2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

To Cross, or Not to Cross Pages for Prefetching?

Georgios Vavouliotis[†] Marti Torrents[*] Boris Grot[†§] Kleovoulos Kalaitzidis[†] Leeor Peled[**] Marc Casas[*‡]

Computing Systems Lab, Huawei Zurich Research Center[†]
Boole Lab, Huawei Tel-Aviv Research Center[**]
Barcelona Supercomputing Center[*] Universitat Politecnica de Catalunya[‡] University of Edinburgh[§]
{georgios.vavouliotis, boris.grot, kleovoulos.kalaitzidis, leeor.peled}@huawei.com, {marti.torrents, marc.casas}@bsc.es

***Abstract*—Despite processor vendors reporting that cache prefetchers operating with virtual addresses are permitted to cross page boundaries, academia is focused on optimizing cache prefetching for patterns within page boundaries. This work reveals that page-cross prefetching at the first-level data cache (L1D) is seldom beneficial across different execution phases and workloads while showing that state-of-the-art L1D prefetchers are not very accurate at prefetching across page boundaries.**

In response, we propose *MOKA*, a holistic framework for designing *Page-Cross Filters*, *i.e.*, microarchitectural schemes that ensure effective and accurate prefetching across page boundaries. MOKA combines (i) hashed perceptron predictors that use prefetcher-independent program features, (ii) predictors that adapt decisions based on the system state (*e.g.*, TLB pressure), and (iii) a scheme to dynamically optimize predictions across different execution phases and workload types. We use the MOKA framework to prototype a Page-Cross Filter, named *DRIPPER*, for three relevant L1D prefetchers (Berti [60], IPCP [61], BOP [57]). We show that *DRIPPER* accurately enables page-cross prefetching only when it is beneficial for performance. For instance, Berti [60] (state-of-the-art prefetcher) combined with *DRIPPER* improves single-core geomean performance over Berti that always permits page-cross prefetches and Berti that always discards page-cross prefetches by 1.7%(1.2%) and 2.5%(2.1%) across 218 seen (178 unseen) workloads, respectively. Across 300 8-core mixes, the corresponding geomean speedups are 2.0% and 3.3%. Finally, we show that *DRIPPER* provides consistent benefits when both 4KB pages and 2MB large pages are used.

I. Introduction

Modern microarchitectural designs [1], [3], [6], [8], [29], [35], [37], [51], [55], [79], [80], [84] implement multiple hardware prefetchers to attenuate the Memory Wall bottleneck [56], [94]. Recent industrial reports [3], [8] highlight that advances in hardware prefetching are responsible for a large portion of the IPC uplifts between processor generations.

Cache prefetchers proposed by the academic community [13], [19], [22], [38], [48], [57], [60], [61], [76], [81] are typically restricted to identify and prefetch for memory access patterns within page boundaries and discard prefetch requests that cross pages. In contrast, CPU vendors permit prefetchers placed alongside *Virtually Indexed Physically Tagged (VIPT)* caches [15], *i.e.*, prefetchers operating with virtual addresses, to cross page boundaries [3], [8], [29], [35]. However, they do not provide additional information about their page-cross prefetching strategy. They do not disclose whether they constantly permit page-cross prefetching or use dedicated runtime techniques to optimize it.

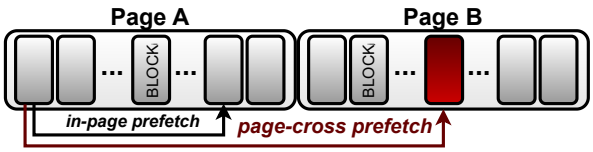


Fig. 1: In-page and page-cross prefetch requests.

Crossing pages for prefetching, depicted in Figure 1, is a high-risk, high-gain technique. *Accurate page-cross prefetching* can provide great performance gains since it increases Translation Lookaside Buffer (TLB) and cache hits, reduces the number of demand page walks, and potentially improves the timeliness of prefetching. *Inaccurate page-cross prefetching* significantly harms performance since it pollutes both the cache and the TLB, triggers useless memory accesses (up to 4 memory accesses for the speculative page walk and 1 memory access to serve the cache prefetch request), increases the dynamic energy, and harms prefetching timeliness.

This paper argues for more attention to page-cross prefetching and presents the first study on characterizing the properties of page-cross prefetching for VIPT caches using three state-of-the-art L1D prefetchers (Berti [60], IPCP [61], BOP [57]) and a diverse set of 396 workloads. We show that (i) static policies for page-cross prefetching, *i.e.*, policies that always permit or always discard page-cross prefetches, are seldom beneficial for performance, (ii) constantly permitting page-cross prefetching provides performance gains over the policy that always discards page-cross prefetches only for a subset of the considered workloads; the opposite behavior is observed for the rest of the workloads, and (iii) state-of-the-art L1D prefetchers have low accuracy with respect to page-cross prefetching, highlighting that a scheme able to ensure accurate and effective page-cross prefetching has the potential to provide significant benefits.

Based on our findings, we propose MOKA, a framework for designing effective and fully legacy preserving *Page-Cross Filters* for prefetchers operating with virtual addresses, *i.e.*, uarch filters that predict the usefulness of page-cross prefetch requests ensuring that only useful page-cross prefetches will be issued. The MOKA framework provides (1) a set of hashed perceptron predictors [17], [85] realized with prediction tables indexed with prefetcher-independent program features (*e.g.*, PC), (2) a set of simple predictors that take into account the system state (*e.g.*, TLB pressure) when deciding whether to permit or not a page-cross prefetch, and (3) an adaptive

scheme that optimizes predictions across execution phases and workload types by tuning the decision thresholds at runtime.

We prove the versatility of the MOKA framework by using it to prototype a Page-Cross Filter, named *DRIPPER*, for three state-of-the-art L1D prefetchers (Berti [60], IPCP [61], BOP [57]). DRIPPER improves performance for all considered L1D prefetchers across a set of 396 workloads (218 seen, 178 unseen) since it accurately predicts the usefulness of page-cross prefetch requests and issues only the useful ones.

In summary, this paper makes the following contributions:

- We provide the first study on page-cross prefetching for VIPT caches using three L1D prefetchers (Berti [60], IPCP [61], BOP [57]) and a set of 396 workloads (218 seen, 178 unseen). This study highlights that constantly issuing page-cross prefetches is seldom beneficial across different execution phases and workload types.
- We propose MOKA, a framework for designing *Page-Cross Filters* for prefetchers placed alongside VIPT caches. MOKA comes with a variety of prefetcher-independent program features, system features that take into account the system state in the decision making, and a scheme that tunes the activation threshold at runtime.
- We use the MOKA framework to prototype a Page-Cross Filter, named *DRIPPER*, for all considered prefetchers (Berti, IPCP, BOP). For example, Berti combined with *DRIPPER* improves single-core geomean performance over Berti that always permits page-cross prefetches and Berti that always discards page-cross prefetches by 1.7%(1.2%) and 2.5%(2.1%) across 218 seen (178 unseen) workloads, respectively. Across 300 random 8-core mixes, the corresponding geomean speedups are 2.0% and 3.3%. We show that *DRIPPER* provides consistent benefits when both 4KB and 2MB pages are used.

II. Motivation

This section shows that whether page-cross prefetching for first-level caches is beneficial for performance varies on a workload by workload basis. We also provide evidence that a scheme able to dynamically decide when to permit or discard page-cross prefetches has the potential to provide great gains. This study targets L1D prefetchers and not lower-level cache prefetchers because first-level caches have direct access to the virtual memory subsystem as opposed to lower-level caches.

A. Cache Prefetching and Page Boundaries

1) First-Level Caches and Page-Cross Prefetching: L1D prefetchers [14], [19], [57], [60], [61], [76], [81] drive prefetching using virtual addresses because first-level caches are typically implemented as VIPT structures. Patterns that are easy to detect in the virtual address space might be hard to detect in the physical address space, since two addresses which are contiguous in the virtual address space might be separated by large distances in the physical address space [48]. Conceptually, L1D prefetchers can cross page boundaries since they have direct access to the TLB [2], [3], [8]. However, there is a critical question to answer: *What should L1D prefetchers do when the translation of the page where the prefetched block resides is not present in the TLB?* Should they discard the prefetch or initiate a page table walk to fetch the corresponding translation from the page table? Triggering page walks for page-cross prefetch requests is a high-risk technique since it may incur up to 5 useless memory accesses (up to 4 memory accesses for the speculative page walk [12] and 1 memory access to serve the cache prefetch request) if the page-cross prefetch request is inaccurate. Therefore, inaccurate page-cross prefetching greatly harms performance due to the additional memory accesses introduced and the pollution caused to the TLB and cache hierarchy. On the other hand, accurate page-cross prefetching can provide significant benefits since it improves the efficacy of cache prefetching and reduces the number of TLB misses by prefetching address translations in the TLB ahead of demand memory accesses.

