Microarchitecture interactions in multicore systems

R&D-II 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

In the past few decades processor performance has been doubling every year , on the other hand the memory(DRAM) performance improvement is only about 15% per year which has led to huge gap in the processor-memory speeds. This leads to high latency instructions whenever an instruction goes to the memory to load/store data or instruction. This is termed as the "Memory wall" problem which has been bugging the computer architecture community ever since.
A plethora of techniques has been proposed and are in use in modern processors to mitigate this problem, essentially by hiding the high latency required to complete a memory operation. Following are some of the techniques:

- **Superscalar and Out-of-order processors** can hide the high latency incurred by an instruction involving a memory operation(load/store), by executing multiple instructions in parallel. This is achieved by executing independent instructions out of order, hence utilizing the idle pipeline while it is stalled executing a long latency memory bound instruction. This seemingly simple technique is overly complex to implement in the hardware due to the dependencies involved between instructions.

- 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.

- As the cache is limited in size, we need to decide which blocks to keep and which to evict when a new block is to be inserted in the cache. This is where the **cache replacement policies** come into play and plays a crucial role in improving cache hit rate and in turn performance.

- **Cache prefetcing** is another complimentary technique to replacement policies used to improve average memory access time by fetching instructions/data from memory to cache ahead of the time of its actual usage.

In this report we will be focusing mainly on the state-of-the-art cache replacement policies and prefetching techniques.

1.1 Cache replacement policy

Belady defined the optimal replacement policy as the one that replaces the cache block that will be used furthest in the future. Hence, this algorithm is not possible to implement practically as it requires knowledge about the future cache access pattern. Ideally we want replacement policies that can make decisions as close to the Belady's optimal policy. Following are some of the techniques that attempt to get near Belady's optimality.

1.1.1 Sub-optimal approach

The simplest replacement policy widely used in modern systems is the Least Recently Used *LRU* policy. As the name suggests, LRU evicts the least recently used cache block at the time of a insertion. But it performs poorly in case of applications with irregular or thrashing access patterns.

1.1.2 Re-reference interval

Re-reference interval prediction chain can be used instead of the LRU chain in a replacement policy. The block at the head of the chain is predicted to have near-immediate re-reference interval while block at the tail represents a distant re-reference interval. As discussed, the problem with LRU is that, it always predicts a near-immediate re-reference interval, which might not work for all applications. // Static-RRIP[1] & Dynamic-RRIP[1] are two techniques which are scan-resistant and scan & thrash resistant which uses RRIP to improve performance over LRU.

1.1.3 Improving Re-reference interval prediction

It is seen that there is a lot of scope of improvement for these policies based on RRIP. Due the the simple techniques used, the predictions are course grained. SHip[2] proposes a policy based on RRIP which makes decisions on a finer granularity by associating signatures with each cache reference . The signature is based on memory region , program counter ad instruction sequence history.

1.1.4 Belady's simulation

Another orthogonal idea to RRIP is to simulate the behavior of Belady's policy by learning the decisions made by Belady on past accesses. Hawkeye [3] is one of the techniques that does the same.

1.2 Cache prefetching

Cache prefetching is one of the more modern techniques to mitigate the memory wall problem, which essentially predicts future accesses by learning from the access pattern history and prefetches the blocks as shown in figure 1. With a high prefetch accuracy prefetchers can provide significant can be done at the software or hardware level, although we will be focusing on hardware prefetchers as they are more prevalent.

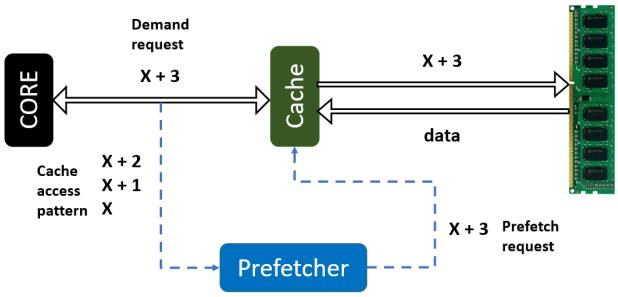

Figure 1.1: Cache hardware prefetching overview

1.2.1 Simple techniques

Next-line prefetcher is the simplest prefetchers used, which basically prefetches the next block ahead of time.
IP-stride is another simplistic prefetcher which learns the cache access stride & prefetches next cache blocks which are at a distance equal to the learned stride.

