2018 SETIF – V Semana de Tecnologia da Informação – ISSN 2526-1924 – Instituto Federal do Paraná (IFPR)

Evaluation of a Mechanism to Reduce Cache Pollution

Arthur Mittmann Krause[1], **Francis Birck Moreira**[1],
Eduardo Henrique Molina da Cruz[2], **Philippe Olivier Alexandre Navaux**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{amkrause,fbmoreira,navaux}@inf.ufrgs.br

[2]Campus Paranavaí – Instituto Federal do Paraná (IFPR)
Rua José Felipe Tequinha, 1400 – CEP 87703-536 – Paranavaí – PR – Brazil

eduardo.cruz@ifpr.edu.br

Resumo. *Cache e prefetch são mecanismos empregados para esconder a alta latência associada com a leitura de dados da memória principal. O prefetcher pode prejudicar o desempenho quando polui a memória cache com seus dados. Vários autores sugerem alterar a política de inserção e remoção da cache para dar prioridade aos blocos mais importantes e diminuir a poluição. Nós avaliamos uma dessas técnicas através de simulação e identificamos como eles melhoram e como pioram o desempenho.*

Abstract. *Caching and prefetching are employed to hide the high latency associated with reading data from main memory. The prefetcher can deteriorate performance when it pollutes the cache with its data. Many authors suggest changing the cache insertion and replacement policy to give priority to more useful blocks and diminish pollution. We evaluate one of these techniques through simulation and identify how they improve and how they degrade performance.*

1. Introduction

The speed of CPUs is evolving at a much faster pace than that of the memory. First with increasing clock speeds, and now with the rapid expansions of the number of cores, the processors are demanding much more bandwidth than the memories can supply, and the trend is that it will only get worse. This is known as memory wall, when the high relative memory-to-CPU latency is such that the performance of a system is mainly determined by the time it takes to bring the data from the memory to the CPU.

To mitigate this performance limitation, techniques to hide memory latency such as CPU cache and prefetcher are employed. The cache is a small but fast memory that resides between the CPU and main memory, storing data that was recently requested by the processor, so that if it needs it again, it will not need to wait for the slow DRAM again. The prefetcher tries to predict the addresses that will be requested next by the processor, and puts them into the cache before the processor needs it, therefore hiding the latency.

There is a trade-off on memory design between speed and size. Since the CPU cache is located inside the chip, where space is very expensive, it is small and fast, thus a smart management of the data that is kept in it is essential for desirable performance. When data that would be needed by the processor is evicted from the cache in order to free space for data that will not be useful, a situation called *cache pollution* is characterized. An important source of cache pollution is the prefetcher. When it prefetches data into the cache, useful data may be displaced in favor of the prefetched blocks that are usually not as crucial for performance.

The goal of this paper is to analyze an state-of-art mechanism that reduces cache pollution. This paper provides an introduction to some necessary computer architecture concepts at Section 2, a review of the state-of-the-art on cache pollution mitigation on Section 3, and a simulation-based analysis of one of the proposed methods on Section 4.

2. Concepts

In order to properly understand the issues with cache pollution and the work that address this problem, it is necessary to have a basic knowledge of two concepts: the CPU cache and CPU prefetcher.

2.1. CPU Cache

To circumvent the problem of high latency for accessing the memory, a cache is placed between the processor execution units and the memory. This cache takes advantage of the temporal locality of memory accesses, storing blocks of addresses that were recently requested by the processor, on the premise that they have a high chance of being accessed again on the near future. Since the cache has a much faster response, the processor will not have to wait for this data to come from the slow main memory and the performance will be greatly improved.

Due to space restraints on the chip, the cache is usually organized in multiple layers on a hierarchy where the caches closer to the core are smaller and faster, and the ones closer to the main memory are bigger and slower. Most modern high performance processors employ three levels of cache, with a private Level 1 (L1) and L2 cache for each core and a L3 cache being shared among all the cores in the same processor. Each level of cache is much larger than the one under it, and one example of the load-to-use latency to each level on the memory hierarchy is presented on Table 1 [Levinthal 2009]. These values are rough approximations based on the performance analysis of a processor from 2009, and the actual values depend on many factors such as memory speeds, number of DIMMS, CPU frequencies, etc.