2) Lower-Level Caches and Page-Cross Prefetching: Prefetchers placed alongside lower-level caches (L2C, LLC) [18], [20], [42], [48], [49], [71], [77], [92], [93] drive prefetching decisions using physical addresses since these caches are implemented as *Physically Indexed Physically Tagged (PIPT)* structures [15]. These prefetchers typically permit prefetching only within physical page boundaries for security reasons since physical address contiguity is not guaranteed [89]. Therefore, allowing page-cross prefetching in the physical address space could introduce new side channels that an adversary could exploit to attack the system [23], [36]; a recent study [90] reveals how to exploit page-cross prefetching at lower-level caches to attack recent Apple processors.

B. Trends on Page-Cross Prefetching

1) Academic Viewpoint: Recent literature has proposed numerous L1D prefetchers [14], [19], [57], [60], [61], [76], [81]. These works typically design new prefetchers that correlate cache accesses with more features than prior designs to capture more distinct patterns. Academic L1D prefetchers are mainly restricted to prefetch within page boundaries and they are not optimized for crossing page boundaries, although they have direct access to the virtual memory subsystem (Section II-A).

2) Industrial Viewpoint: Limiting L1D prefetchers to prefetch within page boundaries provides suboptimal gains when the prefetchers are able to accurately perform page-cross prefetching. While academic works still restrict L1D prefetchers to prefetch within page boundaries, vendors permit L1D prefetchers to prefetch across page boundaries [3], [8], [29], [35]. Page-cross prefetches need to go through the TLB hierarchy and potentially initiate page walks that fetch in the TLB the translations of the pages where the prefetched blocks reside; Section II-A presents the pros and cons of page-cross prefetching. However, vendors do not provide additional information. It is unclear whether vendors follow a static policy that always permits page-cross prefetching or use techniques to coordinate prefetching across page boundaries.

There is no prior work on analyzing and characterizing page-cross prefetching for first-level caches.

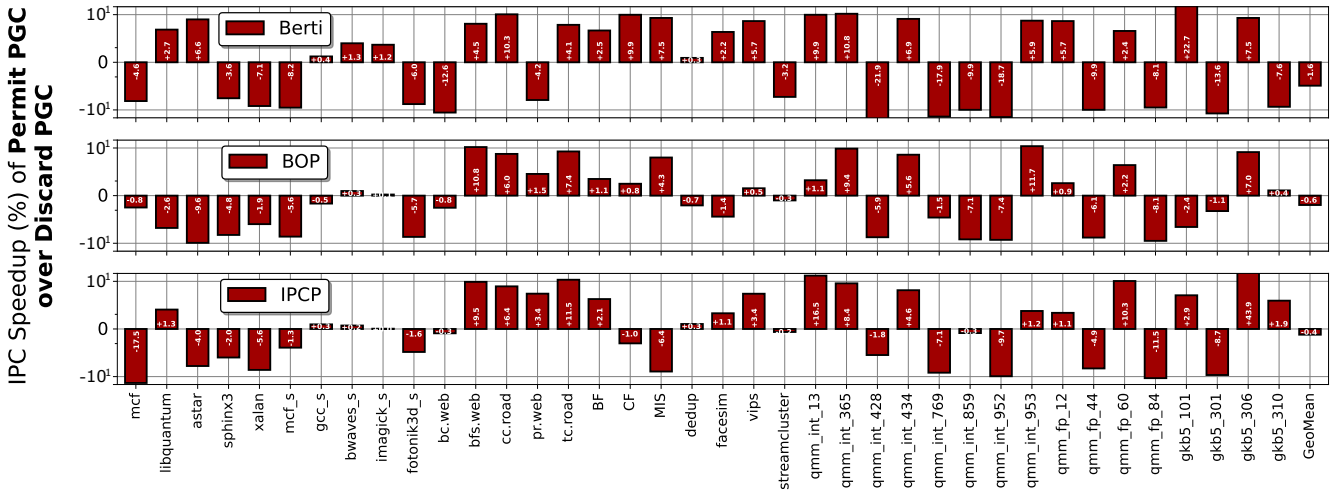


Fig. 2: IPC gains of Berti, BOP, and IPCP when they always permit page-cross prefetching (Permit PGC) over a baseline that does not permit the respective prefetcher to cross page boundaries (Discard PGC) across a set of memory-intensive workloads.

C. Analyzing Page-Cross Prefetching

This section quantifies the performance impact of page-cross prefetching using the ChampSim simulator [5], [34]. Section IV describes in detail the simulation infrastructure. To provide broad conclusions, we consider three well-established L1D prefetchers: Berti [60], IPCP [61], and BOP [57]. Regarding the workloads, we use memory-intensive benchmarks spanning various benchmark suites: SPEC 2006 [9], SPEC 2017 [10], GAP [16], LIGRA [78], PARSEC [11], and GeekBench [7]. We also evaluate industrial integer and floating point workloads provided by Qualcomm for CVP-1 [4], [30]. Our main evaluation campaign (Section V) considers a larger workload set (396 workloads in Section IV-A).

Qualitatively, page-cross prefetching has the potential to provide significant performance gains when it is performed accurately since it can (i) improve the efficacy and timeliness of prefetching and (ii) reduce the number of TLB misses since it prefetches address translations in the TLB. To answer whether L1D prefetchers are effective at prefetching across page boundaries, we evaluate two different versions of all considered prefetchers. The first version always permits the underlying prefetcher to issue page-cross prefetches (Permit PGC) and the second version always discards prefetch requests that cross page boundaries (Discard PGC). Figure 2 presents the IPC improvements of all considered prefetchers when combined with Permit PGC over a baseline that does not permit the respective L1D prefetcher to cross page boundaries (Discard PGC) across the considered workloads.

Looking at Figure 2, we observe that always permitting page-cross prefetching (Permit PGC) behaves differently across different workloads; this holds for all Berti, BOP, and IPCP. Focusing on Berti, we observe that Berti with Permit PGC significantly outperforms Berti with Discard PGC for certain workloads (*e.g.*, `astar`, `cc.road`, `MIS`, `vips`, `qmm_int_365`, `gkb5_101`). This happens because Berti accurately crosses pages for these benchmarks, which improves the timeliness of prefetching and increases the TLB hit rate. The opposite behavior, *i.e.*, Berti with Discard PGC outperforms Berti with Permit PGC, is observed for benchmarks like `sphinx3`, `fotonik3d_s`, `bc.web`, `pr.web`, `qmm_int_859`, `qmm_fp_44`, and `gkb5_310`. For these workloads, allowing Berti to constantly cross pages brings performance degradation over the conservative scenario that permits prefetching only within page boundaries (Discard PGC). This happens because each useless page-cross prefetch of Berti introduces up to 5 useless memory accesses (4 for the speculative page walk and 1 for the actual prefetch request). As a result, Berti increases the pressure on the cache hierarchy that results in cache and TLB pollution. We observe similar behavior for BOP and IPCP prefetchers. The main takeaway of this study is that there is no single static strategy for page-cross prefetching (Permit PGC, Discard PGC) that performs best across all workloads.

Always permitting page-cross prefetching may increase or decrease performance over the scenario that always discards page-cross prefetches across different workloads.

Figure 3 complements the results of Figure 2 by quantifying the usefulness of page-cross prefetching in the scenario that always permits page-cross prefetching (Permit PGC) across the same workloads and prefetchers used in Figure 2. To do so, Figure 3 divides the issued page-cross prefetches into two categories: (i) useful page-cross prefetches, *i.e.*, prefetches that provided at least one hit during their lifetime in the cache and (ii) useless page-cross prefetches, *i.e.*, prefetches that did not provide any hit during their lifetime in the cache. Figure 3 (left) shows the distribution of useful and useless page-crossing prefetches across all workloads and prefetchers while Figure 3 (right) presents the average percentages of useful and useless page-crossing prefetches. Looking at the distributions, we observe all types of possible behaviors. There are workloads for which (i) the vast majority of page-

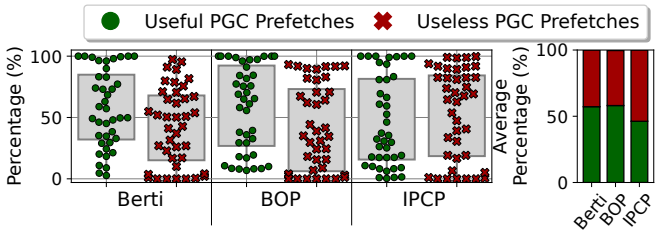


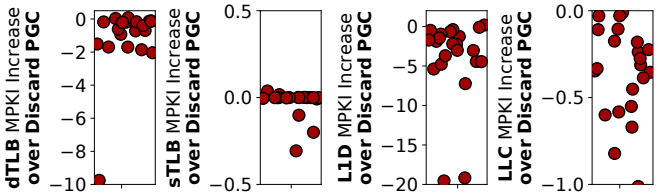
Fig. 3: Distribution (left) and average percentage (right) of useful and useless page-cross prefetches across the same workloads and prefetchers used in Figure 2. In Figure 3 (left), each workload is represented by one green and one red marker showing the percentages of useful and useless page-cross prefetches, respectively, which are summed to 100%.

cross prefetches are useful (green markers ∼100%), (ii) most page-cross prefetches are useless (red markers ∼100%), and (iii) some page-cross prefetches are useful and some useless. Focusing on the averages, we observe that ∼50% of the issued page-cross prefetches are useful (the rest are useless); this holds true across all considered prefetchers. We conclude that state-of-the-art L1D prefetchers are not very accurate at page-cross prefetching. When they do it accurately and effectively (*e.g.*, `tc.road` and `qmm_int_13` in Figure 2), they significantly improve performance over the scenario that does not permit page-cross prefetching. However, inaccurate page-cross prefetching (*e.g.*, `sphinx3` in Figure 2) significantly harms performance due to the additional page walks required and the TLB and cache pollution introduced.