1.2.2 Signature based

Similar to the replacement policies that uses signatures to associate with the RRP value, prefetchers like SPP [4], DSPatch [5], Bingo [6] uses signature based on Instruction pointer, memory region, etc. to improve prefetcher accuracy.

1.2.3 Offset based

Best-offset prefetcher [7] 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.

1.2.4 Memory streaming

SMS[8] learns the access pattern in a specific memory region and associates it with a trigger PC. Once the trigger PC is activated, all the addresses corresponding to memory region are prefetched. This exploits the spatial correlation between PCs and memory regions.

1.2.5 Hybrid approach

IPCP [9] is one of recent prefetchers that propose a hybrid solution by integrating the previously discussed prefetching techniques. Essentially it classifies IPs into three categories based on its access patterns. IPCP leverages the fact that instruction pointers can be classified into a few classes according to their access patterns. It outperforms state-of-the art prefetchers except in a few exceptional cases.

Proposed method

We define a cache interaction, at the time of a cache eviction, in terms of the number of hits the evicted block will get compared to the number of hits the incoming block will get. Using this notion, we categorize the interactions as positive, negative and neutral interactions. For example, a cache block with 0 hits evicted by a cache block with 2 hits is identified as a positive interaction. The intuition behind this is, a useful cache block that will get 2 hits evicting a useless cache block that will not get any future hit is a desired situation with respect to the cache. The useless cache block would have consumed valuable cache space in case it was not evicted and still be resident in the cache. Similarly a useful cache block evicted by a useless cache block would consume valuable cache space as well as convert at least one hit to a miss for the useful cache block. We identify such an interaction as a negative interaction.

As mentioned before, we are concerned about how cache management techniques and cache prefetchers interact with each other. So to capture this, we only take into account interactions where one of the blocks,i.e., either the incoming or the evicted block is a prefetched block. To be specific, we are only concerned about the following interactions:

- C evicts P
- P evicts P
- P evicts C

where P denotes a prefetched block which has not yet been demanded by the core and C denotes a cache block which has been brought in response to a demand request into the cache.

We argue that these interactions provide a good estimate of the actual interaction between cache management techniques and cache prefetchers for two reasons, (1) the category of an interaction depends on the decision taken by the cache prefetcher as we consider interactions with at-least one prefetched block and (2) the category of an interaction also depends on the decisions made by the cache managements technique it is the one taking

all the eviction decisions. We are not concerned about the case **C evicts C** as it does not include prefetches and hence does not contribute to the overall interaction between cache management techniques and cache prefetchers. We will be looking into each of these scenarios in detail to characterize the interactions between prefetcher and cache replacement policies.

P evicts C

1. P evicts a dead C, where P gets a hit later. A dead block is nothing but a useless block. This can be characterized as a positive interaction as a useless C block is replaced by a useful P block.

2. P evicts a non-dead C, where P gets a hit later. As C is non-dead it will be refilled into the cache at a later point of time. If C evicts the aforementioned P before it is used, then the whole interaction will be negative. On the other hand if C evicts the aforementioned P after P is used, then the whole interaction is neutral.

3. P evicts a non-dead C, where P does not get a hit later. This can be characterized as a negative interaction as a useful block is replaced by a useless block.

4. P evicts a dead C, where P does not a get a hit later. This interaction is neutral.

C evicts P

1. A dead C evicts P, where P gets a hit later. This can be characterized as a negative interaction as a useless C block replaces a a useful P block.

2. A dead C evicts P, where P does not get a hit later. This interaction is neutral.

3. A non-dead C evicts P, where P gets a hit later. This can be characterized as a neutral interaction.

4. A non-dead C evicts P, where P does not get a hit later. This can be characterized as a positive interaction as a useful C block replaces a a useless P block.

P_1 evicts P_2

We assume P_1 as the prefetch block that is filled into the cache and P_2 as the prefetch block that is evicted from the cache.