Table 1. Approximate latency for data access on a Xeon 5500 series processor

Location	Latency (cycles)	Latency (time)
Level 1 Cache	4 cycles	1.2 - 2.1 ns
Level 2 Cache	10 cycles	3.0 - 5.3 ns
Level 3 Cache, unshared	40 cycles	12.0 - 21.4 ns
Level 3 Cache, shared	65 cycles	19.5 - 34.8 ns
Level 3 Cache, remote	100-300 cycles	30.0 - 160.7 ns
DRAM, local		60 ns
DRAM, remote		100 ns

When the processor does not find an address in the cache, a *cache miss* occurs, and the search must be repeated at a higher level of the memory hierarchy, which implies

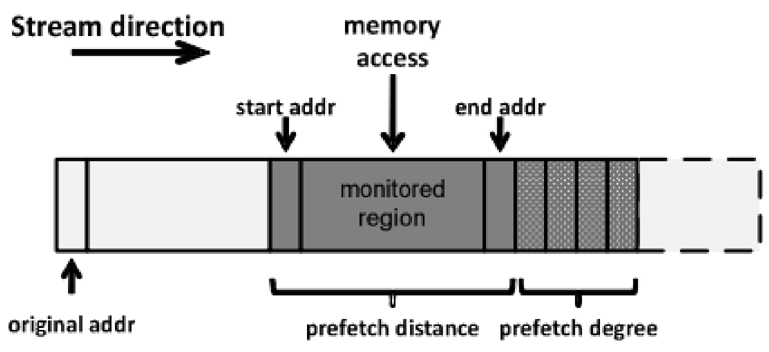

Figure 1. Basic design of a stream prefetcher [Liu et al. 2011]

in a latency penalty. When the data is found, a whole cache block (all addresses that share the same prefix, normally 64 bytes) is brought to the L1 and all the upper levels if they are inclusive caches. Since the caches have a finite capacity, when it's full, some block must be evicted in favor of the new block. The system must have a cache replacement policy, used to decide which block has the lowest probability of being reused soon. The most common approach is to use the assumption that recently used data is more likely to be reused in the near future, and thus the least recently used (LRU) block is evicted. If the higher level of cache is a non-inclusive victim cache, the block is stored into it.

2.2. Prefetcher

Another technique employed to reduce the latency of memory access is prefetching. By predicting the data that will be needed by the processor in the future, it is possible to fetch this data from the slow main memory to the fast cache before the processor demands it, effectively masking the access latency. This can be performed either by software or by hardware. Software based prefetching is normally performed through the insertion of prefetch specific instructions in the code by the compiler. Hardware based prefetching is accomplished by specific hardware mechanisms on the processor that predicts future data reads at runtime based on the application memory access patterns.

One typical implementation of the prefetcher in hardware is known as the stream prefetcher [?]. On this model, the prefetcher monitors possible streams of memory accesses starting with a cache miss, as illustrated on Figure 1. If the processor then issues more accesses to neighboring addresses in the same direction that result in misses, a stream is detected and the prefetcher starts reading the addresses from memory before the processor demands them. Two parameters define the aggressiveness of the stream prefetcher: the *degree* of the prefetcher dictates how many blocks are prefetched and the *distance* defines the region that each stream monitors for demand misses that can trigger the prefetching and how far ahead of the current stream the blocks are fetched. Another common form of hardware prefetcher is the stride prefetcher. It operates similarly to the stream prefetcher, but tries to detect patterns of accesses that are separated by a constant distance from one another.

The prefetchers can operate by monitoring the misses on any level of the cache, and bringing the blocks to either some level of the cache or to a prefetch buffer. A common approach is to have a prefetcher that tracks the misses on the L2 and inserts the prefetched blocks on the L3.

2.3. Cache Pollution

The caches are small and usually can not keep all the necessary data simultaneously. When it is full, for a new block to be stored, some other block has to be evicted. When data that is useful is evicted in order to make space for data that is useless, the system is suffering cache pollution, and the performance is compromised. This situation may happen under many circumstances, specially when one thread is causing the data from another thread to be spilled from the cache, or when the prefetcher is not being accurate and polluting the cache with useless data to the detriment of reusable blocks. This situation is aggravated because the pollution generates more cache misses and these new cache misses will trigger more prefetches, which in turn will increase the pollution even more.