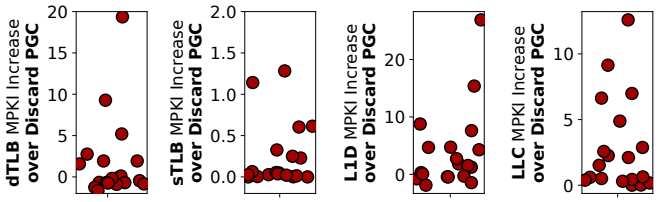
To explain why for some workloads Permit PGC outperforms Discard PGC and for others the opposite behavior is observed (Figure 2), we consider the state-of-the-art L1D prefetcher, Berti, and we split the workloads used in Figure 2 in two categories: (i) workloads for which Permit PGC outperforms Discard PGC and (ii) workloads for which Discard PGC outperforms Permit PGC. Figure 4a and 4b analyze the impact of Berti with Permit PGC on dTLB[1] MPKI, sTLB[1] MPKI, L1D MPKI, and LLC MPKI over Berti with Discard PGC for workload categories (i) and (ii), respectively.

Focusing on the workloads for which Permit PGC outperforms Discard PGC (Figure 4a), we observe a significant decrease in dTLB MPKI and a small decrease in sTLB MPKI because for this set of workloads Berti effectively crosses page boundaries, thus reducing TLB pressure. dTLB MPKI is more sensitive to page-cross prefetching than sTLB MPKI because dTLB is smaller than sTLB, as Section IV shows, and translations brought by page-cross prefetches are stored in both dTLB and sTLB structures, thus a useful page-cross prefetch that hits in dTLB does not impact the sTLB MPKI. Focusing on the cache MPKIs of Figure 4a, we observe a large L1D MPKI reduction since Berti accurately prefetches across page boundaries for this workload set. This L1D MPKI reduction translates to an LLC MPKI reduction, justifying why

[1]For the rest of the paper, we use the terms dTLB and sTLB to refer to the first-level data TLB and last-level TLB, respectively.



(a) Workloads of Figure 2 for which Permit PGC outperforms Discard PGC. Each bullet represents a workload.



(b) Workloads of Figure 2 for which Discard PGC outperforms Permit PGC. Each bullet represents a workload.

Fig. 4: Impact of Permit PGC on DTLB MPKI, STLB MPKI, L1D MPKI, and LLC MPKI over Discard PGC for Berti prefetcher. Reported results are raw MPKIs.

Berti with Permit PGC outperforms Berti with Discard PGC for this set of workloads.

Regarding the workloads for which Discard PGC outperforms Permit PGC (Figure 4b), we observe the opposite trends than the workloads for which Permit PGC outperforms Discard PGC (Figure 4a). Berti does not effectively prefetch across page boundaries for this set of workloads, significantly increasing dTLB, sTLB, L1D, and LLC MPKIs, justifying why Discard PGC outperforms Permit PGC for these workloads. The increase in dTLB MPKI is higher than the sTLB MPKI increase because inaccurate cross-page prefetches have a higher impact on dTLB than sTLB since the former is smaller than the latter. The same applies for L1D and LLC.

The main takeaway of this study is that optimizing page-cross prefetching has the potential to provide significant benefits. An adaptive mechanism capable of accurately deciding whether a given page-cross prefetch is useful or not, and thus issue or discard the corresponding prefetch request, can significantly improve the effectiveness of cache prefetchers and the overall performance of the system.

A scheme able to accurately enable page-cross prefetching only when is beneficial has the potential to deliver significant performance enhancements.

1) Putting Everything Together: Our analysis across three well-established L1D prefetchers (Berti, IPCP, BOP) and a set of memory-intensive workloads demonstrates that (i) there is significant performance on the table by optimizing page-cross prefetching and (ii) state-of-the-art L1D prefetchers are not very accurate at page-cross prefetching. Our findings motivate the need for a dynamic microarchitectural mechanism that enables page-cross prefetching only when it is proven to be beneficial for performance.

III. Page-Cross Prefetch Filtering

This work introduces MOKA, a new framework for designing effective *Page-Cross Filters* that ensure accurate prefetching across page boundaries. MOKA consists of microarchitectural predictors that leverage (1) various program features (*e.g.*, PC) to predict the usefulness of prefetches that cross page boundaries, (2) various system features (*e.g.*, sTLB MPKI) to take into account the system state when deciding whether to issue or not a page-cross prefetch, and (3) an adaptive scheme that tunes the decision thresholds at runtime to ensure accurate predictions across different execution phases and workloads.

A. Design Overview

Figure 5 illustrates the design and the high-level operation of the MOKA framework assuming a VIPT L1D [12], [15]. Upon L1D accesses, the L1D prefetcher produces prefetch requests. For each prefetch request, MOKA checks whether it is a page-cross prefetch or not (A). The *Page-Cross Filter* is activated only for prefetches that cross page boundaries (B) and decides whether to issue or not the corresponding requests. Prefetches that pass the Page-Cross Filter need to go through the TLB hierarchy to find the translation of the page where the prefetched block resides (C). If the requested translation is not TLB resident, a speculative page walk is triggered to bring the corresponding translation in the TLB hierarchy (D). Finally, the page-cross prefetch is issued and eventually the corresponding prefetched block is stored in L1D.

B. Hardware Components of the Page-Cross Filter

The Page-Cross Filter, designed using the MOKA framework, uses five hardware components to decide whether to permit or discard a page-cross prefetch (step (B) in Figure 5):

Perceptron Predictors. The MOKA framework provides hashed perceptron predictors [17], [85] realized with *Weight Tables (WTs)* storing perceptron weights associated with selected program features; the perceptron weights are implemented with saturating counters [32], [35], [44]–[47], [58], [59], [72], [85], [86]. The Page-Cross Filter uses one hashed perceptron predictor for each selected program feature. Section III-D1 presents the considered program features.

Saturating Counters for System Features. The MOKA framework also provides system features, *i.e.*, features that associate the usefulness of page-cross prefetching with the system state (*e.g.*, TLB pressure). System features contribute on predicting whether a page-cross prefetch is useful or not by taking into account the system state in the decision making of the Page-Cross Filter. System features are implemented with saturating counters; we name them system feature weights. Section III-D2 presents the considered system features.

Virtual Update Buffer (vUB). MOKA uses vUB for training purposes. vUB is responsible for capturing false negatives, *i.e.*, cases where the Page-Cross Filter erroneously discarded a page-cross prefetch that would have saved a demand L1D miss if it was issued. Each vUB entry stores the virtual address of a page-cross prefetch that the Page-Cross Filter decided not to issue coupled with the corresponding hash index to ensure correct updating of the weights (for both program and system features). Note that vUB stores virtual addresses since L1D prefetchers operate with virtual addresses. Section III-C2 describes the contribution of vUB in the training of both program and system features.

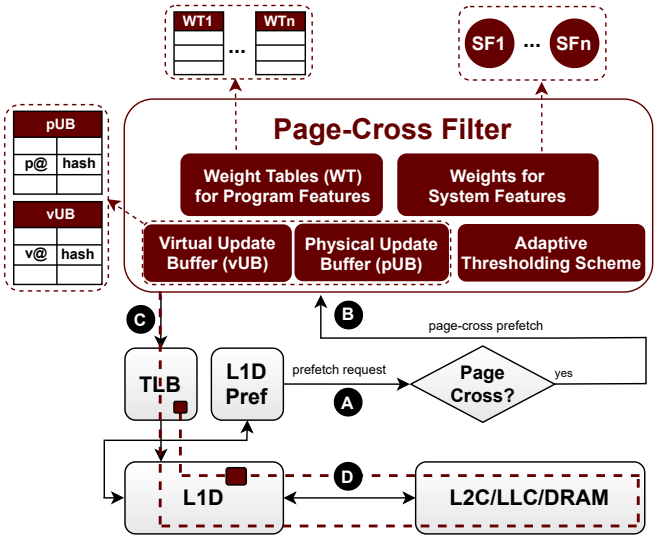


Fig. 5: Overview of the MOKA framework.

Physical Update Buffer (pUB). The Page-Cross Filter uses pUB to ensure correct updating of the weights (for both program and system features) depending on the usefulness of the issued page-cross prefetches. A page-cross prefetch is considered useful when it serves at least one L1D demand access before eviction; otherwise, it is considered useless. To do so, pUB entries store the physical addresses of page-cross prefetches that the Page-Cross Filter decided to issue coupled with the corresponding hash indexes. Note that pUB stores physical addresses (and not virtual addresses as vUB) for training purposes as it updates the weights upon L1D evictions and L1Ds are physically-tagged (Section III-C2).

