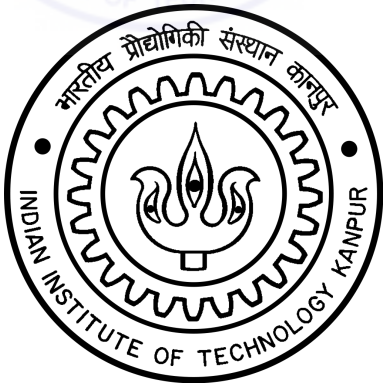
Adaptive Prefetch Filter to Mitigate Prefetcher Induced Pollution

A dissertation submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master Of Technology

by

Rahul Bera



Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
January 3, 2017

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CERTIFICATE

It is certified that the work contained in the dissertation titled "**Adaptive Prefetch Filter to Mitigate Prefetcher Induced Pollution**", by **Rahul Bera**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Mainak Chaudhuri

Dr. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

January 3, 2017

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

ABSTRACT

Name of student: **Rahul Bera** Roll no: **14111025**

Degree for which submitted: **Master Of Technology**

Department: **Computer Science & Engineering**

Title: **Adaptive Prefetch Filter to Mitigate Prefetcher Induced Pollution**

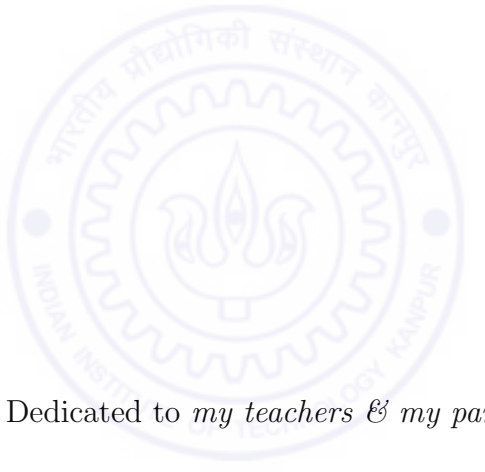
Name of Thesis Supervisor: **Dr. Mainak Chaudhuri**

Month and year of thesis submission: **January 3, 2017**

High-performance caching and hardware prefetching are two most widely used techniques to break the memory wall problem. While intelligent caching delivers performance improvement by storing a small subset of data that are most likely to be accessed again in near-future, hardware prefetcher fetches data from slow memory into fast memory before they are actually demanded, hiding long latency of accesses. However, high-performance caching in the presence of an aggressive prefetcher loses some performance benefit because of prefetcher induced pollution. A prefetcher induces pollution by fetching unnecessary or untimely data in the cache, increasing memory bandwidth overhead and cache capacity bloat.

In this work, we discuss the destructive interference between prefetching and intelligent caching. We model performance improvements coming from adding an oracle prefetch filter to the baseline architecture. Next, we propose a practical adaptive prefetch filter, which can predict usefulness of a prefetch access on the fly and selectively reject prefetch requests. Our proposal improves performance by 8.5% and 15% on average for single core and multicore heterogeneous workloads respectively with respect to the baseline without prefetching. It also reduces off-chip memory traffic by 18.8% and 8.3%, respectively with respect to the baseline with prefetcher.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR



Dedicated to *my teachers & my parents*

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my mentor, Dr. Mainak Chaudhuri. This thesis would have been incomplete without his constant support, motivation and feedback. I also want to thank Sudhanshu Shukla specially, for making me learn simulation tools and bearing with me over countless queries and discussions. Thanks to Siddharth Rai and Subarno Banerjee, two of my seniors, for their invaluable technical knowledge and tips.

I feel fortunate to have studied at Computer Science Department of IIT Kanpur. I am personally grateful to all of our professors for their compassionate teaching, and all the staffs of our department for providing us such a wonderful academic environment and computing resources.

This two years would have been very dull without my classmates and seniors. A sincere thanks to them.

And last but not the least, I thank my parents for their love and blessings. Their consistent support makes me dream higher, to become what I am today.

Rahul Bera

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Contents

List of Figures

Chapter 1

Introduction

With revamped operating frequency of microprocessors in every generation, DRAM technology lags a significant amount in terms of speed and hence, creates a performance bottleneck, which is often termed as the *memory wall*. Over the years, researchers have came up with different solutions to bridge this gap. Two of such approaches are intelligent caching and prefetching.

Caching stores a small subset of recently used data in on-chip faster memory so that the next reuse of the same data can be served quickly without going all the way down to DRAM. The caches are normally organized in a hierarchy of different speed and sizes. The last-level cache(LLC) is typically of megabytes in size and serves as the last line of defence before going to slow DRAM. A large body of research exists to manage the LLC in efficient ways, so that it can store important data which is likely to be reused in future.

Prefetching is another approach, which predicts future memory references, so that a memory access can be initiated well before it is actually needed. In other words, it simply converts a demand access to a prefetch. For example, memory references of a program, which iterates on an array fifty times, can be predicted and fetched prior to the actual use. This technique can improve system performance significantly by hiding memory latency.

Prefetcher predictions can go wrong or be too early or too late for benchmarks which have complex memory access patterns. Fetching an unnecessary data not

only occupies DRAM bandwidth, but also evicts a possibly live block from cache that could have been reused soon. The same thing happens if the prefetcher brings a block too early so that it gets replaced from the cache before seeing a demand access. This is called prefetcher induced pollution and this becomes severe for complex benchmarks which access DRAM very frequently.

Though stand-alone state-of-the-art cache management polices [3, 6, 7, 13, 22] improve performance with respect to the traditional least-recently-used (LRU) policy by large margins, the benefit reduces or even degrades sometimes if the same is evaluated in the presence of an aggressive prefetcher. This is largely due to the prefetcher induced pollution.

In this work, we explore the interference of prefetching with intelligent caching policies and determine bounds on improvements that can be achieved by filtering out all useless prefetches. We also propose an adaptive filter that is capable of predicting useless prefetch requests, and show that dropping those requests improves system performance with respect to the baseline with prefetching by up to 12.1% and 14.5% for single core and multicore system respectively.

Contributions

Briefly, the contributions of our work are as follows:

- The work presents a comprehensive evaluation of the state-of-the-art replacement policies with respect to a baseline LRU policy in absence and presence of a prefetcher, and shows that in many cases the prefetcher reduces the benefits coming from an intelligent policy, and in some cases it degrades performance.

- With offline analysis, we model the ideal improvements that can be achieved by selectively filtering out useless prefetch requests.

- We propose an adaptive prefetch filter, that can drop probable useless prefetches, saving both DRAM bandwidth and cache space and improving performance.

Organization

The rest of the thesis is organized as follows:

- In Chapter 2, we will discuss the background work related to cache replacement policies, hardware prefetchers and prefetch-aware replacement techniques.
- In Chapter 3, we will study the interference of prefetchers with intelligent caching policies. We will motivate this work by modelling an oracle prefetch filter.
- In Chapter 4, We will propose a filter which is capable of predicting re-usability of prefetched lines on the fly, and drops prefetches according to the prediction.
- Chapter 5 will discuss the simulation methodology and workloads construction details.
- Chapter 6 will evaluate our proposal with respect to the state-of-the-art policies for mitigating prefetcher-induced pollution.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 2

Background

This chapter offers a brief overview of the large body of research related to intelligent cache management policies for the last-level cache (LLC), hardware prefetchers, and existing techniques to mitigate prefetcher-caused pollution.

Cache Management Policies

The last-level cache acts as the last line of defence before accessing the considerably slower main memory, and hence, efficient management of the LLC is very critical for improving system performance. The main goal of any LLC management policy is to identify a portion of the recently used memory regions that might be reused in near-future and retain it in the LLC. The optimal replacement policy [2] chooses its replacement block that will be used furthest in the future. But in reality, as we can't have any future information, any practical replacement policy relies on a *prediction* of which block is *likely* to be accessed furthest in future.

The traditional least-recently-used (LRU) policy makes prediction based on a simple temporal heuristic: *A block that is used recently has higher chances to be used again in near-future.* It simply arranges the blocks in a recency list, where the head position points to the most recently used(MRU) block and the tail to the least recently used(LRU) one. When a block gets accessed, it is placed in the MRU position. During its cache residency it moves down toward LRU position, where

it gets selected as a victim and replaced eventually. While in most of the cases this heuristic works fine, for benchmarks having working set size bigger than the cache performs poorly with the LRU policy. In those cases, the cache blocks simply move from the MRU to the LRU position without getting any actual reuse, inefficiently using the cache space. Saving any part of the working set in this thrashing scenario can improve performance significantly. Moreover, in today's hierarchical cache memory, we have two or more levels of the smaller and faster caches closer to the CPU which cache a small subset of the memory blocks. Hence any near-term reuse gets filtered out by these levels and can't be seen by the LLC, breaking the earlier temporal heuristic. Researchers have came up with numerous proposals [3, 4, 6, 8, 13, 22] which work better for the LLC in this scenario.