1. P_1 gets a hit later and P_2 does not get a hit later. This interaction is positive.

2. P_1 gets a hit later and P_2 also gets a hit later. This interaction is neutral.

3. P_1 does not get a hit later and P_2 also does not get a hit later. This interaction is also neutral.

4. P_1 does not get a hit later and P_2 gets a hit later. This interaction is negative.

In the next section we propose two techniques to implement the functionality of categorizing interactions into the different classes as discussed. As mentioned earlier, we need knowledge about future hits of blocks to classify each interaction as positive, negative or neutral. We first used a predictor based approach to estimate if a block would get a future hit or not and based on the prediction, categorized an interaction. But this approach is coarse grained as depends on heuristics to predict the future and in most workloads with varying memory access pattern, the predictions turn out to be incorrect. So we propose another method which tracks the future accesses of cache blocks associated with an interaction. This method gives near perfect estimation of interactions as it uses knowledge of the future cache access.

2.1 Predictor based

Two cache blocks are associated with a single interaction, the one which is evicted and the one which is incoming. To correctly categorize an interaction we need to predict the future hits of the two cache blocks. For the one that is evicted, future hits mean the number hits it would have got if it would be still in the cache. And for the incoming cache block, naturally it is just the number of cache hits it gets. However, due to the limitations of the predictor method we can only predict if a block will get a future hit or not rather than the number of future hits. This results in some of the positive or negative interactions to be predicted as a neutral interaction.
We use a global hysteresis based dead block predictor as it is simple to implement and dynamically adapts to the dynamic program behavior. We make use of 3-bit global hysteresis counter. The counter is incremented whenever a cache block is evicted from the LLC without receiving any hits, and decremented whenever a cache block receives a hit. At the time of prediction if the counter is saturated, the block is predicted to be a dead block. To categorize an interaction we use this global hysteresis counter to make the prediction at the time of an eviction.

2.2 Future access based

In this approach we use a table to store all the interactions, i.e., the evictions entries as shown in figure 2.5. Interactions are classified only until a certain epoch is reached, where

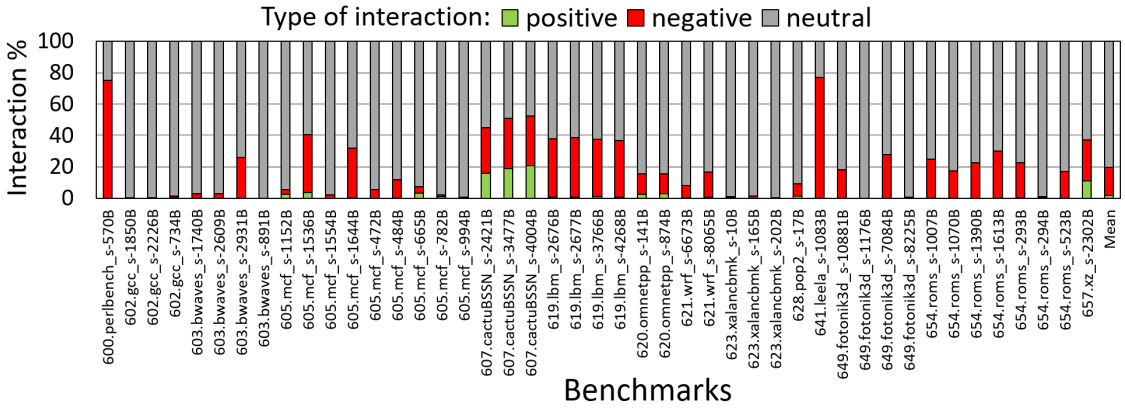

Figure 2.1: Percentage of interactions for IPCP and Hawkeye using predictor method