Adaptive Thresholding Scheme. To determine the usefulness of a page-cross prefetch, the Page-Cross Filter compares a cumulative weight (sum of program feature weights and system feature weights) with a threshold; prefetches with cumulative weights higher than the threshold are issued while the others are discarded (Section III-C1). Using a static threshold provides suboptimal gains due to workload heterogeneity and phase-changing behaviors. The MOKA framework employs an epoch-based mechanism (Section III-C3) that exploits various runtime information (*e.g.*, LLC pressure) to tune the threshold used in decision making.

C. Operation

1) Prediction: Upon an L1D prefetch that crosses page boundaries (step (B) in Figure 5), the Page-Cross Filter is activated to predict the usefulness of the corresponding prefetch request and decide whether to issue or discard it. Figure 6 shows the decision making of the Page-Cross Filter.

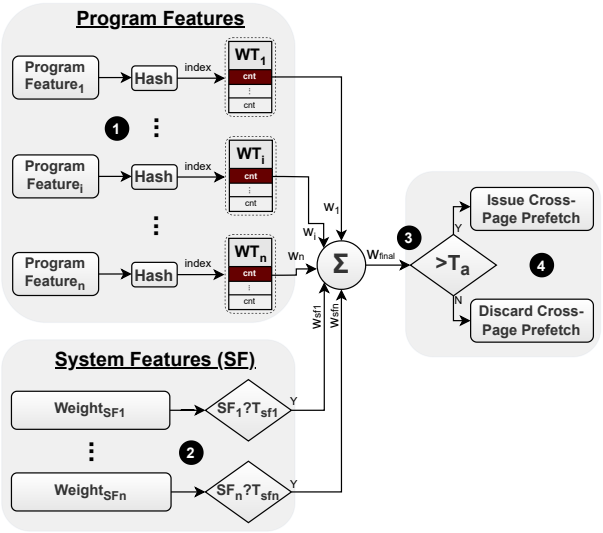


Fig. 6: Prediction operation of the Page-Cross Filter.

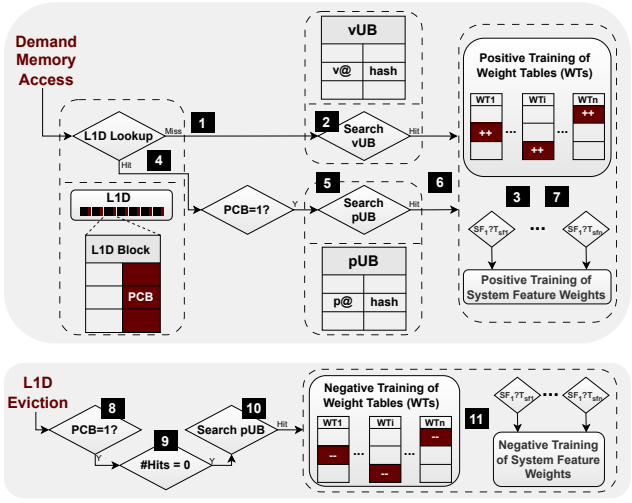


Fig. 7: Training of the Page-Cross Filter.

The prediction happens in four stages. The first stage ❶ consists of three steps. First, the Page-Cross Filter extracts the set of selected program features (Section III-D1) from the current load request. Then, each feature is hashed and used to index the corresponding WT. After indexing, a weight (w_i) is retrieved from each WT. The same procedure takes place for all program features. The second stage ❷ checks if any of the system features (Section III-D2) has to be considered in the decision making. To do so, it computes the value of each system feature (*e.g.*, sTLB MPKI) and compares it with a threshold ($SF_n ? T_{sfn}$ in Figure 6); if the system feature value (SF_n) exceeds (or subceeds, depending on the feature) the associated threshold (T_{sfn}), then the corresponding weight ($weight_{SFn}$) is taken into account in the decision making. Note that system features contribute to the final decision only during specific phases, *i.e.*, when $SF_j ? T_{sfj}$ (? could be $>$ or $<$, depending on the feature). Section III-D2 presents the rationale behind system features. The third stage ❸ sums the weights of all considered program and system features and generates the final weight (w_{final}). Then, w_{final} is compared with an activation threshold T_a ❹. If w_{final} is greater than T_a, the page-cross prefetch is issued; otherwise it is discarded.

2) Training: To ensure correct updating of both program and system feature weights, MOKA uses two structures: the *Virtual Update Buffer (vUB)* and the *Physical Update Buffer (pUB)*, presented in Section III-B. Moreover, MOKA augments each L1D block with an additional bit, named *Page Cross Bit (PCB)*, indicating whether the block has been fetched in L1D by a page-cross prefetch or not.

The training of the Page-Cross Filter is triggered upon L1D demand accesses and L1D evictions. Figure 7 depicts the training operation across all trigger events. Upon L1D demand misses 1, vUB is searched for possible hits 2. A vUB hit 3 indicates that the corresponding page-cross prefetch was erroneously discarded by the Page-Cross Filter, thus the hash indexes of the hit vUB entry are used to increment the respective program and system weights (positive training), increasing the probability of issuing (permitting) the corresponding page-cross prefetch in the future. Similarly, when a demand request hits in L1D 4 and the hit block has its PCB set, *i.e.*, the block was fetched by a useful page-cross prefetch, the pUB is looked up 5. Upon pUB hits 6, the hash indexes of the hit pUB entry are used to increase the weights of the respective program and system features 7.

Upon an L1D eviction of a block with PCB set 8, the Page-Cross Filter checks whether the evicted block has provided at least one hit during its lifetime in the cache 9. If not, pUB is searched to find the matching entry 10 since pUB stores the physical addresses and the hash indexes of the issued page-cross prefetches; this is the reason why pUB stores physical addresses and not virtual addresses as vUB (Section III-B). The hash indexes of the matching pUB entry are used to decrease the respective program and system weights (negative training) 11 since evicted L1D blocks that did not provide any hit indicate that the Page-Cross Filter failed to classify the corresponding page-cross prefetches as useless.

3) Adaptive Thresholding Scheme: During the third stage of the prediction (❸ in Figure 6), the Page-Cross Filter compares the cumulative weight (w_{final}) with the activation threshold T_a (Section III-C1). A static threshold T_a is adequate when targeting to improve the performance of a specific application type, but it provides suboptimal gains when the target is multiple diverse application domains. We empirically find that different application profiles and execution phases have different optimal T_a values. To address this, we design an epoch-based adaptive scheme that takes into account various heuristics to dynamically adjust the activation threshold T_a.

Figure 8 presents the operation of the adaptive thresholding scheme. During an epoch, the following runtime statistics (step ❶ in Figure 8) are collected and used to tune T_a: number of useful and useless page-cross prefetches, IPC, LLC miss rate, ROB pressure, and L1I MPKI. In addition to the collection of statistics, during an epoch the thresholding

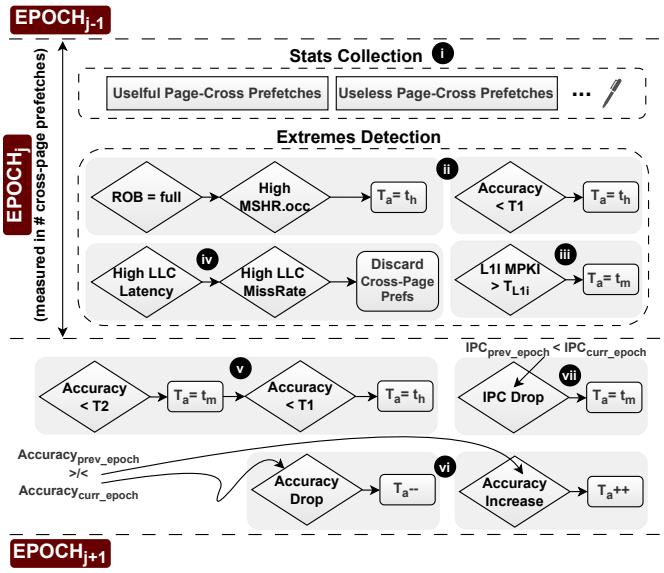


Fig. 8: Operation of the Adaptive Thresholding Scheme.

scheme detects extreme behaviors, *i.e.*, phases with very high cache and ROB pressure and adjusts the value of T_a on the spot. Specifically, it sets T_a to a high threshold (t_h) to only permit page-cross prefetches with very high confidence ii in the following cases: (1) there is high ROB pressure and many inflight L1D misses and (2) the accuracy of page-cross prefetching has reached a low value (T_1). Moreover, the thresholding scheme sets T_a to a medium value (t_m) when there is high L1I pressure ($L1i\ MPKI > T_{L1i}$) iii to avoid exacerbating the contention between page-cross prefetches and demand instruction accesses in the L2C. Finally, during phases with very high LLC pressure, the thresholding scheme disables page-cross prefetching iv; if LLC pressure drops, then page-cross prefetching might be activated again thanks to vUB's operation (Section III-C2).