The Re-reference Interval Prediction (RRIP) technique [6] assigns an n-bit re-reference prediction value(RRPV) to each cache block. A block having higher RRPV has smaller reuse probability than a block having a lower RRPV. During cache block insertion, the static RRIP policy (SRRIP) assigns an RRPV of $(2^n - 2)$, assuming a *near-immediate* reuse distance. If a block is reused, its RRPV is reset to 0, supporting the fact that this block can be accessed again shortly. At the time of replacement, it searches for a block having the highest RRPV value, i.e. $(2^n - 1)$ and selects that as a victim candidate. If none is found, it increments RRPV of all by one and continues searching. In short, SRRIP inserts every newly accessed block to the last but one position in the recency chain. To make this policy resistant to thrashing, a bi-modal flavour of RRIP (BRRIP) places majority of the blocks in the equivalent LRU position of the recency stack assigning an RRPV of $(2^n - 1)$, and the rest with RRPV $(2^n - 2)$ with low probability. While this may be beneficial for some cases, rigidly assigning RRPV is not flexible over a wide range of benchmarks, hurting performance. Hence to make it adaptive, dynamic RRIP policy (DRRIP) applies set duelling technique [13], where cache sets are partitioned into three categories. Two minor partitions(typically of size 32/64 sets) follow SRRIP and BRRIP, and a common counter monitors the relative number of misses from these two dedicated

partitions. The winning policy is applied to the remaining follower sets.

Another orthogonal approach of LLC management policy is to predict dead blocks. A sampling dead-block predictor (SDBP) [7] tries to learn the correlation between a cache block and the instruction(i.e. program counter or PC) that last touched the block. The heuristic is: if an instruction sources the last use of a cache block, all the blocks touched by the same instruction are likely to be dead too. The block predicted dead will be selected as a replacement candidate. If there's no such dead block present in cache, the baseline policy will select the victim.

Hardware Prefetchers

Prefetching is another orthogonal technique to overcome the memory wall. A prefetcher learns from the past memory access pattern and starts fetching memory regions from DRAM to cache before it is actually demanded, reducing the effective memory access time by hiding long DRAM access latency. Various types of prefetchers [5, 10, 17, 20] have been modeled which are capable of learning simple strided patterns to complex irregular patterns.

A simple example is the stream prefetcher [5], which clusters memory accesses based on small memory regions, typically a page. It tries to learn the stride present in consecutive accesses in a stream, if any. If two consecutive memory accesses touch lines A and $(A + d)$, it learns a stride of d. If the next access touches line $(A + 2d)$, the learnt stride gets confirmed, and the prefetcher starts fetching lines $(A + l * d), (A + (l + 1) * d), (A + (l + 2) * d)$, and so on. Here l is look-ahead, which is responsible for *timeliness* of prefetch decisions. If the DRAM access latency is c clock cycles and CPU generates a memory request per s cycles, then

$$l \geq \lceil \frac{c}{s} \rceil$$

Now if l is tuned to be much bigger than $\frac{c}{s}$, the prefetcher may bring some lines to the cache which will evict some other lines to make room for themselves, but will

get evicted eventually before seeing any demand accesses.

Stream prefetcher performs well for scientific applications where typical memory accesses come from iterating over big arrays of data structures. However, more complex benchmarks like commercial server or pointer-chasing applications show non-strided memory access patterns, which are not so trivial to be learnt by a stream prefetcher. Often these applications show access patterns within a large memory region, which re-occurs in other regions too. This type of recurring stream is called *spatial streams* and the spatial memory streaming (SMS) prefetcher [17] tries to learn this pattern by correlating code with the spatial streams over a region of memory. The observation is that access patterns in different memory regions triggered by the same memory operation PC will likely be the same. As an example, let's consider a region size of 64 cache lines. If the accesses to a region R_0 are triggered by PC pc_0 and touch lines $L_i, L_j, L_k, ...$, and so on, the same pattern might repeat if pc_0 triggers accesses to another region R_1 starting with line L_i.

The SMS prefetcher can learn more patterns and as a result, improves performance significantly over the stream prefetcher. However, due to its aggressiveness, the SMS prefetcher demands more DRAM bandwidth and sometimes pollutes caches severely.

Prefetch-aware Policies

Wrong or untimely prefetch decisions not only consume DRAM bandwidth, but also evict lines from the LLC which might get reused in future. Both of these side effects hurt performance. In fact, benchmarks which already show improvement in the presence of prefetchers can be improved even further, if we can mitigate this pollution. In this section, we will discuss three orthogonal techniques which aim to reduce prefetcher-induced pollution.

Last-level cache management policies take the same decision for both prefetch and demand accesses. As a result, harmful prefetched lines stay longer occupying scarce cache space. Moreover, a major fraction of the prefetched lines, irrespective

of their usefulness, never see a reuse in LLC because of filtered temporal locality by inner-level caches. Based on this observation, the prefetch aware cache management policy (PACMan) [21] actively de-prioritizes the prefetch-filled blocks over the demanded ones in the LLC. PACMan works in conjunction with a baseline policy like DRRIP, which is followed for all demand accesses. For prefetch accesses, there are four variations of PACMan

- **PACMan-M**: On a prefetch miss, it puts newly the inserted lines to the bottom of the recency stack. In other words, it treats all prefetched lines to have a distant reuse interval.

- **PACMan-H**: If a prefetched line gets a reuse, it does not bring the line to the head of the recency stack. This is largely due to the observation that a prefetch line seldom sees a reuse in the LLC.

- **PACMan-HM**: It is basically a joint flavour of PACMan-H and PACMan-M, i.e. all prefetch filled lines are installed in the LRU position, and won't be updated in the case of a reuse.

- **PACMan-DYN**: Statically fixing policies for all prefetch lines harms in cases where the prefetch lines do have multiple reuses in the LLC. Hence, PACMan-DYN uses set duelling [13] to choose the best among the competing policies $SRRIP + PACMan_H, SRRIP + PACMan_{HM}$ and $BRRIP + PACMan_H$.

Another orthogonal strategy is to tweak the DRAM controller. The state-of-the-art DRAM controllers are of two types. Either they are completely oblivious to prefetch and demand requests or they prioritize demand accesses over prefetches. The latter may sound logical, but in the case of the benchmarks which highly favour prefetching might lose opportunity to hide demand access latency. On the other hand, treating all requests equally suffers in situations where the prefetcher makes wrong prediction and unnecessarily lengthens demand access service time. In short, no static policy can lead us to the optimal result. The prefetch-aware DRAM controller [9] tracks the usefulness of the prefetcher, and dynamically switches between

the all-equal mode and the demand-first mode. If the accuracy of the prefetcher drops below a threshold, it drops prefetch requests to free up memory queue.

While the first strategy addresses LLC pollution, the second one handles DRAM bandwidth overhead caused by bad prefetches. Apart from the prefetch dropping component of the prefetch-aware DRAM controller, none of them focuses on both aspects. Aggressive prefetch filtering mechanisms [11, 23] learn prefetcher accuracy by correlating parameters like memory operation PC, address, region, or any combination of these with prefetched block usage and selectively drop prefetches. Hence, it guards caches from unnecessary prefetches as well as lowers DRAM bandwidth. Our work is similar to this strategy, where we will learn unused prefetch pattern and block prefetches that are not likely to be demanded.

Summary

To summarize, intelligent LLC management policy and hardware prefetching are two independent strategies for bridging the speed gap between core and memory. While the cache management policies deliver better performance by caching data that are most likely to be accessed again, a prefetcher fetches data from DRAM before they are actually demanded. However, high-performance caching in the presence of prefetcher loses performance because of prefetcher's pollution.

Chapter 3

Motivation

To understand the interference of hardware prefetching with caching, we evaluate different cache management polices with and without prefetcher. We model a fairly aggressive SMS prefetcher [17] and simulate it on selected workloads from the SPEC CPU2006 suite. The workload selection criteria and more about simulation configurations are discussed later in Chapter 5. Unless otherwise stated, SRRIP is considered as the baseline LLC management policy throughout the work.

