makes decisions on a finer granularity by associating signatures with each cache reference . The signature is based on a memory region , program counter or instruction sequence history. SHiP aims to predict if a cache block will get future hits or not. It maintains a bit per cache line, which indicates if ti was re-referenced or not. Also for each cache line there is a signature associated. This signature is used to update the signature history counter table(SHCT) maintained by SHiP as shown in figure 1.8. On a cache hit, the SHCT entry associated

with the signature of a cache line is incremented. SHiP decrements the SHCT entry associated with a line evicted from cache but not been re-referenced since insertion. This way SHiP ensures that cache blocks that have the same signature will have similar re-reference interval and the predictions can be reused.

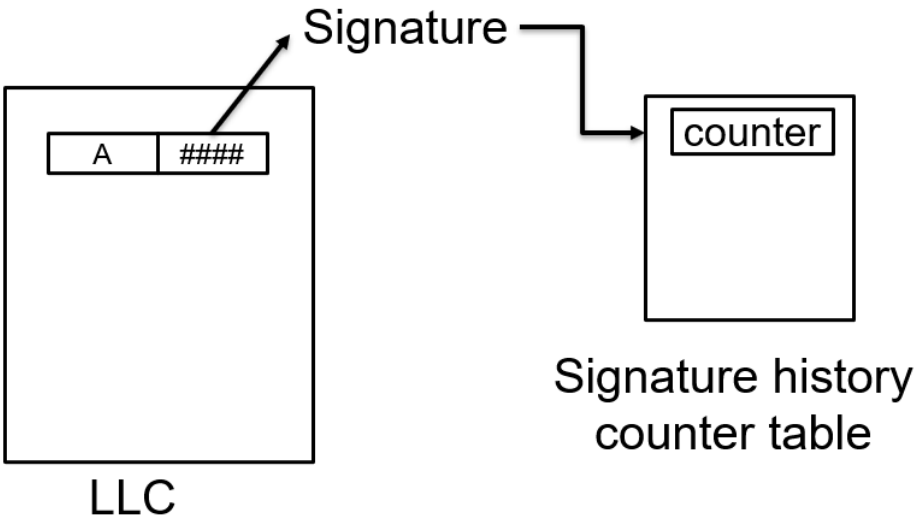

Figure 1.8: Signature based re-reference interval prediction used by SHiP

1.4 Emulating Belady's Optimal policy

In this section we will look at techniques which emulates the Belady's optimal policy as discussed earlier and are completely orthogonal to re-reference interval prediction technique which uses heuristics to improve their prediction. Hawkeye [4] was the first cache management technique to successfully emulate the optimal policy. Figure 1.9 shows the basic overview of the cache management technique. It uses a component called the OPTgen which efficiently emulates the optimal policy on a long history of cache accesses for a few sample set and predicts a cache block to be cache friendly or cache averse. The hawkeye predictor stores these predictions on a per PC basis and uses these predictions to influence the insertion priority a cache block. Essentially the hawkeye technique performs a **binary classification** of the incoming cache blocks into cache friendly or averse lines. But the problem with binary classification is that small mis-prediction causes a global impact.

Mockingjay [5] solves the problem of binary classification by adopting an Estimated time of arrival(ETA) based approach. It is similar to the Hawkeye technique and emulates the Belady's optimal policy on past cache accesses and tries to predict the ETA of future accesses. Thus instead of binary classification of cache blocks into friendly and averse lines,

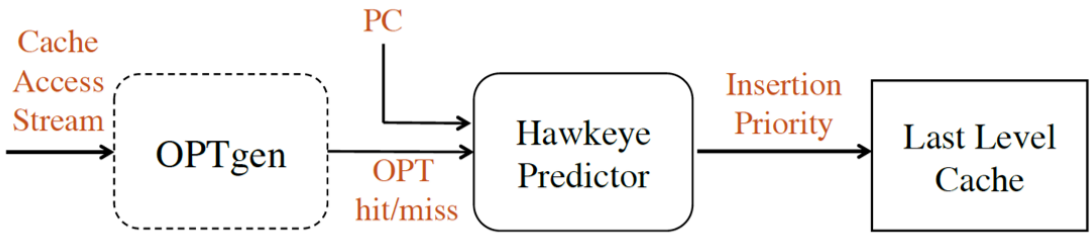

Figure 1.9: Overview of the Hawkeye cache management technique

Mockingjay performs a multi-class classification based on ETAs. It evicts the line that is predicted to be reused furthest in the future. Multi-class classification is more resilient to prediction errors and has a small impact on the relative ordering. Also a mis-prediction often does not lead to an ordering error as the relative order is preserved even though the ETA is mis-predicted.