At the end of an epoch, the thresholding scheme potentially updates T_a using the runtime information collected during the previous epoch. Specifically, it takes into account the accuracy of page-cross prefetching v and it forces a medium or high threshold when the accuracy is lower than the thresholds T_2 and T_1, respectively. Moreover, if there is an increase (decrease) in page-cross prefetching accuracy between two consecutive epochs vi, the thresholding scheme increases (decreases) the T_a value by one. Finally, if there is a drop in IPC between two consecutive epochs vii, the thresholding scheme sets T_a to t_m (if T_a was lower than t_m).

D. Bouquet of Features

1) Program Features: The MOKA framework comes with a wide range of program features that can be used to design a Page-Cross Filter for any given prefetcher. In total, MOKA contains 55 program features crafted using our expertise as well as prior work in domain [17], [47]. Table I presents the subset of program features, identified by offline feature exploration, that (i) correlate best with page-cross prefetching patterns and discard harmful for performance page-cross prefetches and (ii) provide the highest performance gains in isolation among the evaluated prefetchers. Table I reveals that most program features use different bits of the virtual address, PC, and the delta used by the prefetcher when issuing page-cross prefetches. Note that the MOKA framework contains program features that are not specialized to a specific prefetcher but are transparent to which prefetcher is used. Crafting specialized features that exploit metadata of specific prefetchers (*e.g.*, lookahead) has the potential to further improve the effectiveness of a Page-Cross Filter. Our work does not focus on a specific prefetcher but rather provides a holistic scheme for effective page-cross prefetching.

Program Features	•VA •VA≫12 •VA≫21 •CacheLineOffset •PC •PC+CacheLineOffset •VA_{i-2}⊕VA_{i-1}⊕VA_i •(VA_{i-2}≫12)⊕(VA_{i-1}≫12)⊕(VA_i≫12) •(PC_{i-2}⊕PC_{i-1}⊕PC_i) •PC⊕VA •PC⊕(VA≫12) •VA⊕Delta •PC⊕Delta •(VA≫12)⊕Delta •PC⊕FirstPageAccess •VA⊕FirstPageAccess •(VA≫12)⊕FirstPageAccess •CacheLineOffset+FirstPageAccess •Delta+FirstPageAccess
System Features	•L1D MPKI •L1D Miss Rate •LLC MPKI •LLC Miss Rate •sTLB MPKI •sTLB Miss Rate

TABLE I: Best performing program and system features.

2) System Features: Program features (Section III-D1) are critical for the accuracy and the performance of a Page-Cross Filter. However, they do not take into account the system state (*e.g.*, TLB/cache pressure) in their decision making. In other words, a page-cross prefetch that has been proven useful (useless) in the past might be useless (useful) during different execution phases with different properties. The MOKA framework considers 6 system features, presented in Table I, to capture these scenarios. Each system feature is implemented with a saturating counter and monitors the usefulness of page-cross prefetching in different phases. For example, the sTLB Miss Rate system feature monitors the usefulness of page-cross prefetching during phases where the sTLB Miss Rate is above a pre-defined threshold ($SF_{sTLB\_missrate} > T_{sTLB\_missrate}$).

3) Combining Features: The feature selection process takes place offline and uses IPC speedup as optimization metric.[2] First, we quantify the IPC speedups of each program and system feature in isolation by evaluating 60 different single-feature Page-Cross Filters (Table I) across 218 seen workloads (Section IV-A). Then, features are sorted in increasing order of geomean IPC speedup. The final round combines different features to further increase the IPC gains. We start with an initial set that contains only the best performing feature. Then, we examine whether considering additional features improves geomean IPC speedup. If a new feature proves to be useful, *i.e.*, improves geomean IPC by more than 0.3% over the best configuration so far, it is included in the set of selected features. We follow the same procedure until all features have been examined. This process is repeated for each prefetcher considered.

[2]Alternative optimization metrics are page-cross prefetching coverage and accuracy. We do not use coverage because different misses have different criticality for performance. We do not optimize for accuracy because it often leaves performance on the table due to the requirement of high accuracy.

Berti [60]	Delta, sTLB MPKI, sTLB Miss Rate
BOP [57]	PC⊕Delta, sTLB MPKI, sTLB Miss Rate
IPCP [61]	PC⊕Delta, sTLB MPKI, sTLB Miss Rate

TABLE II: DRIPPER features per considered prefetcher.

E. DRIPPER: A Page-Cross Filter Prototype

The MOKA framework can be used to design Page-Cross Filters for any given prefetcher. To prove the versatility of MOKA, we use it to prototype a Page-Cross Filter for three L1D prefetchers (Section V-A) by following the feature selection process described in Section III-D3. We refer to these Page-Cross Filter prototypes as *DRIPPER*. Table II presents the selected features of DRIPPER for all L1D prefetchers.

We observe that all DRIPPER configurations (Table II) use one program feature and two system features. The same system features (sTLB MPKI, sTLB Miss Rate) are selected for all considered prefetchers, while two prefetchers (BOP, IPCP) also have in common the used program feature (PC⊕Delta). The rationale behind the selected features is the following:

• ***Delta.*** This program feature uses the delta used by the L1D prefetcher to issue a page-cross prefetch and learns whether specific deltas are beneficial or not when they are used to generate page-cross prefetch requests.

• ***PC⊕Delta.*** This program feature is computed by XOR-ing the PC with the delta used by the prefetcher to issue a page-cross prefetch and provides information of whether a certain PC favors specific delta(s) used for page-cross prefetching.

• ***sTLB MPKI.*** This system feature correlates phases with low sTLB MPKI rates ($SF_{sTLB\_mpki} < T_{sTLB\_mpki}$) with the usefulness of page-cross prefetching. The core idea behind this feature is that when sTLB MPKI is low, the probability of a page-cross prefetch request hitting in the TLB hierarchy is high (the probability of triggering a page walk is low). Hence, if page-cross prefetching is useful/useless during these phases, this feature makes DRIPPER more/less aggressive towards page-cross prefetching by increasing the final cumulative weight (step ❸, Figure 6). This system feature contributes to the decision making only when $SF_{sTLB\_mpki} < T_{sTLB\_mpki}$.

• ***sTLB Miss Rate.*** This system feature is complementary to the sTLB MPKI system feature since it targets phases with high sTLB pressure whereas the sTLB MPKI system feature targets phases with low sTLB pressure. We use the sTLB Miss Rate metric instead of the sTLB MPKI metric to capture phases with high sTLB pressure since the former is more sensitive to changes than the latter. This system feature measures the usefulness of page-cross prefetching during phases with high sTLB Miss Rate ($SF_{sTLB\_missrate} > T_{sTLB\_missrate}$) to identify missed opportunities that other features cannot capture. The core idea is that when most demand memory accesses miss in the sTLB, permitting the L1D prefetcher to trigger page-cross prefetches might improve the sTLB hit rate due to the page walks introduced for page-cross prefetches. Hence, if page-cross prefetching is proven useful during phases with high sTLB Miss Rate, this feature increases the probability that DRIPPER will permit page-cross prefetching. This feature is used only when $SF_{sTLB\_missrate} > T_{sTLB\_missrate}$.

1) Storage Overhead: Table III presents the storage overhead of DRIPPER. This overhead is the same across all evaluated L1D prefetchers (Section V-A) since their DRIPPER versions use one program and two system features (Table II). In total, DRIPPER requires 1.44KB of storage per core; the number of WT, vUB, and pUB, entries are empirically selected after tuning. DRIPPER aims at low storage overheads to make its implementation feasible for a real design. A design that can dedicate tens of KBs for a Page-Cross Filter can use additional features from Table I that improve performance for a few workloads but their impact on geomean IPC speedup is small.

Program Features	1×512×5bits	**0.625KB**
System Features	sTLB MPKI: 5bits, sTLB MissRate: 5bits	**0.00125KB**
vUB, pUB	4×(36+12)bits, 128×(36+12)bits	**0.024KB**, **0.768KB**
Total		**1.44KB**

TABLE III: Storage overhead of DRIPPER.

IV. Experimental Methodology

We evaluate our proposal using the ChampSim simulator [5], [34], modeling 1-core and 8-core out-of-order processors with 3-level cache hierarchy [60], [89] and a decoupled front-end [41], [70]. We simulate a 5-level radix tree page table, an x86 hardware page table walker [24], and split Page Structure Caches [21], [67]. The hardware page table walker models (i) the variant latency cost of page walks, (ii) the page walk references to memory hierarchy, and (iii) the cache locality in page walks. Prefetching for data/instructions is applied upon L1D/L1I accesses with prefetched blocks filled in the corresponding cache. We evaluate different L1D prefetchers to show the versatility of our proposal. No prefetching is applied at the lower-level caches [8] but Section V-B7 provides evaluation when lower-level cache prefetchers are used. Table IV presents our experimental setup in detail.

Simulated Page Sizes: Our evaluation is mainly focused on 4KB pages, similar to prior work [60], [61]. Section V-B6 presents evaluation when the system utilizes both 4KB and 2MB large pages, using the methodology of prior work [89].