Performance Impact of Prefetcher

Figure 3.1 summarizes the IPC improvement due to the prefetcher with SRRIP replacement policy with respect to LRU without prefetching. Though the selected set of benchmarks are not too prefetcher-friendly (some of them have fairly complex access patterns), still they have 10% IPC improvement on average over LRU, whereas baseline SRRIP without prefetching only improves by 3%. It clearly shows that prefetching significantly improves system performance.

Figure 3.2 shows that the improvement coming from intelligent caching reduces if we turn on the prefetcher. For the single-core benchmarks, DRRIP improves over LRU by 3.6% on average without prefetcher, whereas it improves by only 1.2% if we have prefetcher turned on. The same number for quad-core heterogeneous mixes drops from 11.5% to 6%. Though we have already seen that prefetching improves

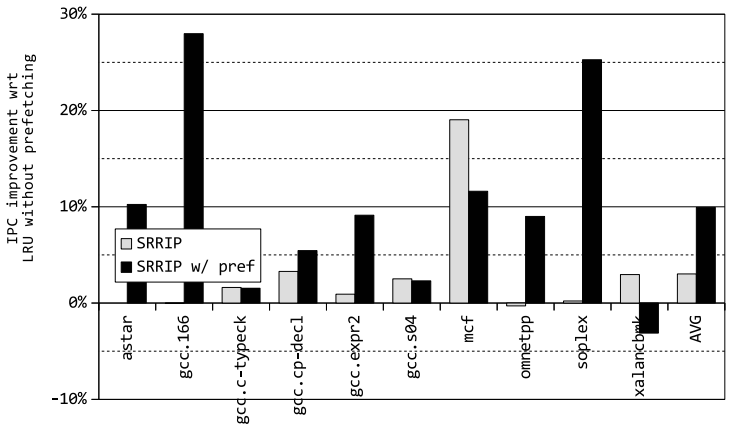


Figure 3.1: IPC improvement of SRRIP policy with and without prefetch w.r.t LRU without prefetch.

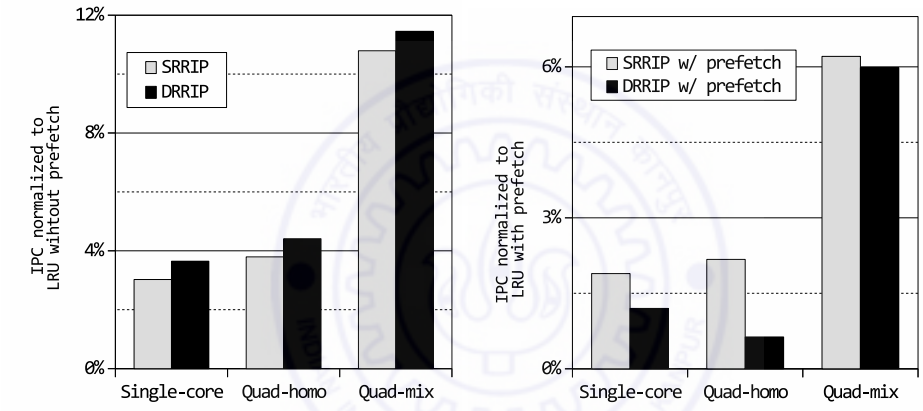


Figure 3.2: IPC performance gain in systems with and without prefetcher. Note that the Y-axis scales are different in two panels.

performance over the non-prefetching systems, the performance gain coming from better replacement policies is reduced in the presence of prefetchers. It can happen for two reasons. Either the prefetcher itself might be aggressive enough to hide most of the demand accesses, thus leaving a very narrow margin between the LRU and the optimal policy to be exploited by any intelligent caching scheme, or it is interfering with caching destructively.

To explore it further, we simulated Belady's optimal policy [2] by collecting and running benchmark traces offline. As Figure 3.3 shows SRRIP with prefetch can reduce 29% LLC misses over baseline SRRIP without prefetch, whereas OPT can reduce it by 58% on average. In fact `mcf` and `xalancbmk` are two benchmarks

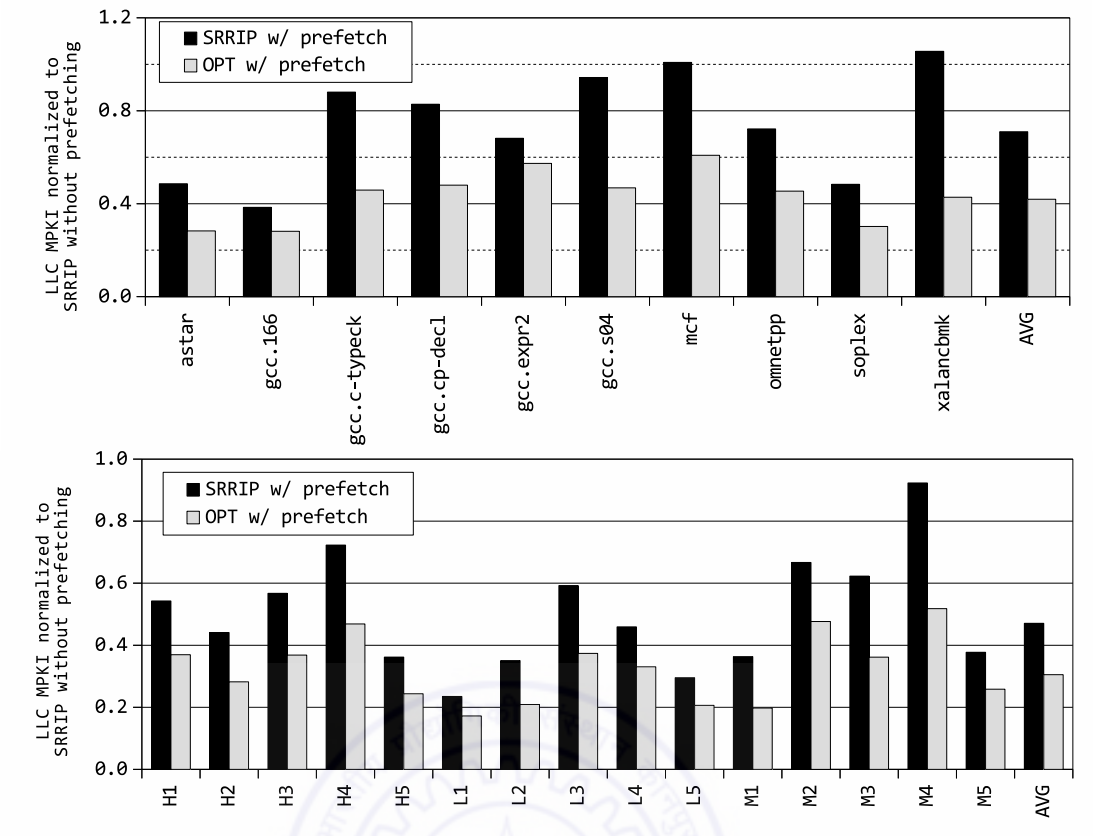


Figure 3.3: MPKI reduction in SRRIP and OPT policy with prefetcher w.r.t baseline SRRIP without prefetcher.

where MPKI *increases* by 0.8% and 5.6% respectively with prefetcher, whereas the optimal policy could have reduced it by 39.1% and 57.2%. This essentially hints at the interference caused by the prefetcher with the cache management policy.

Oracle Filter

One simple solution to mitigate prefetcher-induced pollution is to identify the prefetch accesses that are likely going to be unused before issuing the access and drop the request completely. As the reuse information requires future knowledge, we do a simple study with an oracle filter. An oracle filter is capable of distinguishing all harmful prefetches from a pool of prefetch requests using pre-computed knowledge, and drop them accordingly. Harmful prefetches are termed as those which get filled in the cache and get evicted without seeing a demand reuse during its cache lifetime.

Performance Modelling

We simulate the oracle filter in the following way. We annotate each access in the memory operation trace with its next *demand* reuse distance and bypass only those *prefetch* accesses whose reuse distance is larger than that of the blocks currently present in the targeted cacheset. As strict inclusion is maintained in the LLC with core-private caches throughout the entire work, we model a flat 2MB cache (8MB for multi-core runs) instead of a three level cache hierarchy, and the cache management policy is set to SRRIP. The prefetched blocks which get evicted from the cache without seeing any demand requests are further tracked by a small cache for premature evictions. With this setting, we make sure all prefetches that are bypassed are truly useless and would have been replaced eventually by other blocks without seeing any demand reuse. However, some potentially useless prefetches can still occupy cache space, but we will show that this number is not significant.