1.5 Summary

Figure 1.10 shows all the cache management techniques discussed so far list down some of the state-of-the-art proposals that uses these techniques. Starting from the textbook techniques to more complex techniques based on RRIP to recently proposed techniques which emulates Belady's optimal policy, the complexity of the techniques increases and so does the performance improvements.

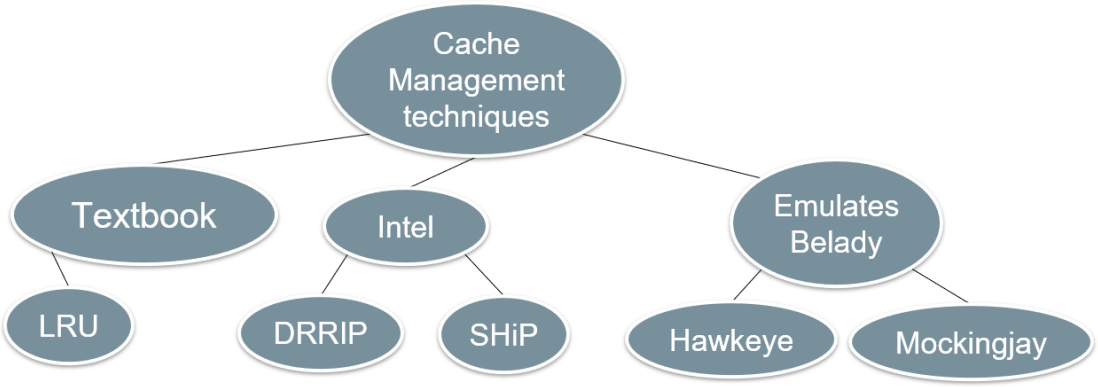

Figure 1.10: Cache management techniques

1.6 Performance improvement

Figure 1.11 shows the performance improvements with the state-of-the-art cache management techniques for single and multi-core(4 core) systems, where the baseline uses the

LRU policy. The workloads used are a mix of SPEC2006, SPEC2017 and GAP benchmarks.

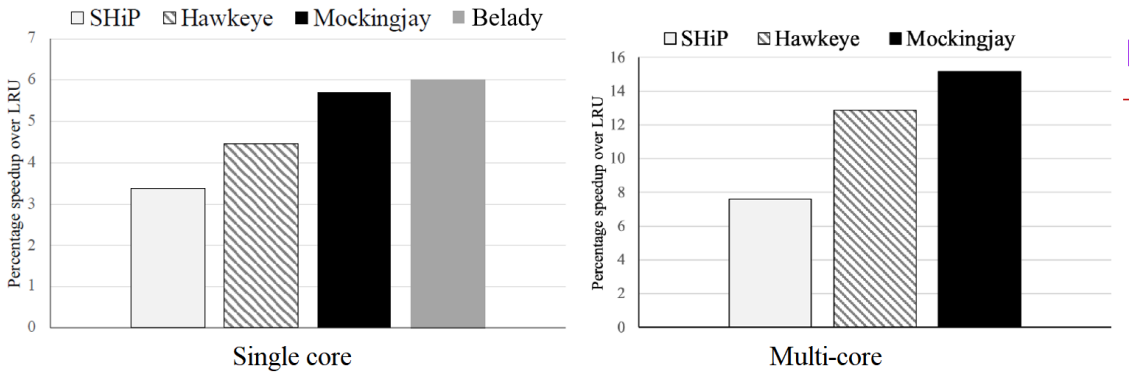

Figure 1.11: Performance improvement for cache management techniques in single core and multi-core systems

Cache prefetchers

Cache prefetching is a technique to mitigate expensive off chip memory latency by predicting future memory accesses and fetching those that are not in the cache before the processor demands them. In particular we are interested in domain of prefetchers called Hardware prefetchers which as the name suggest are fully implemented in hardware. Hardware prefetchers observes load/store access patterns and prefetches data based on past access behavior. Let's take a practical example to understand hardware prefetching in details. Figure 2.1 shows a simplistic view a system. 1 denotes the cache access pattern by the cpu. The prefetcher tries to learn this pattern and predicts future cache accesses. 2 indicates the cache access predicted by the prefetcher and this prefetch request is issued to memory ahead of time idicated by 3 and the data is prefetched into the cache as shown by 4. When the cpu actually issues the request as shown in the figure by the number 5, it gets a cache hit instead of a cache miss. This improves performance significantly.