Hardware prefetchers have parameters that dictate how far ahead of the current access stream it prefetches data, and how much data it brings from the memory. These parameters characterize the prefetcher aggressiveness. A prefetcher that is not aggressive enough will not yield all the potential performance it is capable of; blocks can arrive too late or do not arrive at all, effectively not delivering the performance improvement that a good prefetcher can. A prefetcher that is too aggressive can jeopardize system performance in different ways, such as: the blocks may arrive in the cache too early and by the time the processor demands it, it is no longer in the cache, or the blocks may not be needed at all, since the aggressive prefetch will be much less thoughtful about blocks that get prefetched, resulting in low prefetcher accuracy. On prefetchers that bring the blocks into the cache and not into a separate buffer, which is the case in most current high-performance processors, bad timeliness or accuracy can result in cache pollution that will degrade performance if the system is running cache-sensitive applications.

3. Related Work

There are two main approaches used to mitigate cache pollution. Some mechanisms modify the prefetcher, others change the cache insertion policy, and certain papers use a combination of both.

Jaleel et al. [Jaleel et al. 2010] propose a cache replacement policy called RRIP (Re-Reference Interval Prediction), which has a static and a dynamic variation. It is an extension of NRU (Not Recently Used), with *near-immediate* and *distant* re-reference intervals, but with intermediate values. On the static version, the blocks that are first inserted in the cache are predicted to have an *intermediate* re-reference interval, while the dynamic version uses set dueling to decide between inserting missing blocks at an *intermediate* or *distant* re-reference interval. Blocks that receive a cache hit are promoted to a *near-immediate* re-reference interval. With these modifications, the authors claim to mitigate the performance penalties from bursts of references to non-temporal data, called *scans*, which are common in many applications and cause cache pollution.

Wu et al.[Wu et al. 2011] propose a *prefetch-aware* cache management (*PACMan*) built over RRIP. On this mechanism, the re-reference interval prediction of blocks also considers if a block is on the cache due to a demand request or a prefetch. The mechanism has four versions that differ on the event that triggers it. *PACMan-M* is specially relevant because it aims to reduce cache pollution by predicting that all prefetch requests that miss on the cache have a *distant* re-reference interval. In this way, it prioritizes blocks that came from a demand request, reducing the prefetcher pollution. On *PACMan-H*, the mechanism differs from RRIP by not updating the re-reference prediction of prefetch requests that hit on the cache, also prioritizing demand requests. Another version combines the two previous mechanisms and the last uses set dueling to predict the re-reference behavior of prefetch requests at runtime.

Seshadri et al. [Seshadri et al. 2015] observe a pattern across multiple applications that prefetched blocks are used only once 95% of the time. As a result of this observation, the authors introduce a mechanism called *ICP-D*, that demotes a prefetched block to the LRU position when it receives its first demand request, predicting that it will not be reused, allegedly reducing the cache pollution generated by the prefetcher. The authors also propose an increment to the mechanism called *ICP-AP* that monitors the accuracy of each prefetch stream and modifies the cache replacement policy for prefetch requests depending on the prefetcher accuracy. When a prefetch request hits in the cache, the block is promoted to a higher priority only if it is predicted to be accurate, otherwise the block's position on the queue is left unchanged. When the request misses in the cache, the block is inserted with the highest priority only if it is predicted to be accurate; if not, it is placed at the lowest position.

The paper also suggest a filter that augments the cache, storing the addresses of predicted-inaccurate prefetched blocks that are evicted from the cache before being used, called the *Evicted-Prefetch Filter*, similar to the Evicted-Address Filter [Seshadri et al. 2012]. When there is a cache miss on a block that is present in this filter, the prefetcher accuracy is incremented. This is useful because when the prefetcher is predicted to be inaccurate, its prefetched blocks are inserted with the lowest priority and that drastically reduces the chances of it being in the cache when it receives a demand request, decreasing the predicted accuracy of the stream even though it was accurate, in a positive-feedback loop that prevents the mechanism from classifying an accurate prefetcher.