We will evaluate the impact of the oracle filter on the LLC misses as well as DRAM bandwidth. As this study requires future knowledge, this will only give us an estimate of performance improvement that can be expected from a practical prefetch filter.

Impact on Demand Misses

Figures 3.4 and 3.5 show the reduction in demand cache misses of single-core and multi-core heterogeneous workloads with the oracle filter over the baseline SRRIP policy. As we can see, LLC misses can be reduced by up to 19.36% and 5.57% for single-core and multicore benchmarks respectively. Heterogeneous mixes are prepared with benchmarks having different prefetchability. Some of the mixes contain highly and accurately prefetchable benchmarks like `bwaves` or `lbm`. That reduces the impact of the oracle filter on multi-core mixes compared to single-core affiliations.

Some of the single-core benchmarks have very less reduction in demand misses. This can be for two reasons. First, the SMS prefetcher is aggressive enough to

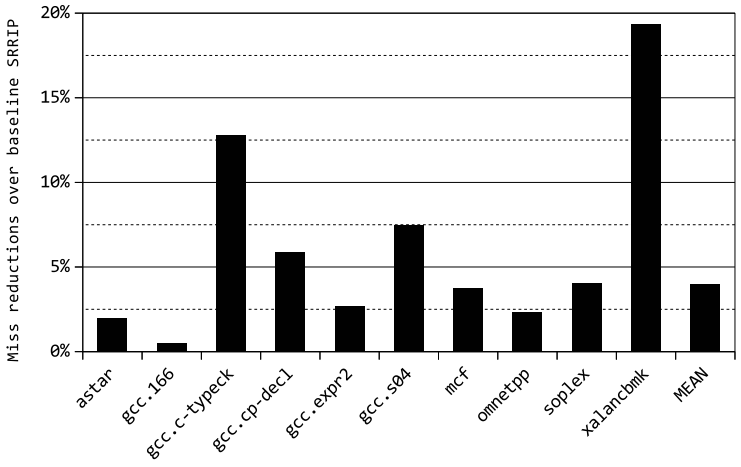


Figure 3.4: Reduction in demand misses by the oracle filter in single core workloads.

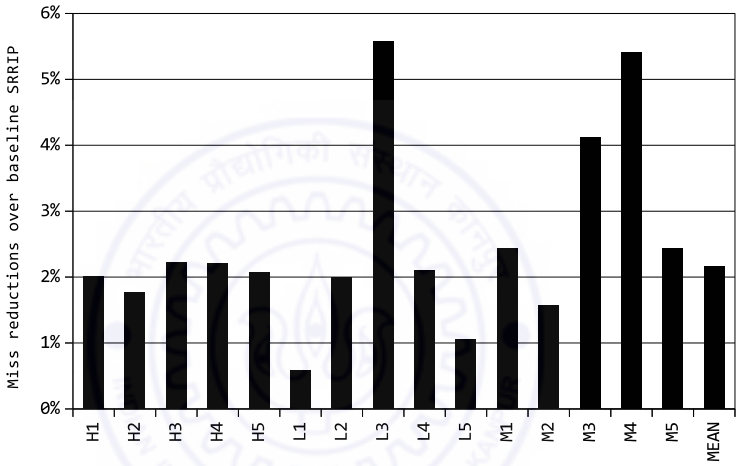


Figure 3.5: Reduction in demand misses by the oracle filter in heterogeneous mixes. H1-H5, M1-M5 and L1-L5 are the abbreviations of multi-core mixes, discussed in Chapter 5.

prefetch cache blocks, which were evicted earlier to make room for harmful prefetched blocks, *again* before getting a demand access. This only makes the prefetch volume larger, without changing the demand miss count significantly. Secondly, SRRIP already deploys a better insertion and aging policy that might do partial prefetch filtering by itself. Hence, adding an oracle isn't making a considerable difference. To support this hypothesis, we evaluated other replacement policies using the same framework with and without oracle filter.

As Figure 3.6 shows, demand miss reduction affected by the oracle filter reduces as we go from single-bit NRU to all the way up to two-bit DRRIP. In fact, the miss reduction in the optimal policy with an oracle filter won't improve at all because

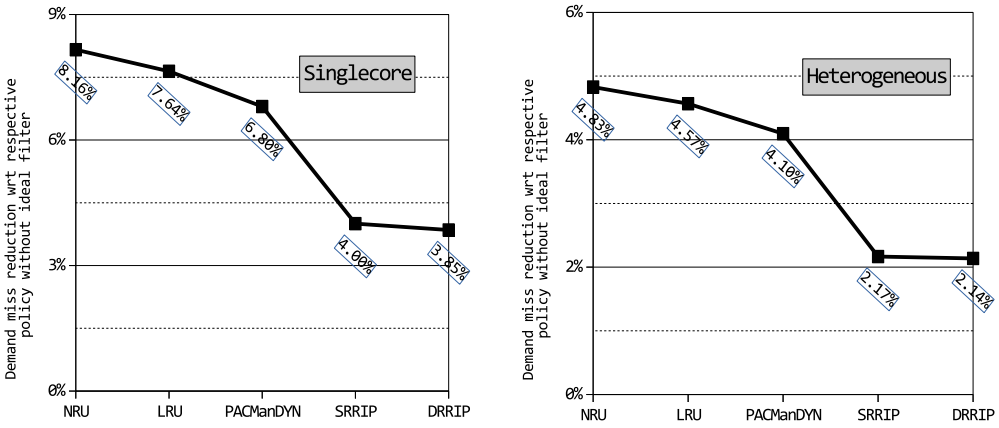


Figure 3.6: Comparison of demand miss reduction by oracle filter over different replacement policies.

optimal insertion takes ideal filtering decisions by itself. Thus, we conclude that an intelligent cache management policy has inherent filtering characteristics which protect caches from getting polluted by the unused prefetches and adding an online prefetch filter will only make small impact in miss reduction. However, filtering will reduce the overall prefetch miss volume, which will lower the DRAM bandwidth overhead caused by prefetching.

Impact on Prefetch Volume

We classify all prefetch misses into the following four categories:

- `Bypassed`: Prefetch requests which are dropped by the filter. All other prefetch misses will be filled in the cache.
- `Demanded`: Requests which are filled in the cache and demanded at least once before eviction.
- `Premature`: Prefetches which are filled and get evicted without seeing a demand reuse, but demanded while staying in the small victim cache.
- `Useless`: Prefetches which aren't demanded in both primary and victim caches.

Table 3.1: Prefetch misses categorization for single-core benchmarks

	% prefetch misses that are			
	bypassed	demanded	premature	useless
astar	43.34	45.99	5.94	4.73
gcc.166	32.26	61.28	3.61	2.86
gcc.c-typeck	60.12	31.75	5.79	2.34
gcc.cp-decl	67.99	23.22	4.44	4.35
gcc.expr2	64.98	29.58	4.01	1.43
gcc.s04	76.77	21.06	1.70	0.47
mcf	59.73	16.58	17.30	6.40
omnetpp	46.06	45.51	5.72	2.71
soplex	39.99	49.38	4.26	6.36
xalancbmk	81.89	12.52	1.90	3.69
AVG	57.31	33.69	5.47	3.53

The oracle filter should minimize the `Useless` prefetches and categorize them as `Bypassed`. Tables 3.1 and 3.2 show this categorization for single-core and multi-core benchmarks respectively. As we can see, `Useless` prefetches are less than 3% on average in both cases. Moreover, the oracle filter is capable of reducing the prefetch volume by 55% and 30% on average for single-core and multi-core respectively due to bypassing.

Table 3.2: Prefetch misses categorization for multi-core workloads

	% prefetch misses that are			
	bypassed	demanded	premature	useless
H1	26.33	67.11	4.65	1.90
H2	34.16	59.80	3.07	2.98
H3	34.37	57.58	5.59	2.46
H4	32.01	56.32	6.93	4.74
H5	18.16	79.80	1.19	0.85
L1	15.42	80.17	2.79	1.62
L2	28.75	70.00	0.83	0.42
L3	58.46	39.58	1.19	0.77
L4	33.72	62.73	2.51	1.03
L5	22.65	76.33	0.63	0.39
M1	20.03	77.70	1.56	0.71
M2	24.05	64.47	6.57	4.91
M3	53.21	43.33	2.25	1.21
M4	55.47	35.89	5.37	3.28
M5	27.20	70.27	1.52	1.01
AVG	32.26	62.74	3.11	1.89

To get the equivalent IPC improvement for bandwidth reduction, we do a simple study. In online simulation, we drop prefetch requests that miss in the LLC with a sampling rate equal to the oracle bypassing ratio of the corresponding benchmark. As the oracle filter cannot be implemented online, this experiment will roughly reduce the DRAM controller pressure by the oracle filter's bypass rate. This gives us up to 3.34% of IPC improvement for multi-core benchmarks over the baseline SRRIP running with dual-channel DDR4 DRAM controller.