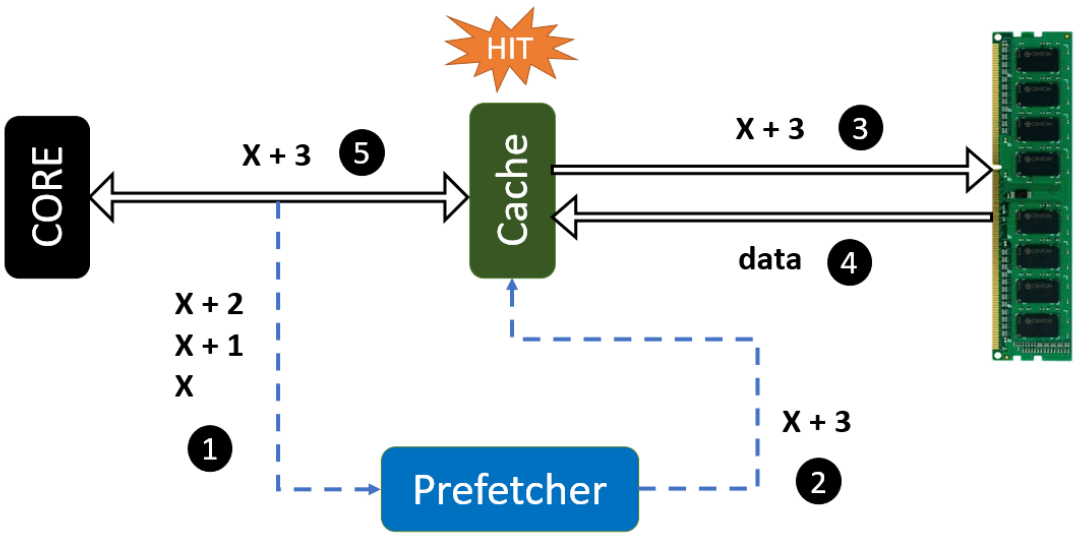

Figure 2.1: Basic flow of hardware prefetching

2.1 Metrics of interests

Following are some of the metrics of interest of prefetcher performance and are used in later sections:

$$Accuracy = useful\ prefetches/total\ prefetches \tag{2.1}$$

$$Coverage = Misses\ eliminated\ by\ prefetching/total\ misses \tag{2.2}$$

$$Lateness = Late\ prefetches/total\ prefetches \tag{2.3}$$

2.2 Prefetcher aggressiveness

To define prefetcher aggressiveness, we need to define a few terms. The prefetch distance is the distance of the farthest prefetch from demand access. Prefetch degree is the number of prefetches issued per demand access. An aggressive prefetcher is the one with a high prefetch distance as well as high prefetch degree. As the aggressiveness of a prefetcher increases, the coverage increases but at the same time the accuracy is usually low which leads to cache pollution. Also high aggressiveness leads to a higher memory bandwidth consumption. On the other side a conservative prefetcher with a low prefetch degree and distance usually has higher accuracy which leads to lower cache pollution as well as lower memory bandwidth consumption. But the problem is they usually have lower coverage.