A. Workloads

We evaluate our proposal using a diverse set of workloads, spanning various benchmark suites and workload domains. We use (i) general purpose workloads from SPEC CPU 2006 [9], [40], SPEC CPU 2017 [10], and Geekbench [7], [50] benchmark suites, (ii) big memory footprint workloads included in the GAP [16] and Ligra [78] benchmark suites, (iii) parallel workloads from PARSEC [11], and (v) industrial integer and floating point workloads provided by Qualcomm for the 1st Contest on Value Prediction (CVP-1) [4], [30]. For the rest of the paper, we use SPEC, GKB5, QMM_{INT}, and QMM_{FP} to refer to the SPEC CPU 2006 and SPEC CPU 2017, Geekbench, Qualcomm integer, and Qualcomm floating point workloads, respectively. All traces were obtained using the SimPoint methodology [68]. SimPoints with a weight smaller than 0.05 are discarded and we report weighted geomean speedups [17], [18], [47], [57], [60], [61], [63], [89].

CPU Core	1-8 cores, 4GZ, 352-entry ROB, 6-wide issue, hashed perceptron branch predictor [85]
L1I TLB (iTLB)	64-entry, 4-way, 1-cycle lat, 8-entry mshr, LRU
L1D TLB (dTLB)	64-entry, 4-way, 1-cycle lat, 8-entry mshr, LRU
L2 TLB (sTLB)	1536-entry, 12-way, 8-cycle lat, 16-entry mshr, LRU
Page Structure Caches (PSCs)	4-level Split PSC, parallel search, 1-cycle lat. L5: 1-entry, L4: 2-entry, L3: 8-entry, L2: 32-entry
L1I Cache	32KB, 8-way, 4-cycle lat, 8-entry mshr, LRU, fnl-mma [75]
L1D Cache	48KB, 12-way, 5-cycle lat, 16-entry mshr, LRU
L2 Cache	512KB, 8-way, 10-cycle lat, 32-entry mshr, LRU
LLC (per core)	2MB, 16-way, 20-cycle lat, 64-entry mshr, LRU
DRAM	4GB (single-core), 16GB (8-core), 3200MT/s

TABLE IV: System Configuration.

Our main experimental campaign focuses on workloads with an LLC MPKI of at least 1 as we consider them memory-intensive. We use 396 workloads which we split into two categories: (i) 218 workloads that were used during the development of our proposal, DRIPPER, and (ii) 178 unseen workloads that we did not take into account during the design of DRIPPER. Section V-B8 presents evaluation for the unseen workloads. We also show that DRIPPER does not harm the performance of non memory-intensive workloads by presenting evaluation across the entire benchmark suites, including the non-intensive workloads (Section V-B9).

1) Single-core Experiments: SPEC, GAP, PARSEC, and LIGRA workloads run the first 250M instructions to warm up the uarch structures and the subsequent 250M instructions are executed to obtain the experimental results. GKB5 workloads use variable warm-up and simulation instructions depending on the simpoint length. Finally, for the Qualcomm workloads, 50M warm-up instructions and 100M simulation instructions are used, similar to prior work [58], [87], [88].

2) Multi-core Experiments: Our multi-core evaluation considers 300 randomly generated mixes from our workload set. Our evaluation reports the weighted speedup over the baseline [20], [47], [60], [89]. First, we compute the IPC on the multi-core context and the IPC in isolation on a system with the multi-core configuration for each application running on a core. Next, we compute the weighted IPC as the sum of ($IPC_{multicore}/IPC_{isolation}$) for all workloads in the mix. Finally, we normalize this sum with the weighted IPC of the baseline. For each mix, when a core finishes its instructions, it gets replayed until all the cores finish their respective instructions, similar to prior work [17], [18], [47], [60], [61], [63], [89].

V. Evaluation

A. DRIPPER for Different L1D Prefetchers

This section highlights the benefits and versatility of our proposal by implementing a DRIPPER prototype for three L1D prefetchers: Berti [60], IPCP [61], and BOP [57]. Section III-E and Table II present the design and configuration of DRIPPER for all considered prefetchers. We compare DRIPPER against the following scenarios that exploit/optimize page-cross prefetching:

• ***Permit PGC.*** The L1D prefetcher is constantly permitted to issue page-cross prefetches, similar to Section II-C.

• ***Discard PGC.*** The L1D prefetcher is never permitted to issue page-cross prefetches, similar to Section II-C.

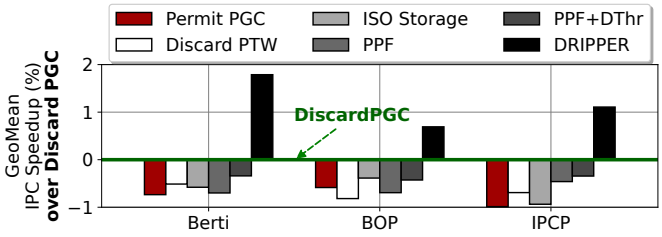


Fig. 9: Performance comparison between different schemes that exploit page-cross prefetching. IPC speedups are computed over a baseline that does not permit the L1D prefetcher to cross page boundaries (Discard PGC).

• ***Discard PTW.*** The L1D prefetcher is permitted to cross pages for prefetching only when the corresponding translation is found in the TLB hierarchy; if not, the prefetch request is discarded to avoid triggering a speculative page walk.

• ***ISO Storage.*** The storage overhead of DRIPPER (Section III-E1) is used to enlarge the most relevant for performance structure(s) of the underlying prefetcher (Berti, BOP, IPCP). This scenario permits page-cross prefetching.

• ***Perceptron-based Prefetch Filtering (PPF) [20].*** Originally, PPF uses program features to filter out inaccurate L2C prefetches issued by the SPP prefetcher [48] which operates in the physical address space. We convert PPF to work as a page-cross filter for L1D prefetchers, excluding the program features that are specialized to SPP's metadata. Section VI elaborates on the differences between our proposal and PPF.

• ***PPF+Dthr.*** PPF [20] uses a pre-defined threshold to predict the usefulness of prefetches. We combine PPF with the dynamic thresholding scheme of MOKA (Section III-C3) to provide a direct comparison with DRIPPER.

Figure 9 compares DRIPPER with the considered scenarios that exploit page-cross prefetching in terms of single-core geometric mean performance over a baseline that does not permit the L1D prefetcher to cross page boundaries (Discard PGC) across all considered prefetchers (Berti, BOP, IPCP) and a set of 218 workloads (Section IV-A). We use Discard PGC as the baseline because prior L1D prefetchers are typically restricted to prefetch within page boundaries. Figure 9 reveals that DRIPPER provides the highest geomean IPC gains across all considered scenarios. The main takeaways of this study are:

- The scenario that always discards page-cross prefetches (Discard PGC) outperforms the scenario that always issues page-cross prefetches (Permit PGC) because crossing pages for prefetching is a high-risk technique; when it is inaccurate can deteriorate performance. However, there are benchmarks that enjoy great benefits when page-cross prefetching is permitted, as Sections II-C and V-B indicate.

- The scenario that discards page-cross prefetches for which the translation is not found in the TLB (Discard PTW) slightly outperforms the Permit PGC scenario for Berti and IPCP prefetchers (still lower than the performance of Discard PGC) because it does not trigger speculative page walks for page-cross prefetches that avoid polluting the cache and the TLB

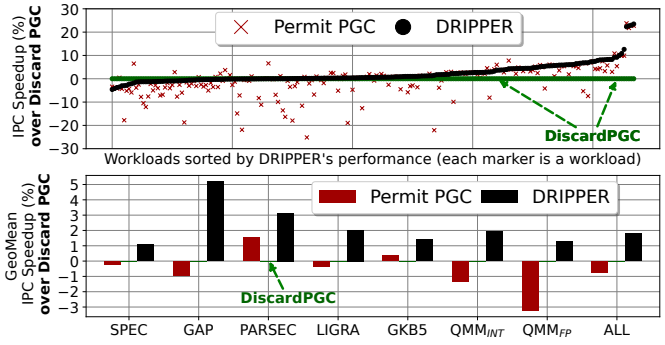


Fig. 10: Performance of Permit PGC and DRIPPER when Berti is used at L1D over the same baseline as Figure 9.

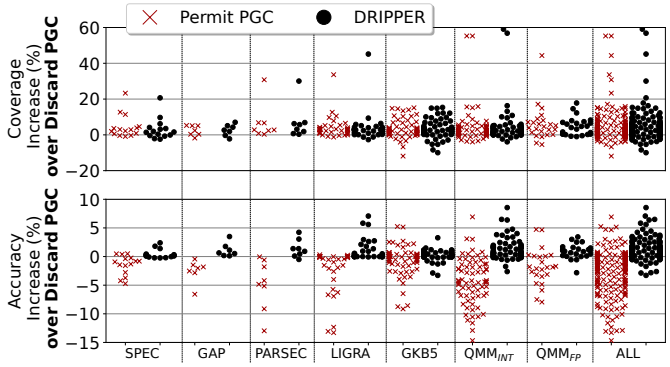


Fig. 11: Coverage (top) and Accuracy (bottom) comparison between Permit PGC and DRIPPER when Berti is used as L1D prefetcher over the same baseline as Figure 9.

in case of inaccurate page-cross prefetching. However, this scenario leaves significant performance on the table when page-cross prefetching is beneficial for performance.

- The ISO Storage scenario yields performance similar to the Permit PGC scenario. We conclude that increasing the storage budget of an L1D prefetcher does not improve its effectiveness with respect to page-cross prefetching.