Summary

Prefetching improves system performance significantly by hiding memory latency. But it can interfere with the intelligent LLC management policies and reduce their benefits. We show that a large gap exists between optimal and any other policy even with the prefetcher turned on. To mitigate the prefetcher-caused interference, we introduce an oracle filer which is capable of bypassing unused prefetches using future look-ahead, and model its performance benefit on demand misses and prefetch volume reduction. In the next chapter, we propose an online adaptive filter.

Chapter 4

Adaptive Prefetch Filter

The spatial memory streaming prefetcher stores the learnt prefetch patterns in a set-associative table called the pattern history table (PHT), which is indexed by a function of the PC and the region offset of the triggering access. Our prefetch filter tries to evaluate the usefulness of each learnt pattern stored in the PHT of each core over a phase of benchmark execution and selectively drops all prefetch requests generated by the patterns which have lower use ratio. The key insight is if a pattern has historically prefetched unused blocks to the cache, all prefetches generated by it are likely to be unused. But as benchmarks change phases frequently, accuracy measurement should adopt to new behaviour, without completely depending on historical knowledge.

Prefetch Accuracy Measurement

Prefetch accuracy can be measured easily by augmenting a *prefetch bit* (P) with each cache block in the LLC. This bit is set during a prefetch fill and reset on demand reuse. On eviction, the P bit will act as a confidence towards prefetch usefulness. However, this has some serious drawbacks:

- The Baseline LLC policy may have replaced a prefetched block prematurely which could have enjoyed demand reuses otherwise. Hence, accuracy measurement should be independent of the underlying cache management policy.

- In this work, the prefetcher fills prefetch accesses both in the LLC and the L2 cache. There might be some prefetched blocks which get demand reuse in L2 cache but not in the LLC. Hence, we need to forward this reuse information from the L2 cache to LLC during write-back, and this demands more bandwidth between L2 and LLC.

- Blocked prefetches arising from a low-confidence pattern won't be filled in the cache, and as a result, it won't get a chance to be re-evaluated. Thus, a pattern which gets blocked once in a phase stays blocked until the phase gets over.

To address these issues, we use the working set sampling (WSS) cache, introduced in [14], as a separate structure to measure prefetcher accuracy independently from the baseline policy. The WSS cache is a traditional set-associative cache which tracks every n^{th} block of a few sampled pages. For each block, we are maintain one bit which is set during the LLC fill of a prefetch request to that block and is reset if a demand access is generated for that block. Each WSS cache entry also stores the index of the PHT pattern which started prefetching on the page tracked by the entry. The structure of a WSS cache entry is shown in Figure 4.1. On the other hand, each PHT entry of the SMS prefetcher is augmented with two 8-bit counters namely *total count*(TC) and *use count*(UC) and one 3 bit counter named *confidence*(C). WSS cache insertion and updation policies are as follows.

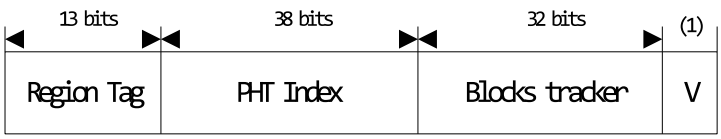


Figure 4.1: Structure of a WSS cache entry

Insertion and Replacement Policy

The WSS cache tracks all prefetch requests that go or would have gone off-chip. The insertion happens in two cases. First, when a prefetch request misses in the LLC.

Secondly, when a prefetcher's decision gets overridden by the filter's decision. In the latter case, the targeted prefetch addresses are first probed in the LLC. If it is not present there, the WSS cache starts tracking the request. It is essential to track the usefulness of any PHT pattern, which is not allowed to prefetch due to lower use ratio, but still gets a chance to be re-evaluated and unblock itself if proven useful in future.

During insertion, first the WSS cache is searched for the page to be prefetched. If the page is found, it looks for the corresponding PHT pattern index stored. If the current prefetch is also generated by the same pattern index, it sets the tracking bit corresponding to the page offset. However if the indices mismatch, it invalidates the previous tracking pattern by resetting all offset bits, stores the current pattern index, and sets the tracking bit of the current offset.

If the page is not found at all, it looks for an invalid entry in the WSS cache. If none is found, this prefetch won't be tracked in the current phase. Thus, the WSS cache doesn't have any victim selection policy. It gets filled up at the beginning of an epoch and collects demand reuses during the whole interval. At the end of an epoch, all entries are invalidated and the WSS cache starts afresh from the next epoch. The TC and the UC counter values of all PHT entries gets halved. This gives us a chance to capture the demand reuses which couldn't have been captured in the LLC due to premature eviction by the baseline policy. If the insertion is successful, TC of the generating pattern's entry in the PHT is incremented by one.

Updation Policy

Whenever a demand memory request is generated, it is looked into the WSS cache in parallel with the cache hierarchy. If the page is found and the corresponding bit of the block is set, it gets reset. Then the PHT is searched with the pattern index stored in the WSS cache entry and its UC is incremented, if found.

Decision Policy

From the insertion and the updation policy, it is clear that the TC of a PHT entry is tracking the total number of prefetches generated from the corresponding pattern that went or would have gone off-chip and the UC measures how many of them would get a demand reuse. During the prefetch request generation, UC/TC ratio is calculated in parallel. If it is less than or equal to a *cutoff threshold*, the confidence counter C of the corresponding PHT entry is checked. If C has reached its maximum value, the prefetch request gets probed in the LLC and is tracked as per the insertion policy, and is not injected in the memory system. If C has not reached the maximum, it is incremented by one, and the prefetch is allowed to proceed to DRAM. If the ratio is greater than the *cutoff threshold*, C gets reset. Basically, the confidence counter buys more time to saturate the use ratio in an interval and gains good confidence to block prefetches.

Summary

In this chapter we looked into the structure of the working set sampling cache and used it to measure prefetcher accuracy in a way that is independent of underlying baseline LLC policy. We described the management policies and the decision making logic for selectively blocking prefetch requests. In the next chapter, we will discuss our experimental setup and benchmarks selection criteria.

Chapter 5

Experimental Methodology

Simulation Environment

We use an in-house version of the Multi2sim simulator [19] to model the CPU cores. Each dynamically scheduled, out-of-order issue core is clocked at 4GHz. The three level cache hierarchy is modelled roughly after the Intel Skylake Core i7 systems [1]. The last-level cache maintains strict inclusion with the core-private caches. Each L2 cache as well as each LLC bank has 16 MSHRs. We model per-core SMS prefetcher which learns from the committed access stream and triggers prefetching decision in every L2 cache demand miss and fills prefetches in the L2 cache and LLC. The DRAM system and bus timings are modelled by attaching the DRAMSim2 [15] simulator. Table 5.1 summarizes all baseline architectural parameters.

Table 5.1: Core simulation parameters

CPU	4 cores, 4GHz, 224 entry ROB, 128 entry Iqueue and Lsqueue
L1 I and D cache	Private, 32KB, 8ways, 64B block, 2 cycles, LRU
L2 Cache	Private, 256KB, 8 ways, 64B block, 5 cycles, LRU
LLC	Shared, 2MB per core, 16 ways, 8 cycles, SRRIP, Inclusive
Others	16 MSHR per LLC bank, 16 entry DRAM transaction queue, 2x2 quad core network mesh with 1 cycle fixed link latency
DRAM	Dual-channel DDR4, 1200MHz, x8, BL=8, tRAS=39, tRCD=15, tRP=15
Prefetcher	4KB region, 16 entries Accumulation table, 32 entries Filter table. 16K entry 8 way PHT, concatenation of PC and page offset for indexing PHT

We set the WSS cache organization to 1K entries, 8 way associative. Each entry tracks every even block of a page. Epoch length is set to 512K L2 cache demand misses, and the prefetch use ratio *cutoff threshold* is set to 0.1.