Srinath et al. [Srinath et al. 2007] propose FDP, a mechanism to tune a stream prefetcher aggressiveness based on three metrics collected at runtime: prefetch timeliness, accuracy and prefetcher-generated cache pollution. The technique defines five prefetcher configurations from *very conservative* to *very aggressive*, that are selected accordingly. FDP also changes the cache insertion policy based on the measured cache pollution. Based on two thresholds, if the prefetcher-generated cache pollution is low, the prefetched blocks are inserted in the middle position on the LRU stack; if it is medium, the blocks are placed in the last quarter of the stack, and if it is high, they are inserted in the LRU position.

Wu and Martonosi [Wu and Martonosi 2011] analyze the interference of concurrent applications on other's performance on a multicore system, characterizing the degree by which factor such as page table walks and specially prefetches influence the usage of the shared Last Level Cache. They propose to include cache blocks requested by the operating system with a lower priority, either at the end or the middle of the queue, reducing the interference of the OS on user applications. The authors also implement a prefetch manager that estimates prefetch-induced cache pollution at runtime and adjusts the aggressiveness of the hardware prefetcher accordingly.

4. Simulation

To verify the validity of the presented solutions, we performed a simulation of the most recent mechanism, ICP, which claims to outperform the previous methods.

4.1. Methodology

We implement the modifications from ICP on a x86 event-driven simulator [Seshadri 2014]. We model a single core system with a 32KB Level 1 cache, 256KB L2 and 1MB L3 cache, and a stream prefetcher that is fed with misses from the L2 cache and inserts data into the LLC. This prefetcher has aggressive parameters: its distance is 24 and degree is 4. We tried to replicate the system used in the paper, on the same simulator used by the authors. The proposed mechanisms were tested separately (ICP-D and ICP-AP) and combined (ICP) on the benchmarks from SPEC 2006 [Henning 2006], the same the authors used on the original paper. We collected traces with 200 million instructions from the most representative portions of each application [Sherwood et al. 2002]. The simulator than used the first 100 million as warm-up and collected the statistics for the last 100 million.

Table 2. Simulation parameters

Core	in-order/out-of-order x86
L1-D Cache	Private, 32KB, 2-way associative, 1 cycle latency
L2 Cache	Private, 256KB, 8-way associative, 8 cycles latency
L3 Cache	Shared, 1MB, 16-way associative
Prefetcher	16 streams/core, Degree=4, Distance=24
Memory	8 banks, 200-cycle bank access latency

4.2. Results

In this Section, we evaluate the performance and accuracy of the selected mechanisms, and compare the results to the original paper.