- PPF and PPF+Dthr do not improve performance over Discard PGC. PPF fails to form an effective filter for page-cross prefetching because (i) it does not consider system features, *i.e.*, features that take into account the system state in the decision making (Section III-D2) and (ii) its set of program features is suboptimal for page-cross prefetching.

- DRIPPER provides the highest IPC gains across the considered scenarios and prefetchers thanks to its effectiveness in filtering useless page-cross prefetches. DRIPPER outperforms (in geomean) PPF by 2.4%, 1.4%, and 1.6% when the L1D prefetcher is Berti, BOP, and IPCP, respectively. DRIPPER offers gains for all considered prefetchers since it uses program and system features that correlate well with page-cross prefetching and mostly issue beneficial-for-performance page-cross prefetches. DRIPPER offers lower gains for BOP since BOP has lower prefetching capabilities than IPCP and Berti.

B. Case Study: Berti Prefetcher

This section quantifies the gains of DRIPPER at a finer granularity than Section V-A and justifies its benefits. To do so, it focuses on the state-of-the-art L1D prefetcher, Berti.

1) Single-Core Performance: Figure 10 (top) presents the performance of Berti when combined with the Permit PGC scenario and DRIPPER over a baseline that uses Berti with Discard PGC, similar to Figure 9, across all workloads. Figure 10 (bottom) shows the breakdown of the geomean speedups across the considered benchmark suites over the same baseline.

Looking at Figure 10 (top), we observe that DRIPPER provides higher performance gains than both Permit PGC and Discard PGC scenarios for the vast majority of the considered workloads since it accurately predicts the usefulness of the page-cross prefetch requests and issues only the useful ones. Permit PGC slightly outperforms DRIPPER for a few workloads because DRIPPER is conservative to page-cross prefetching until it achieves high confidence. Moreover, DRIPPER provides lower performance than Discard PGC for a few workloads (tail in Figure 10 (top)) because DRIPPER permits page-cross prefetching during a few phases that is not beneficial for performance; this happens mostly for short-running workloads (QMM_INT and QMM_FP).

Figure 10 (bottom) highlights that DRIPPER provides the highest IPC gains across all considered benchmark suites. Overall, DRIPPER outperforms Permit PGC and Discard PGC by 2.5% and 1.7% in geomean, respectively. DRIPPER delivers the highest speedups over Discard PGC for the GAP suite because these benchmarks put high pressure on both cache and TLB hierarchies, thus effective page-cross prefetching improves the timeliness of prefetching and reduces the address translation overheads due to page walks introduced for page-cross prefetches. Finally, Figure 10 (top, bottom) shows that Permit PGC provides benefits for a subset of the workloads but for most workloads it harms performance, justifying why Discard PGC provides higher speedups in Figure 9.

2) Coverage and Accuracy: To explain the benefits of DRIPPER, Figure 11 compares the miss coverage and prefetching accuracy of Berti when combined with Permit PGC and DRIPPER over Berti with Discard PGC across all benchmark suites. Coverage and accuracy metrics consider all prefetch requests (both in-page and page-cross prefetches).

Figure 11 (top) reveals that DRIPPER achieves almost the same coverage results as the Permit PGC scenario in the benchmark suites considered. Specifically, Berti with Permit PGC and DRIPPER improves average miss coverage over Discard PGC by 4.2% and 4.1% across all considered workloads, respectively. The main takeaway is that DRIPPER can identify which page-cross prefetches are useful and issue them.

Figure 11 (bottom) highlights that Berti with DRIPPER achieves significantly higher accuracy than Berti with Permit PGC; we observe the same trends across all considered suites. Overall, DRIPPER improves the average accuracy over Discard PGC by 1.2% while Permit PGC reduces prefetching accuracy over Discard PGC by 2.6% across all workloads. The

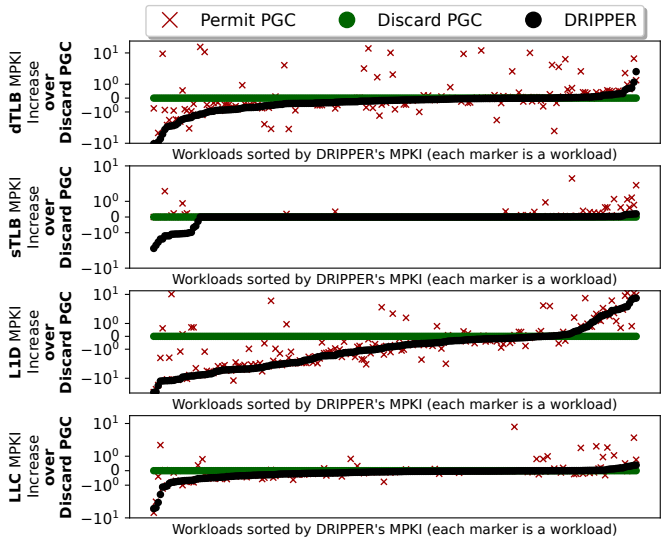


Fig. 12: Impact of Permit PGC and DRIPPER on dTLB, sTLB, L1D, and LLC MPKIs when Berti is used at L1D over Discard PGC, similar to Figure 9). Reported results are raw MPKIs not percentages. Lower is better.

bottom line is that DRIPPER accurately identifies the majority of the useless page-cross prefetches and discards them.

The miss coverage and accuracy results of Figure 11 justify the performance enhancements of DRIPPER, presented in Figure 10. DRIPPER achieves the same coverage as Permit PGC while increasing the accuracy of prefetching thanks to accurately discarding useless page-cross prefetches.

3) Impact on TLBs and Caches: To further explain the benefits of DRIPPER, Figure 12 compares the impact of Berti+Permit PGC and Berti+DRIPPER on dTLB MPKI, sTLB MPKI, L1D MPKI, and LLC MPKI over a baseline with Berti+Discard PGC, similar to previous sections. Looking at Figure 12, we observe the same trends as in Figure 10 (top). Note that DRIPPER's curves in Figure 12 and Figure 10 (top) are flipped since the former shows MPKI increase (lower is better) and the latter IPC speedups (higher is better).

DRIPPER provides higher reduction in dTLB, sTLB, L1D, and LLC MPKIs than both Permit PGC and Discard PGC for the vast majority of the workloads since it accurately predicts the usefulness of page-cross prefetches, justifying its performance, coverage, and accuracy gains (Figures 10 and 11). On average, DRIPPER reduces dTLB, sTLB, L1D, and LLC MPKIs in absolute values over Discard PGC by 0.6, 0.1, 2.1, and 0.2, respectively. DRIPPER has a higher impact on dTLB MPKI than sTLB MPKI since dTLB is smaller than sTLB (Table IV), thus more sensitive to page-cross prefetching. Focusing on the L1D and LLC MPKIs, we observe a large L1D MPKI reduction for many workloads. The L1D MPKI reduction sometimes translates to an LLC MPKI reduction, further explaining why DRIPPER outperforms Discard PGC and Permit PGC scenarios. Permit PGC provides higher MPKI reductions than DRIPPER for a few workloads because DRIPPER is conservative to page-cross prefetching until it reaches high confidence. Finally, we note that DRIPPER increases MPKIs over Discard PGC for a few workloads (head in Figure 12) because DRIPPER permits page-cross prefetching during a few phases that is not beneficial for performance, as explained in Section V-B1.

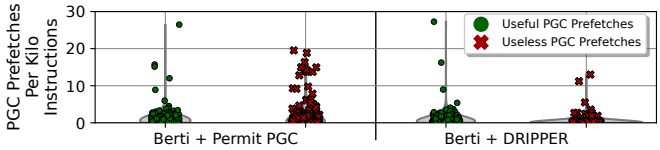


Fig. 13: Distribution of useful and useless page-cross prefetches for Permit PGC and DRIPPER.

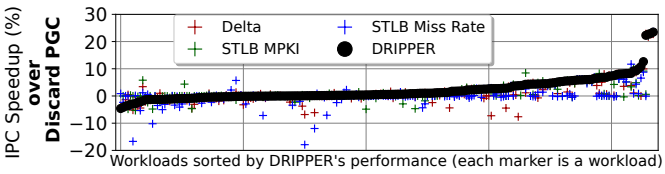


Fig. 14: Performance comparison between DRIPPER and its constituent features, over the same baseline as Figure 9.

4) Usefulness of Page-Cross Prefetching: Figure 13 complements Figure 11 by presenting the distribution of useful and useless page-cross prefetches of Berti+Permit PGC and Berti+DRIPPER, using the page-cross prefetches per kilo instructions metric. There are two takeaways from this study. First, the distribution of useful page-cross prefetches of Permit PGC and DRIPPER is very similar, *i.e.*, Permit PGC and DRIPPER have almost the same number of L1D hits due to page-cross prefetches, justifying the coverage results of Figure 11 (top). Second, the distribution of useless page-cross prefetches for DRIPPER is concentrated around zero while the one for Permit PGC has much higher values, showing that DRIPPER issues much fewer useless page-cross prefetches than Permit PGC, justifying the accuracy results of Figure 11 (bottom). The bottom line is that DRIPPER has almost the same number of useful page-cross prefetches and significantly fewer useless page-cross prefetches than Permit PGC.