Workload Construction

Single-core Workloads

We identify a single one-billion instruction trace from each 54 benchmark-input combination of the SPEC CPU2006 suite [18] using SimPoint [12]. We run offline simulations extracting the memory traces with traditional a 2MB cache and a big 1GB cache, both having the SRRIP policy. Table 5.2 shows only those benchmark-input combinations which have IPC improvement more than 10% in the presence of near-infinite 1GB cache. Out of these benchmarks only those are selected finally which have less than 70% of prefetch accuracy for a 2MB optimal cache.

The idea is to select only those memory-intensive benchmarks, which can improve by caching, but at the same time, are suffering from prefetcher pollution. Note that `403.gcc` with most of the supplied inputs shows poor prefetching and hence, gets selected for five different inputs.

Multi-core Workloads

There are two types of multicore mixes: homogeneous and heterogeneous. Homogeneous mixes are created by running four copies of each of the selected benchmarks. For the heterogeneous mixes, we divide all application-input combinations into two categories. Those with more than 2 MPKI of demand misses with prefetching are classified as HIGH and the rest are LOW. Mixes H1-H5 are created by mixing applications from the HIGH category, L1-L5 are mixes from the LOW, and M1-M5 are hybrid mixes from both categories. Each benchmark runs simultaneously and restarts after one billion instructions until all of them have completed one billion instructions. Table 5.3 shows the multi-core mixes.

Table 5.2: The 17 benchmarks with LLC demand misses per kilo instruction(MPKI) count for the baseline SRRIP policy with and without prefetcher and OPT cache with prefetcher. OPT prefetch accuracy is calculated as total used prefetches over total prefetch fills. The ten selected benchmarks for this study are marked in **bold**.

Name	**MPKI (no pref)**	**MPKI (w/ pref)**	**MPKI (OPT)**	**OPT pref accuracy**
mcf	24.55	24.75	14.94	0.218
soplex.pds-50	17.54	8.48	5.30	0.536
omnetpp	9.67	6.98	4.39	0.436
gcc.expr2	4.43	3.02	2.54	0.301
sphinx3	12.89	2.83	1.52	0.779
gcc.166	8.01	3.08	2.25	0.639
lbm	24.68	5.23	3.30	0.986
gcc.c-typeck	1.17	1.03	0.54	0.293
cactusADM	4.47	4.27	2.74	0.999
xalancbmk	1.43	1.51	0.61	0.129
gcc.s04	2.12	2	0.99	0.177
bzip2.source	2.96	1.33	0.60	0.719
leslie3d	11.51	2.42	1.77	0.983
astar.BigLakes2048	1.81	0.88	0.51	0.470
soplex.ref	17.75	2.83	2.01	0.845
gcc.cp-decl	2.62	2.17	1.26	0.183
bzip2.text	5.63	0.72	0.27	0.956

Table 5.3: Heterogeneous multicore mixes

HIGH	mcf, soplex.pds-50, lbm, omnetpp, milc, gcc.166, sphinx3, gcc.cp-decl
LOW	bwaves, zeusmp, gcc.s04, xalancbmk, bzip2.source, gcc.c-typeck, astar.BigLakes2048
H1	mcf, lbm, gcc.166, sphinx3
H2	soplex.pds-50, omnetpp, milc, gcc.cp-decl
H3	gcc.166, gcc.cp-decl, mcf, lbm
H4	lbm, omnetpp, mcf, soplex.pds-50
H5	milc, sphinx3, lbm, gcc.cp-decl
L1	astar.BigLakes2048, bwaves, bzip2.source, gcc.c-typeck
L2	gcc.c-typeck, gcc.s04, xalancbmk, bwaves
L3	astar.BigLakes2048, xalancbmk, zeusmp, gcc.s04
L4	zeusmp, bzip2.source, gcc.c-typeck, astar.BigLakes2048
L5	xalancbmk, gcc.s04, zeusmp, bwaves
M1	bwaves, gcc.s04, omnetpp, sphinx3
M2	mcf, lbm, bwaves, soplex.pds-50
M3	gcc.cp-decl, gcc.166, gcc.s04, gcc.c-typeck
M4	xalancbmk, mcf, soplex.pds-50, astar.BigLakes2048
M5	gcc.166, sphinx3, milc, gcc.c-typeck

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 6

Performance Evaluation

In this chapter, we compare our Adaptive Prefetch Filter (APF) with the state-of-the-art PACMan-DYN [21], the informed caching policy for prefetches (ICP) [16] and the weighted-mean filter [11] in terms of prefetch volume reduction, total off-chip bandwidth reduction, and overall IPC improvement. We also simulate the offline optimal policy to compare these four in terms of LLC miss counts.

Overview of Results

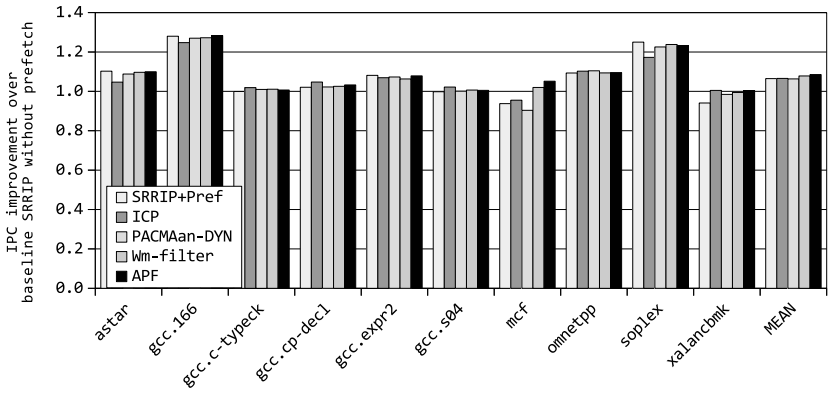


Figure 6.1: Performance comparison for single core benchmarks

Figure 6.1 shows performance comparison of APF with respect to PACMan-DYN, ICP, and WM-filter for the single-core benchmarks. Baseline SRRIP with prefetching and PACMan-DYN improves IPC by 6.5% and 6.3% on average with

respect to no prefetching, whereas APF improves IPC by 8.5%. Note that benchmarks like `mcf` and `xalancbmk`, which suffer from prefetching the most, enjoy the biggest benefit. IPC of `mcf` drops by 10% in the presence of prefetcher in the baseline SRRIP policy, whereas it increases by 5% with APF.

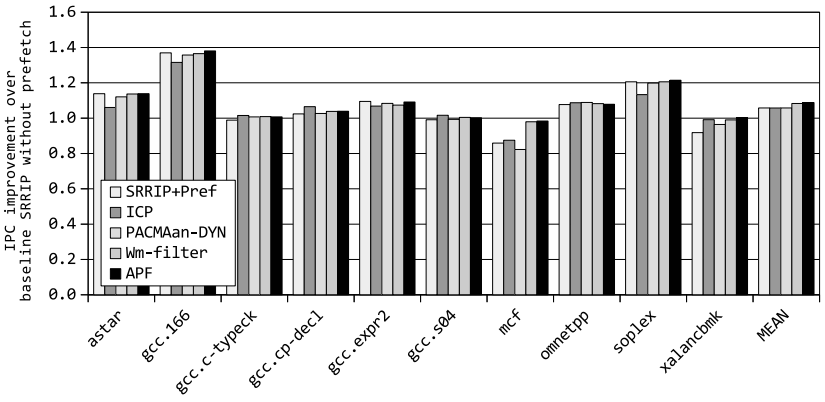


Figure 6.2: Performance comparison for multi core homogeneous mixes

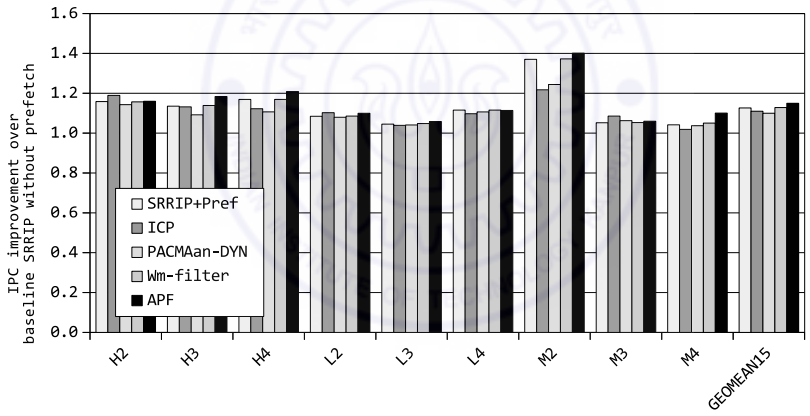


Figure 6.3: Performance comparison for multi core heterogeneous mixes

Figure 6.2 and 6.3 show the results for the multi-core homogeneous and heterogeneous mixes. We show only 9 out of the 15 heterogeneous mixes in Figure 6.3, but the column GEOMEAN15 is calculated by taking the mean of all mixes. For homogeneous mixes, PACMan-DYN and APF improves performance by 5.8% and 8.8%, respectively relative to the baseline SRRIP policy with no prefetch, whereas it increases by 10% and 15% respectively for heterogeneous mixes. In mix like M2, where all the constituents `mcf`, `lbm`, `bwaves`, and `soplex.pds-50` prefetch heavily but only some of them (`lbm` and `bwaves`) are accurate, get affected most by APF.