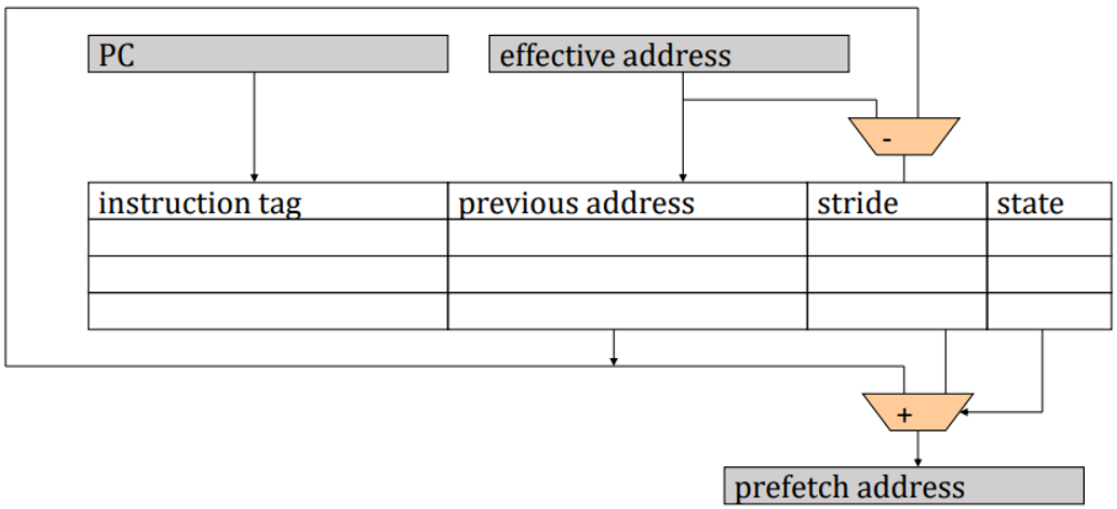

Figure 2.2: Simplistic model of the IP-stride prefetcher used in modern cpus

2.3 Stride prefetching

Stride based prefetching is a simple prefetching technique where the prefetcher tries to learn constant strides in cache accesses. IP-stride prefetching is an example of such

prefetchers which associates a stride with a particular PC. As show in figure 2.2 this prefetcher tries to learn the stride by subtracting the current address with previous address associated with the particular program counter. As a particular stride occurs for the same PC, the confidence is incremented. When confidence reaches a threshold, prefetches are issued for the particular PC with the learned stride.

2.4 Correlation based prefetchers

Recent prefetching techniques attempt to predict complex access patterns but are limited by prefetching depth. However, increasing depth naively may hinder coverage and accuracy.Previous works exploit the look-ahead mechanism for complex access patterns but suffer from high hardware complexity. Most of the prefetchers work in the physical address space and are incapable of handling access patterns that cross page boundaries. SPP [3] uses a look-ahead based prefetching technique using a signature that stores access patterns in a compressed form as shown in figure 2.3. The signature is also used to detect locality between two physical pages and is able to speculate access patterns crossing page boundaries.

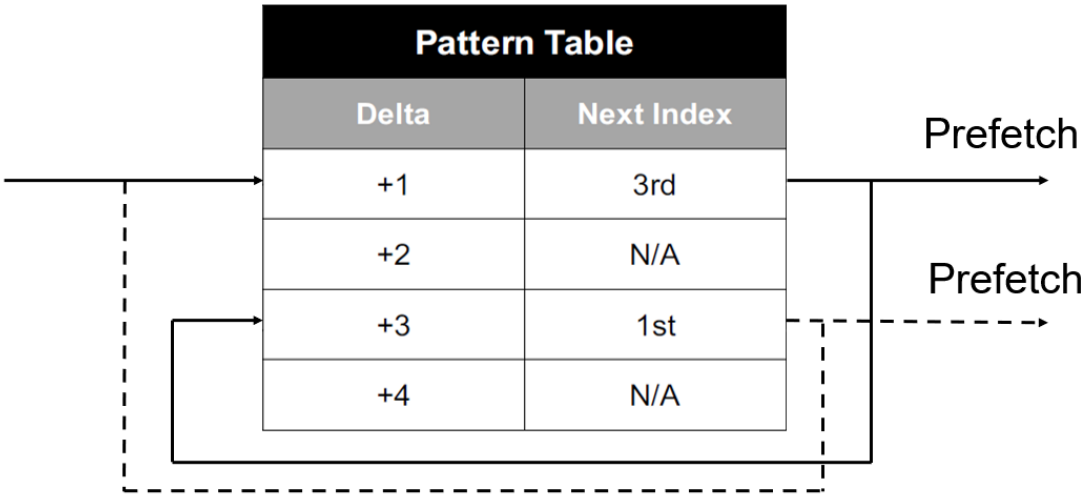

Figure 2.3: Pattern history table employed in look-ahead based hardware prefetchers

2.5 Offset-based prefetchers

Best-offset prefetcher [8] is a stride-based prefetcher which attempts to find the optimal stride between accesses unlike signature based approaches which are fine grained, BOP tries to construct a global view of the access pattern & based on that makes predictions.

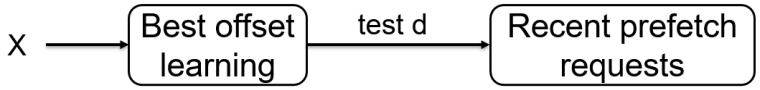

Figure 2.4: Best offset prefetcher mechanism

2.6 Region based prefetchers

Spatial data prefetching exploit the recurrence of access patterns over memory regions to prefetch future memory references. Existing prefetchers associate observed access patterns to either short events with high probability of recurrence or long events with low probability of recurrence, which leads to low accuracy or very less prediction opportunity. Bingo [7] solves this problem by taking into account both short and long events to achieve high accuracy with minimum loss of prediction opportunities.