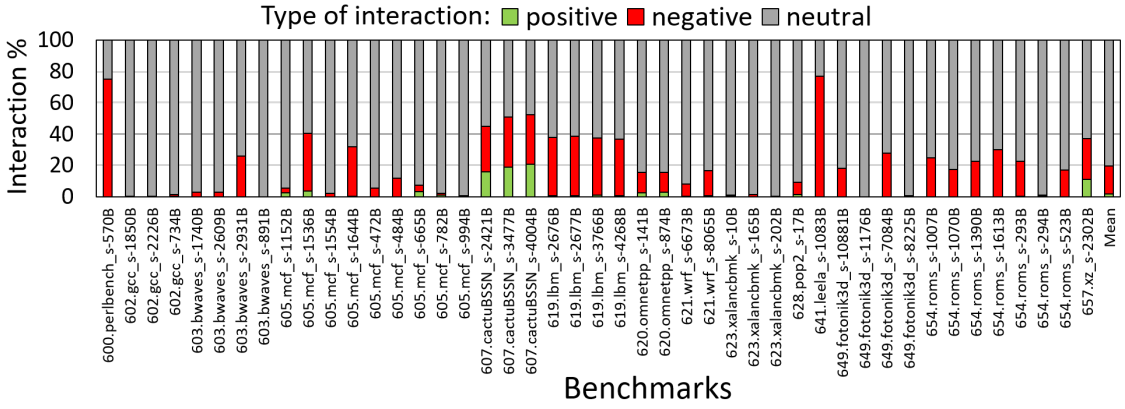

Figure 2.2: Percentage of interactions for SPP and SHiP++ using predictor method

epoch size is defined in terms of the number of cache evictions. For our experiments we set the epoch size to 2 times the cache size which is enough to track the future hits of the cache blocks involved in an interaction. We also store accesses to all cache blocks made within the epoch to track the future hits of cache blocks involved in an interaction. When an epoch is reached the table is parsed in a bottom-up fashion where each interaction is categorized using the hit count of respective cache blocks which is available from the cache access information stored in the table. Unfortunately, the fact that this method requires knowledge about the future makes it an impractical approach. Although impractical, the data generated using this method is accurate and we use it to measure the ideal performance improvement for the case where we remove all the negative interactions. Although it is impossible to remove all the negative interactions in a practical system, the ideal performance improvement gives us a upper bound for achievable performance improvement. In the next section we will look into the details of the model used to measure the ideal performance improvement.

Eviction/Access	Incoming Line	Type	Evicted Line	Type
Eviction	X	Prefetch	Y	Demand
Access	Y			
Access	Y			
Eviction	Z	Demand	W	Prefetch

Figure 2.3: Table used for future access based method

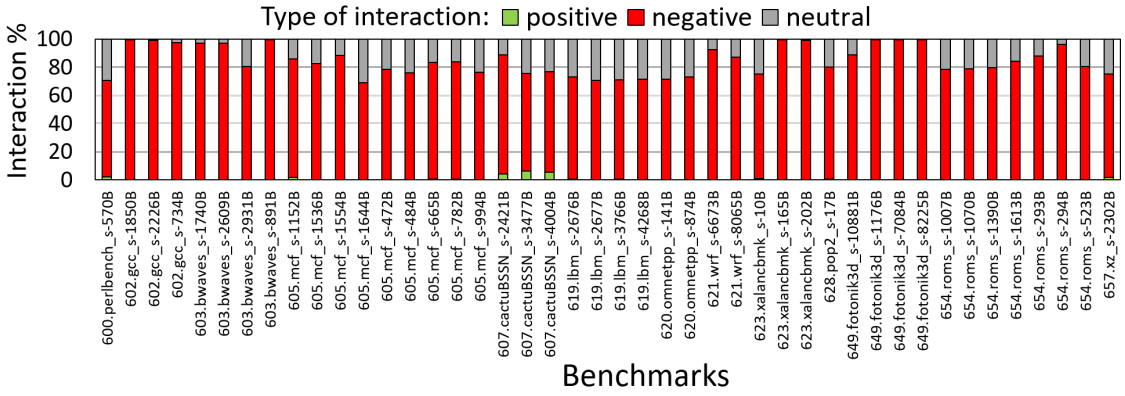

Figure 2.4: Percentage of interactions for IPCP and Hawkeye using future access based method

2.3 Ideal performance improvement

In this section we will measure the ideal performance improvement possible when we remove all negative interactions. We call it ideal as it is near to impossible to eliminate all the negative interactions in a real system. However, this will give an upper limit of the maximum performance improvement possible. We propose a mathematical model to find the performance improvement on eliminating negative interactions. From the perspective of LLC, the negative interaction hog the DRAM bandwidth, which introduce an additional latency on every DRAM access. Let the additional delay introduced by the negative interactions the negative interactions per DRAM access be P. Thus every load request from the CPU that goes to the DRAM suffers an additional latency of P. And the load requests that go to the DRAM are nothing but the loads that miss in the LLC. We define the additional latency per LLC Miss (P) as follows:

$$P = avg.\ miss\ latency\ with\ \ prefetcher - avg.\ miss\ latency\ without\ prefetcher \quad (2.1)$$