PACMan-DYN increases performance by 24.4% in M2 whereas APF improves it by 40%.

Overall, APF consistently improves or maintains performance benefit caused by the prefetcher across a wide range of workloads. Prefetcher-friendly benchmarks like `lbm` or `libquantum` maintain its benefit whereas benchmarks like `mcf` and `xalancbmk` which suffer from prefetching get a boost in performance by elimination of useless prefetches.

Off-chip Miss Reductions

Figure 6.4 compares the amount of prefetch requests that miss in the LLC and go off-chip normalized to baseline SRRIP with prefetch. As we can see, ICP and PACMan-DYN increase the prefetch volume by 18% and 6.8%, respectively. This is due to active de-prioritization of prefetch filled blocks, which get evicted without getting reused and get prefetched again. APF, on the other hand, reduces off-chip prefetch traffic by 37.7% on average with respect to baseline prefetching. In case of `mcf` and `xalancbmk`, where most of the prefetches are inaccurate, it reduces prefetch volume by 87.1% and 67.9%, respectively.

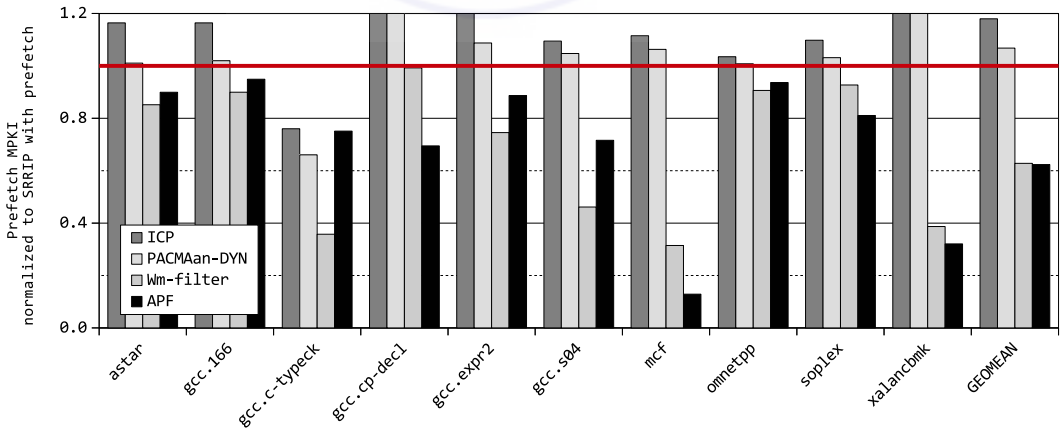


Figure 6.4: Prefetch volume reduction compared to baseline SRRIP with prefetch

APF also manages to reduce the total LLC misses by 18.8% on average for single core benchmarks, whereas PACMan-DYN and ICP *increase* it by 2.9% and 8.8%, respectively, as summarized in Figure 6.5.

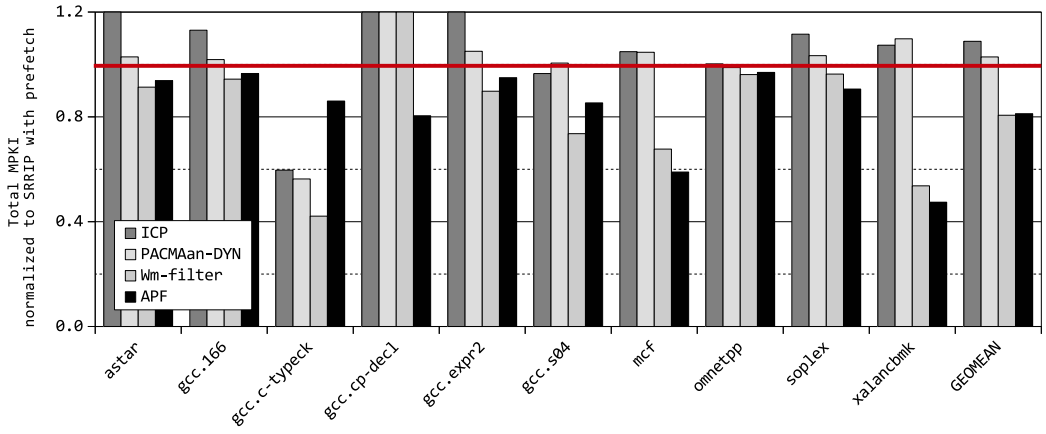


Figure 6.5: Total LLC MPKI reduction compared to baseline SRRIP with prefetch

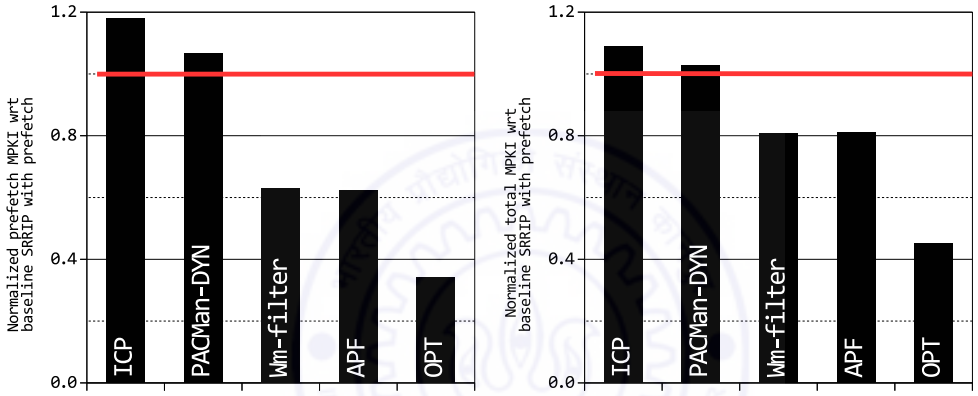


Figure 6.6: Average prefetch and total LLC miss reductions for single core benchmarks

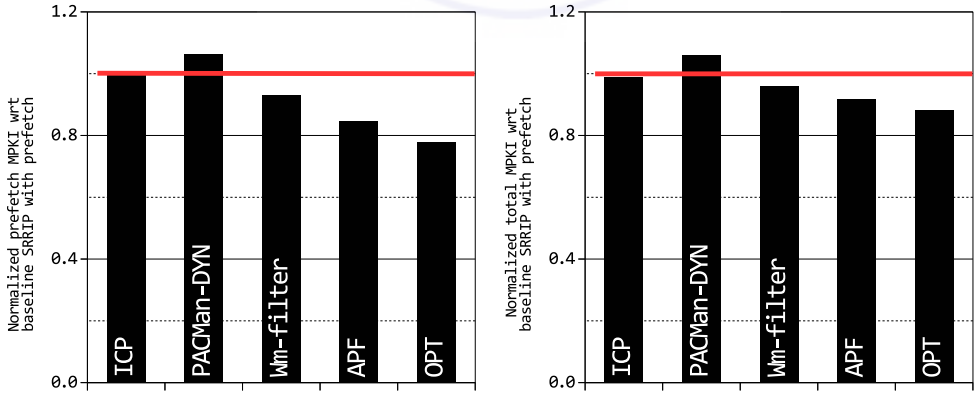


Figure 6.7: Average prefetch and total LLC miss reductions for multicore heterogeneous mixes

Figure 6.6 and 6.7 show the average reduction of prefetch and total misses of single-core and multi-core heterogeneous mixes with the optimal replacement policy. For single-core benchmarks, OPT reduces prefetch and total miss volume by 66%

and 55%, respectively with respect to baseline SRRIP with prefetch, whereas APF reduces it by 37.7% and 18.8%, respectively, thus bridging almost one-third gap between baseline and OPT policy. For multicore heterogeneous mixes, OPT reduces prefetch and total misses by 22.1 % and 11.8%, respectively, whereas APF reduces it by 15.4% and 8.3% on average.