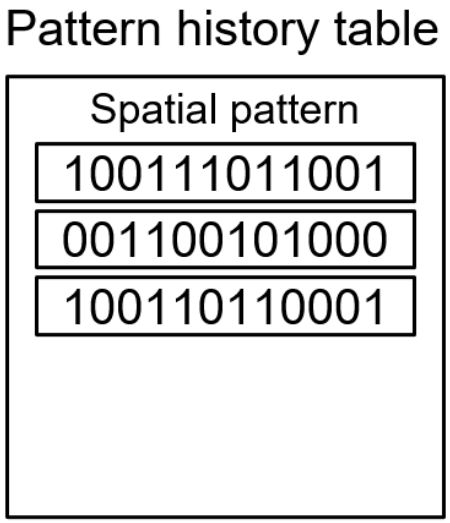

Figure 2.5: Pattern history table to store spatial access patterns in certain memory regions

2.7 Performance improvement

Figure 2.6 shows the performance improvements with the state-of-the-art hardware prefetchers for single systems, where the baseline is without any prefetching. The results are based on some of the memory intensive workloads from the SPEC2017 benchmark

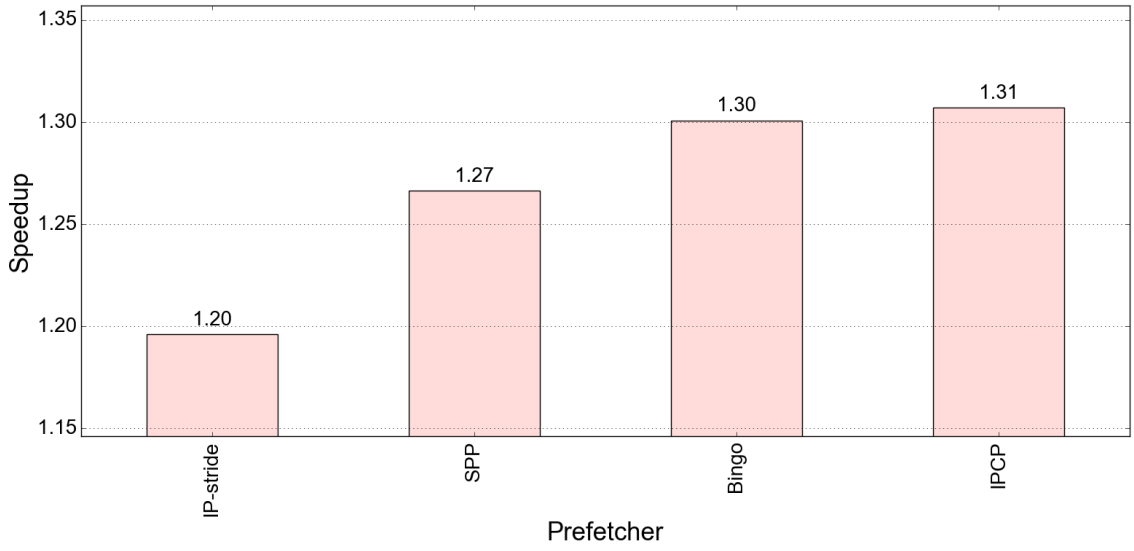

Figure 2.6: Performance improvement for hardware prefetchers

Interactions

Cache management techniques and hardware prefetchers two orthogonal techniques used to mitigate the memory wall problem. However, as they work on the same structure which is the cache(in most cases the LLC), they can interfere with each other to impact system performance negatively as show in figure 3.1. So, essentially they should work in together to minimize this negative interaction. Some of the techniques developed so far to tackle