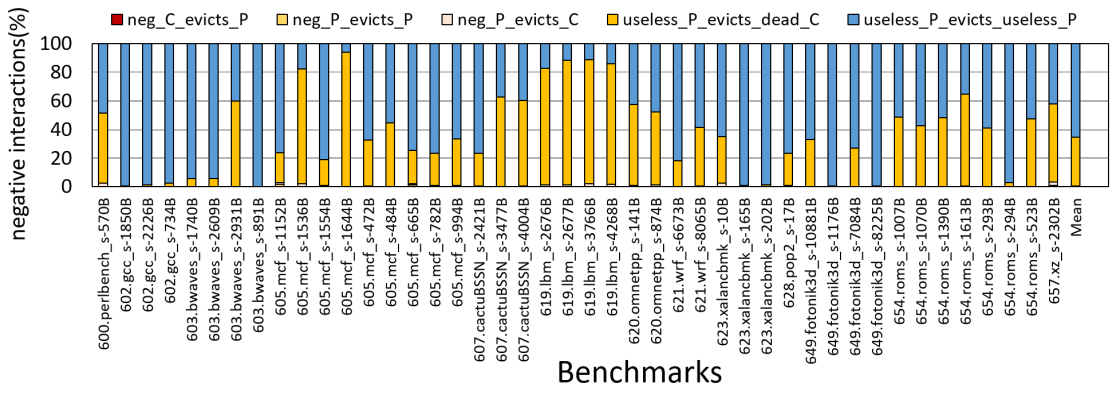

Figure 2.5: Breakdown of negative interactions for IPCP and Hawkeye using future access based method

But in addition to this, we need to keep two additional things in mind. First, as we are dealing with an out-of-order processor, multiple load instructions can be waiting for a memory operation to complete in parallel. Secondly, with the support of Miss status holding registers(MSHR) multiple load instructions can miss at the LLC and wait for the miss to get resolved in parallel. So, considering there are multiple LLC misses from multiple instructions waiting in the MSHR, the additional penalty will amortized over all the loads active in the MSHR. We use the average MSHR occupancy as a proxy for the average number of loads active in the MSHR. Thus, we can define the additional latency per LLC miss (P) as:

$$P = \frac{avg.\ miss\ latency\ with\ \ prefetcher - avg.\ miss\ latency\ without\ prefetcher}{avg.\ MSHR\ occupancy} \tag{2.2}$$

Let F be the fraction of LLC misses that suffers from this penalty, so we can define F as:

$$F = num\ of\ LLC\ misses \times fraction\ of\ negative\ interactions \tag{2.3}$$

Using these quantities F and P we can easily find the improved metric instructions per clock cycle(IPC) as follows:

$$IPC(no\ negative\ interaction) = \frac{number\ of\ instructions}{cpu\ cycles - F \times P} \tag{2.4}$$

2.4 Evaluation

In this section we will be looking into the overall performance improvement for a fixed prefetcher with varying cache management techniques at the LLC. The baseline for all cases is a system with LRU cache management technique without any prefetching. Figure 2.6 and 2.7 shows the performance improvement for SPEC2017 benchmarks and figure 2.8 and 2.9 shows the performance improvement for GAP benchmarks.