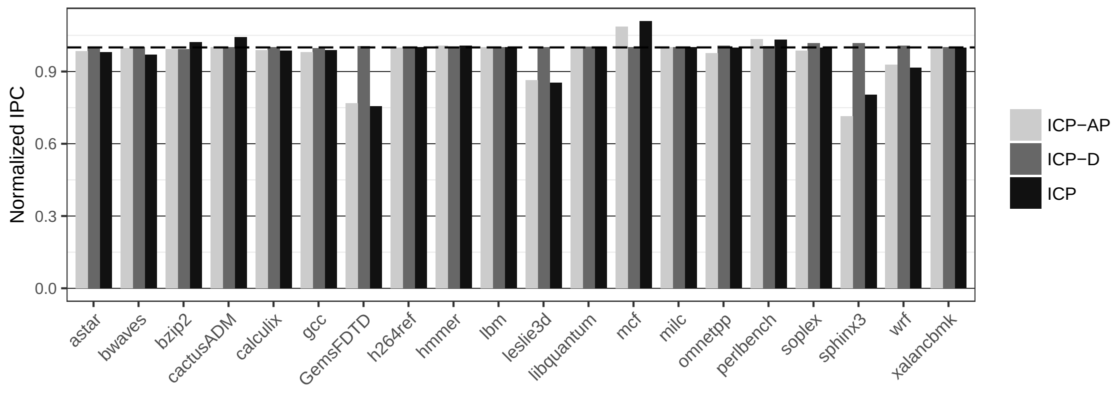

Figure 2. Effect of the different mechanisms on ICP

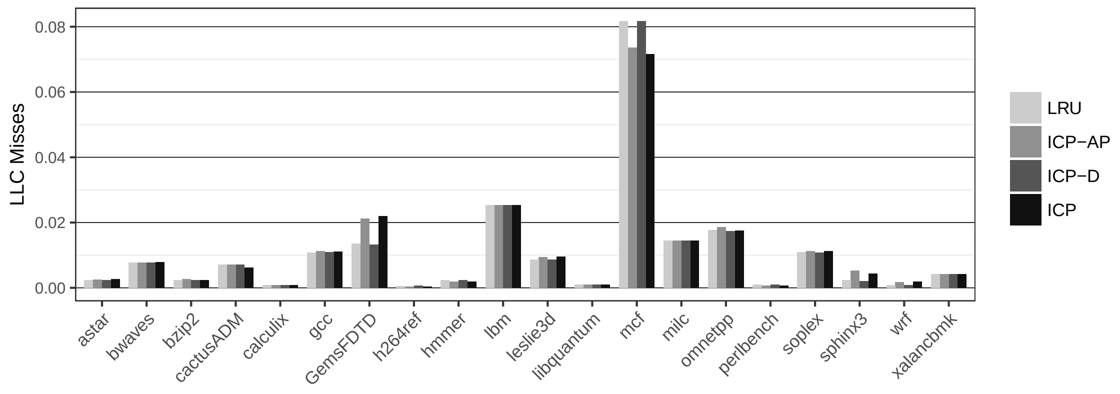

Figure 3. LLC misses under distinct mechanisms

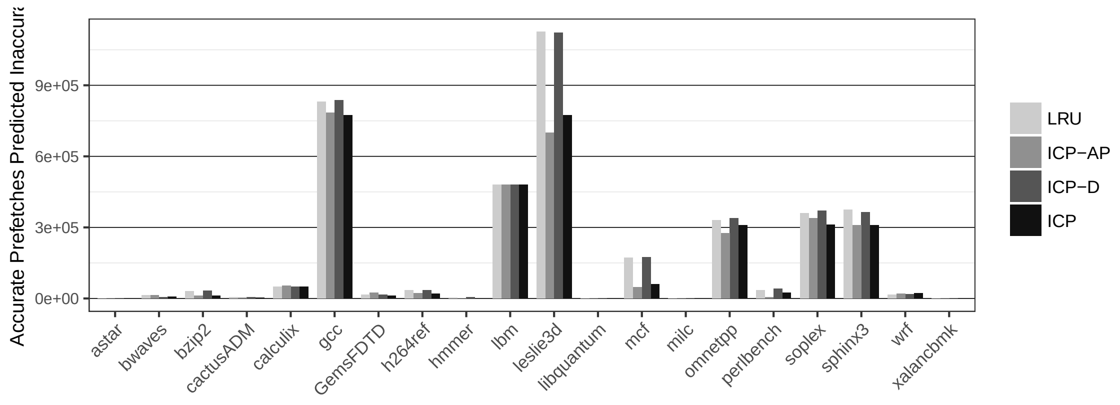

Figure 4. Accurate prefetches predicted inaccurate under different mechanisms

4.2.1. Performance

Figure 2 shows the instructions-per-cycle (IPC) of the benchmarks under the different versions of the mechanism normalized to the *baseline* configuration, where cache insertions simply follows the LRU policy. Our results show that the overall performance is degraded with ICP, with very few exceptions. On average, ICP-AP reduced performance by 4%, ICP-D improved by less than 1% and the combination of both lowered the performance by 3%. However, on *mcf*, the accuracy-aware prefetch increased the IPC by more than 8%, and in combination with ICP-D, achieved almost an 11% improvement. This can be explained by the high miss ratio for this application shown in Figure 3, where the mechanism managed to improve cache utilization by reducing the cache misses. The variation on the miss ratio also explains the performance degradations of up to 25% on *GemsFDTD*, *leslie3D*, *sphinx3* and *wrf*, where the mechanism caused the misses to increase.