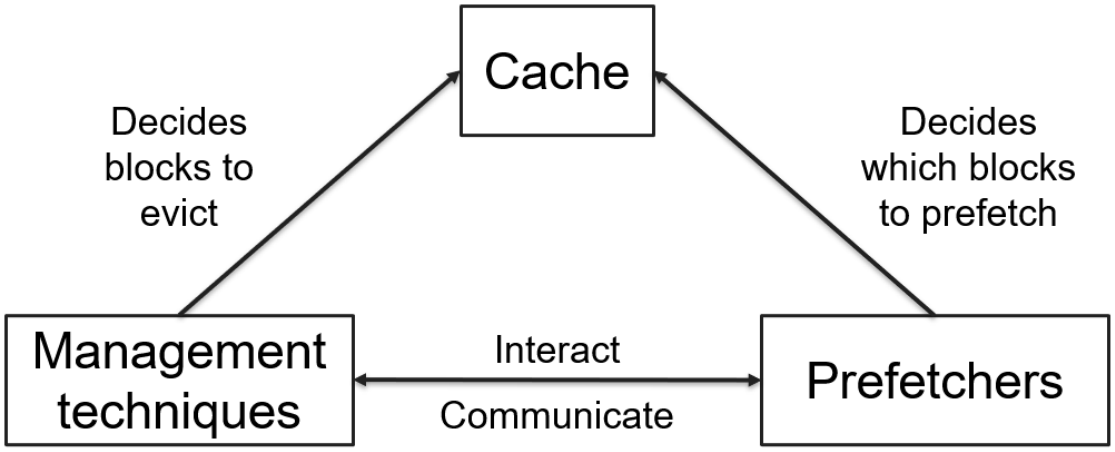

Figure 3.1: Interaction between cache management techniques and hardware prefetchers

this problem are:

PACMaN[11] Modifies the RRIP cache management technique to accommodate prefetching. It modifies the insertion policy for prefetches, where it inserts prefetches with a distant RRPV always. It also modifies the promotion policy for prefetches by not updating the RRPV on prefetch hits.

Harmony[12] Redefines Belady's optimal policy to accommodate prefetching. Belady's policy evicts block that will be used furthest in the future. Harmony proposes a new optimal policy called DEMAND-MIN which evicts the block that will be prefetched furthest in the future. It emulates the DEMAND-MIN policy using the hawkeye predictor.

KPC[13] , kill the program counter is a cooperative cache management and prefetch-

ing scheme. It uses feedback on prefetch accuracy to determine usefulness of prefetches and does Cache management based on dead block prediction. But the problem with this approach is that it is rigid and difficult to integrate with other cache management and prefetching techniques.

Conclusion & future work

LLC management techniques are key to performance and have almost reached optimal performance threshold.Hardware prefetchers are ubiquitous in modern systems and improves performance by many folds. There is an urgent need to study the interaction between the two to further improve performance, bandwidth, and energy consumption for single and multi-core systems.

Acknowledgements

I am grateful to **Prof. Biswabandan Panda** for guiding me and helping me throughout my seminar work. I would also like to extend my gratitude towards my peers who have helped me in every possible way.

Sumon Nath
IIT Bombay
8 May 2022

References

[1] Jaleel, Aamer, Kevin B. Theobald, Simon C. Steely Jr, and Joel Emer. "High performance cache replacement using re-reference interval prediction (RRIP)." ACM SIGARCH Computer Architecture News 38, no. 3 (2010): 60-71.

[2] Wu, Carole-Jean, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr, and Joel Emer. "SHiP: Signature-based hit predictor for high performance caching." In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 430-441. 2011.

[3] Kim, Jinchun, Seth H. Pugsley, Paul V. Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. "Path confidence based lookahead prefetching." In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1-12. IEEE, 2016.

[4] Jain, Akanksha, and Calvin Lin. "Back to the future: Leveraging Belady's algorithm for improved cache replacement." In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 78-89. IEEE, 2016.

[5] Shah, Ishan, Akanksha Jain, and Calvin Lin. "Effective Mimicry of Belady's MIN Policy."

[6] Bera, Rahul, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. "Dspatch: Dual spatial pattern prefetcher." In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 531-544. 2019.

[7] Bakhshalipour, Mohammad, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. "Bingo spatial data prefetcher." In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 399-411. IEEE, 2019.

[8] Michaud, Pierre. "Best-offset hardware prefetching." In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 469-480. IEEE, 2016.

[9] Somogyi, Stephen, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. "Spatial memory streaming." ACM SIGARCH Computer Architecture News 34, no. 2 (2006): 252-263.

[10] Pakalapati, Samuel, and Biswabandan Panda. "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching." In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 118-131. IEEE, 2020.

[11] Wu, Carole-Jean, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr, and Joel Emer. "PACMan: prefetch-aware cache management for high performance caching." In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 442-453. 2011.

[12] Jain, Akanksha, and Calvin Lin. "Rethinking belady's algorithm to accommodate prefetching." In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 110-123. IEEE, 2018.

[13] Kim, Jinchun, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy." ACM SIGPLAN Notices 52, no. 4 (2017): 737-749.