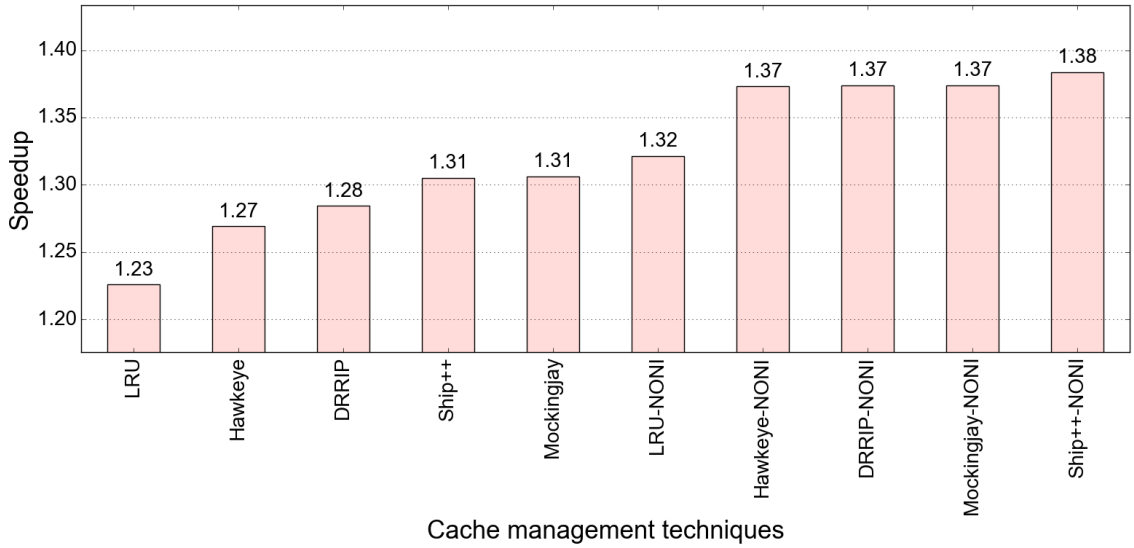

Figure 2.6: Performance improvement without negative interactions for IPCP prefetcher with varying management techniques on SPEC2017 Benchmarks

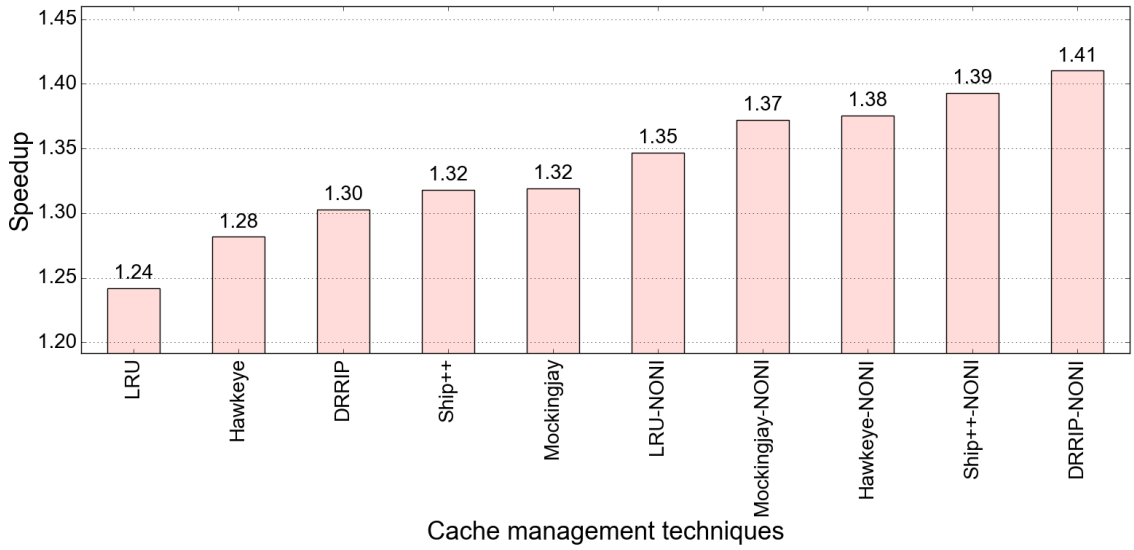

Figure 2.7: Performance improvement without negative interactions for IP-stride and Next Line prefetcher with varying management techniques on SPEC2017 Benchmarks