4.2.2. Accuracy Prediction

Even on the implementations with no accuracy prediction, the code for generating statistics was retained for evaluation purposes, including the *Evicted-Prefetch Filter*. The graph

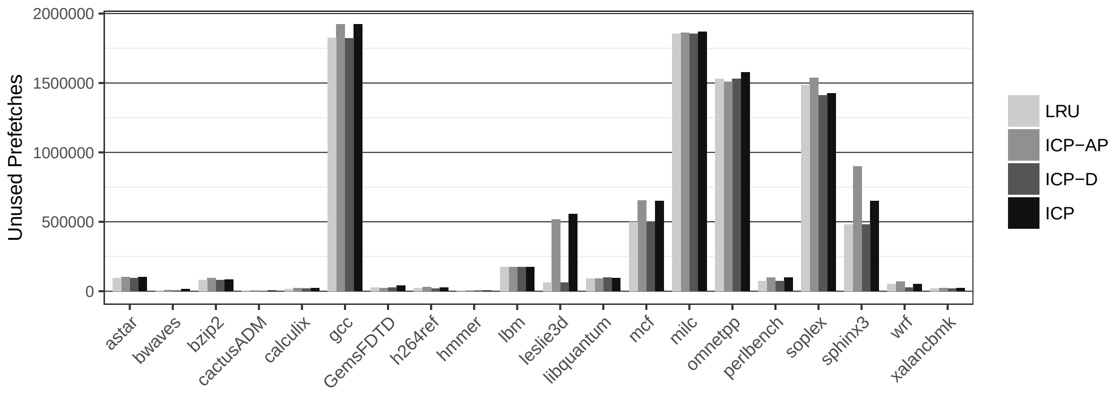

Figure 5. Unused Prefetches on different versions of ICP

on Figure 4 demonstrates the usefulness of the EPF, successfully identifying accurate prefetches that were predicted to be inaccurate and properly adjusting the prefetcher accuracy classification and reducing the occurrences of this misclassification. However, the lower priority in which the prefetchers were inserted into the cache increased its lateness and increased cache pollution on some applications as shown by the increased number of unused prefetches on Figure 5.

4.2.3. Reproducibility

The results obtained on this experiment are not consistent with the results presented on the original paper [Seshadri et al. 2015]. This can be due to multiple variables on the methodology, specially on the origin of the traces. The authors used traces of one billion instructions, while we used only 200 million. Both traces were generated using Simpoints, but the exact parameters the authors used have not been disclosed, which potentially could make the traces for the same benchmakrs drastically different. Some details of the implementation were not specified on the original paper, specially the threshold of accuracy that is used to classify a prefetcher as accurate or inaccurate, which could also have a significant impact on the final results.

5. Conclusion

Caching and prefetching are powerful techniques used to diminish the impact of high memory latencies on modern systems. One technique can jeopardize the benefits of the other when the prefetched data evicts too much useful blocks from the cache, causing cache pollution.

The state-of-the-art techniques employed to reduce cache pollution are based on changing the cache replacement policy to give more priority to demand-requested instead of less critical data that may use the cache only as a buffer between the memory and CPU, such as prefetches. Some authors suggest modifications on the hardware prefetcher to detect when it is harming performance and reduce its aggressiveness.

Our implementation of one of the most recent mechanisms shows that these techniques can improve the performance of some applications but deteriorate that of others.

As a future work, we intend to test the mechanism with different configurations, such as prefetcher aggressiveness, cache sizes and accuracy threshold.

References

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Jaleel, A., Theobald, K. B., Steely Jr, S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM.

Levinthal, D. (2009). Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18.

Liu, G., Huang, Z., Peir, J.-K., Shi, X., and Peng, L. (2011). Enhancements for accurate and timely streaming prefetcher. *J Instr Level Parallelism*, 13.

Seshadri, V. (2014). Source code for Mem-Sim. Retrieved May 30, 2018 from www.ece.cmu.edu/ safari/tools.html.

Seshadri, V., Mutlu, O., Kozuch, M. A., and Mowry, T. C. (2012). The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 355–366. ACM.

Seshadri, V., Yedkar, S., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2015). Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):51.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News*, 30(5):45–57.

Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N. (2007). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 63–74. IEEE.

Wu, C.-J., Jaleel, A., Martonosi, M., Steely Jr, S. C., and Emer, J. (2011). Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453. ACM.

Wu, C.-J. and Martonosi, M. (2011). Characterization and dynamic mitigation of intra-application cache interference. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 2–11. IEEE.