Impact on DRAM Latency

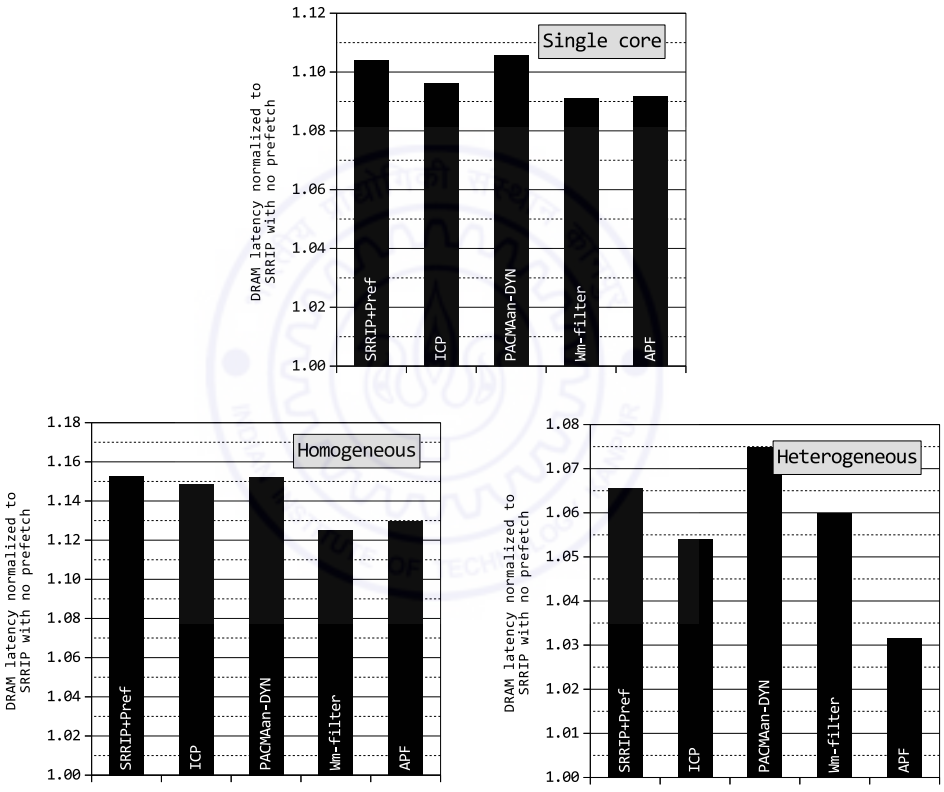


Figure 6.8: Average DRAM latency normalized to baseline SRRIP without prefetch

To understand the impact of prefetch volume reduction on DRAM subsystem, we compare the average DRAM latency of different benchmarks. Average DRAM latency for a workload is calculated as follows

$$\text{Average Latency} = \frac{\text{Total CPU cycles spent in LLC MSHR for DRAM load reply}}{\text{Total number of LLC misses}}$$

Figure 6.8 compares latencies with respect to baseline SRRIP without prefetch-

ing. Note that, dual channel DDR4 system is used in this work. For multicore homogeneous and heterogeneous mixes, prefetching adds 15.3% and 6.6% of average latency respectively. Heterogeneous mixes put less pressure on the DRAM channels as constituent benchmarks of different mixes have different prefetchability. APF on the other hand increases average latencies by 12.6% and 3.1% only. The impact on latency for single-core benchmark is almost negligible due to the large available bandwidth in the dual-channel DDR4 system. APF reduces latency only by 1.2% over SRRIP with prefetching in this case.

Results Summary

APF consistently maintains or improves performance benefit due to the prefetcher over a wide range of benchmarks, eliminating most of the useless prefetches. For single core workloads, it increases performance by up to 12.15% (2.1% on average) with respect to baseline SRRIP policy with prefetching. For multicore homogeneous and heterogeneous mixes, it improves performance by up to 14.55% (2.98% on average) and 5.7% (3.2% on average), respectively. It also reduces off-chip prefetch volume by up to 87.1% (29.1% on average) and 46.2% (14.58% on average), respectively for single-core and multi-core heterogeneous mixes, as compared to the baseline policy with prefetch. The reduction in LLC miss volume results in a 2.1% and 3.6% reduction in average DRAM latency with respect to baseline prefetching for multicore homogeneous and heterogeneous mixes, respectively.

Chapter 7

Conclusion

In this work, we showed prefetching and intelligent cache management policy, two orthogonal approaches to defend the memory wall problem, might interfere destructively, reducing the overall performance benefit. This is largely due to unnecessary prefetches that aren't going to be demanded in near-immediate future. We first modelled an oracle prefetch filter to address this problem and established that intelligent caching schemes already show some intrinsic property of selective filtering. We proposed adaptive prefetch filter (APF), which has a small working set sampling (WSS)cache in its heart that is capable of predicting useless prefetches on the fly. APF improves performance of single-core and multicore heterogeneous workloads by 8.5% and 15% with respect to baseline without prefetching, whereas prefetching only improves the same by 6.5% and 12.1%, respectively.

Future Work

Another approach to mitigate prefetcher induced pollution might be partitioning cache into *easily prefetchable* and *non-prefetchable* lines. As *easily prefetchable* lines have higher chances to be prefetched again before getting demanded, these blocks will serve as better victim candidates. This must be accompanied by accurate dead block prediction for *prefetchable* blocks, otherwise it might increase the overall miss volume due to back-invalidation by LLC to maintain inclusion.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Bibliography

[1] *6th Generation Intel® Core™ i7 Processors*. URL: http://ark.intel.com/products/88196/Intel-Core-i7-6700-Processor-8M-Cache-up-to-4_00-GHz.

[2] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[3] Mainak Chaudhuri. "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 401–412.

[4] Mainak Chaudhuri et al. "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches". In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM. 2012, pp. 293–304.

[5] Tien-Fu Chen and Jean-Loup Baer. "Effective hardware-based data prefetching for high-performance processors". In: *IEEE transactions on computers* 44.5 (1995), pp. 609–623.

[6] Aamer Jaleel et al. "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)". In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 60–71. ISSN: 0163-5964. DOI: 10.1145/1816038.1815971. URL: http://doi.acm.org/10.1145/1816038.1815971.

[7] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. "Sampling dead block prediction for last-level caches". In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2010, pp. 175–186.

[8] Mazen Kharbutli and Yan Solihin. "Counter-based cache replacement and bypassing algorithms". In: *IEEE Transactions on Computers* 57.4 (2008), pp. 433–447.

[9] Chang Joo Lee et al. "Prefetch-aware DRAM controllers". In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2008, pp. 200–209.

[10] Kyle J Nesbit and James E Smith. "Data cache prefetching using a global history buffer". In: *Software, IEE Proceedings-*. IEEE. 2004, pp. 96–96.

[11] Biswabandan Panda and Shankar Balachandran. "Expert Prefetch Prediction: An Expert Predicting the Usefulness of Hardware Prefetchers". In: (2015).

[12] Erez Perelman et al. "Using SimPoint for accurate and efficient simulation". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 31. 1. ACM. 2003, pp. 318–319.

[13] Moinuddin K Qureshi et al. "Adaptive insertion policies for high performance caching". In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 381–391.

[14] Siddharth Rai and Mainak Chaudhuri. "Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: ACM, 2016, 3:1–3:14. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926266. URL: http://doi.acm.org/10.1145/2925426.2926266.

[15] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A cycle accurate memory system simulator". In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[16] Vivek Seshadri et al. “Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015), p. 51.

[17] Stephen Somogyi et al. “Spatial memory streaming”. In: *ACM SIGARCH Computer Architecture News* 34.2 (2006), pp. 252–263.

[18] *SPEC CPU® 2006*. URL: `https://www.spec.org/cpu2006/`.

[19] Rafael Ubal et al. “ Multi2Sim: A Simulation Framework for CPU-GPU Computing ”. In: *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 2012.

[20] Thomas F Wenisch et al. “Practical off-chip meta-data for temporal memory streaming”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE. 2009, pp. 79–90.

[21] Carole-Jean Wu et al. “PACMan: prefetch-aware cache management for high performance caching”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 442–453.

[22] Carole-Jean Wu et al. “SHiP: Signature-based hit predictor for high performance caching”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 430–441.

[23] Xiaotong Zhuang and H-HS Lee. “A hardware-based cache pollution filtering mechanism for aggressive prefetches”. In: *Parallel Processing, 2003. Proceedings. 2003 International Conference on*. IEEE. 2003, pp. 286–293.