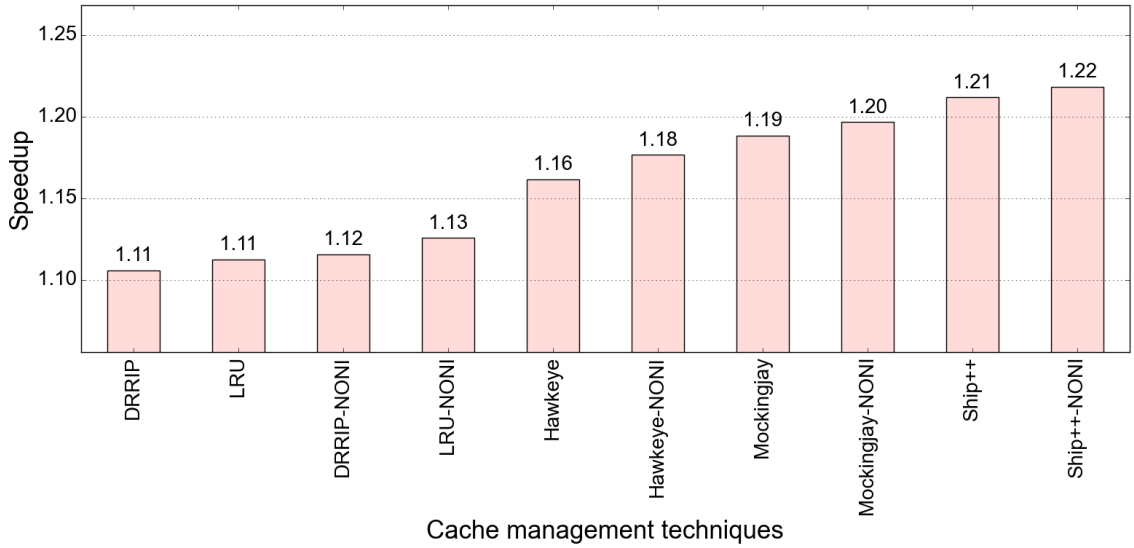

Figure 2.8: Performance improvement without negative interactions for SPP prefetcher with varying management techniques on GAP Benchmarks

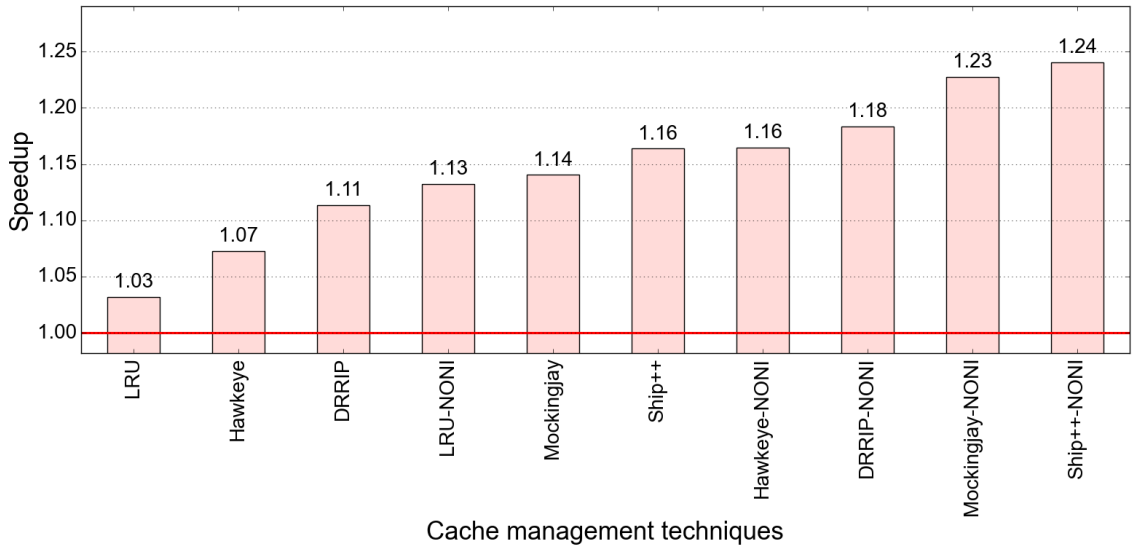

Figure 2.9: Performance improvement without negative interactions for IP-stride and Next Line prefetcher with varying management techniques on GAP Benchmarks

Conclusion & future work

We observe high performance improvement on eliminating negative interaction. Although we have used an ideal model that is not implementable in practice, it gives us an upper bound for the maximum performance improvement achievable. Next we will be focusing on ideas and techniques to achieve performance improvement as close to the ideal case.

Acknowledgements

I am grateful to **Prof. Biswabandan Panda** for guiding me and helping me throughout my R&D work. I would also like to thank my peer **Jayati** for all her contributions in the work.

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] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.