5) Comparison with Individual Features: Figure 14 compares DRIPPER with its constituent features (Delta, sTLB MPKI, sTLB Miss Rate in Table II). To do so, we implement three page-cross filters. Each filter uses only one of DRIPPER's constituent features. Results in Figure 14 are computed over Discard PGC, similar to previous sections.

Looking at Figure 14, we observe that DRIPPER outperforms the single-feature page-cross filters for the vast majority of the considered workloads since it manages to effectively combine the benefits of its features. However, there are a few QMM_INT and QMM_FP workloads for which DRIPPER degrades performance over Discard PGC (negative tail in Figure 14), for the same reasons outlined Section V-B1.

Figure 15 compares DRIPPER with another version of DRIPPER that uses only the selected system features of Table

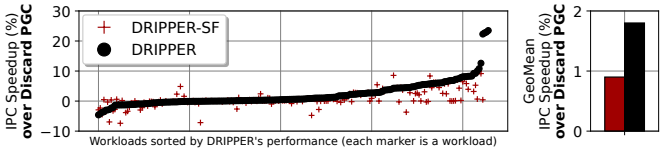


Fig. 15: Comparison between DRIPPER and a version of DRIPPER that uses only the system features (DRIPPER-SF).

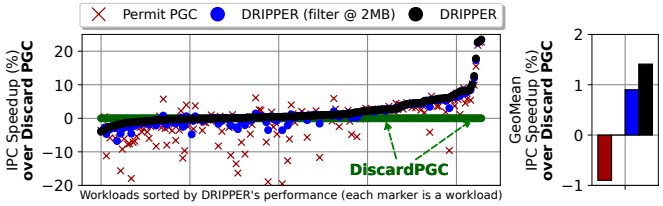


Fig. 16: Performance of Permit PGC, DRIPPER (filter@2MB), and DRIPPER when Berti is used as L1D prefetcher over a baseline that uses both 4KB and 2MB pages. Speedups are computed over Berti+Discard PGC, similar to Figure 9.

II, named DRIPPER-SF, to highlight the contribution of the selected program feature in the performance results. Figure 15 shows that DRIPPER outperforms DRIPPER-SF for the majority of the workloads (0.9% in geomean across 218 workloads) because it effectively combines the benefits of systems features with the benefits of program features while DRIPPER-SF does not consider any program feature.

6) Evaluation with Large Pages: This section considers a system that uses both 4KB and 2MB pages, following the methodology of Section IV. Figure 16 shows the performance of Berti combined with (a) Permit PGC, (b) DRIPPER(filter@2MB) that filters prefetch requests at (i) 4KB boundaries when the block resides in a 4KB page and (ii) 2MB boundaries when the block resides in a 2MB page, and (c) DRIPPER that filters prefetch requests at 4KB boundaries no matter the size of the page where the block resides over Berti with Discard PGC. Note that Permit PGC scenario is aware of the page size, thus it is equivalent to the proposal of [89] but operating in the virtual address space.

Figure 16 shows similar trends with previous sections. DRIPPER delivers higher IPC uplifts than both Permit PGC and Discard PGC for most workloads when both 4KB and 2MB pages are used (2.2% and 1.3% in geomean). DRIPPER outperforms DRIPPER(filter@2MB) by 0.5% in geomean because for blocks residing in 2MB pages DRIPPER stills decides about filtering prefetch requests at 4KB boundaries while DRIPPER(filter@2MB) decides about filtering prefetch requests at 2MB boundaries that limits its potential since there are only a few cases where the prefetcher is so aggressive to cross 2MB boundaries for a given access. This would require the prefetcher to start from a given 4KB page and cross 512 4KB pages to apply for a 2MB boundary crossing. We measure that more than 99% of page-cross prefetches across all tested prefetchers do not cross 2MB boundaries, thus filtering at

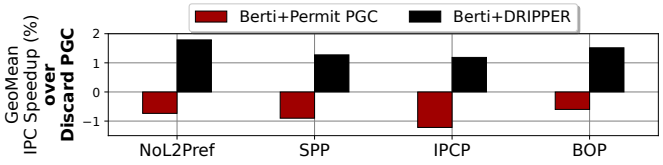


Fig. 17: Performance comparison when different L2C prefetchers (x-axis) are used in the baseline. Results are computed over Berti with Discard PGC, similar to Figure 9.

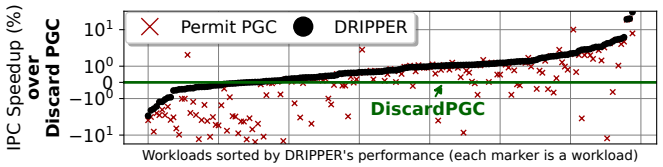


Fig. 18: Performance of Berti combined with Permit PGC and DRIPPER over Berti with Discard PGC for unseen workloads.

2MB boundaries performs similarly to Permit PGC scenario when the block resides in a 2MB page. DRIPPER provides the highest IPC gains across the considered approaches because it continues to accurately predict the usefulness of prefetch requests that cross 4KB boundaries no matter the size of the page where the prefetched block resides. For accurately filtered prefetch requests that cross 4KB boundaries it eliminates (i) both sTLB and cache pollution when the block resides in a 4KB page and (ii) cache pollution when the block resides in a 2MB page. Finally, DRIPPER provides slightly lower performance than Permit PGC for a few workloads because DRIPPER is conservative until it reaches high confidence and provides lower performance than Discard PGC for a few workloads because DRIPPER permits page-cross prefetching during a few phases that is harmful for performance, similar to Figure 10. The main takeaway of this study is that DRIPPER is also effective when both 4KB and 2MB pages are used.

7) Impact of L2C Prefetching: Previous sections present evaluation over a baseline without an L2C prefetcher. This section quantifies the impact of L2C prefetching on DRIPPER's performance. Figure 17 presents the geomean speedups of Berti combined with Permit PGC and Berti combined with DRIPPER over Berti combined with Discard PGC when the baseline uses different L2C prefetchers: NoL2Pref, SPP [48], IPCP [61], and BOP [57]. The trends are consistent with previous sections; Permit PGC degrades performance over Discard PGC and DRIPPER provides the highest speedups no matter which L2C prefetcher is used. The benefits of DRIPPER are slightly higher when there is no L2C prefetcher than when an L2C prefetcher is used because the L2C prefetcher saves misses that Berti's page-cross prefetching could also capture in the absence of an L2C prefetcher.

8) Unseen Workloads: This section quantifies the impact of DRIPPER on 178 unseen workloads (Section IV-A) which were not used during DRIPPER's design (Table II). Figure 18 presents the IPC speedups of Berti when combined with

	Seen	Unseen	All (Seen+Unseen+Non-Intensive)
Berti+Permit PGC	-0.8%	-0.9%	-0.6%
Berti+DRIPPER	+1.7%	+1.2%	+0.4%

TABLE V: Geomean speedups over Berti+Discard PGC across seen, unseen, and non-intensive workloads.

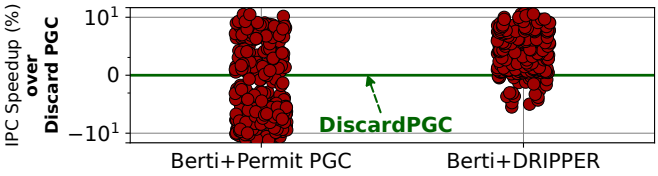


Fig. 19: Distribution of 8-core speedups of Permit PGC and DRIPPER over Discard PGC, similar to Figure 9.

Permit PGC and DRIPPER over Berti with Discard PGC across all unseen workloads. Overall, the trends for the unseen workloads are similar to the ones of the seen workloads (Figure 10): (i) Permit PGC provides benefits for only a subset of the unseen workloads, (ii) DRIPPER outperforms both Permit PGC and Discard PGC for most unseen workloads, and (iii) there are a few QMM_INT and QMM_FP workloads for which Discard PGC outperforms DRIPPER. Overall, DRIPPER outperforms (geomean) Permit PGC and Discard PGC by 2.1% and 1.2% for the unseen workloads, respectively. This study demonstrates that DRIPPER provides significant IPC gains for workloads that it has not been optimized for.

9) Non-Intensive Workloads: We quantify the impact of DRIPPER on non-intensive workloads by presenting evaluation across the entire benchmark suites (Section IV-A). Table V shows the geomean speedups of Berti with Permit PGC and DRIPPER over Berti with Discard PGC across seen, unseen (Section V-B8), and all workloads. We observe that (i) Permit PGC performs poorly while DRIPPER provides benefits across all workload sets, (ii) the speedups of DRIPPER are lower than the ones reported when considering only the memory-intensive workloads (seen, unseen) because the non-intensive workloads lower the reported geomean speedups, and (iii) DRIPPER provides significant benefits for the intensive workloads without harming the performance of non-intensive workloads.

10) Multi-Core Evaluation: This section quantifies the performance benefits of DRIPPER in the 8-core context. Figure 19 presents the IPC speedup distributions of Berti when combined with Permit PGC and DRIPPER over a baseline with Berti and Discard PGC across 300 randomly generated 8-core mixes (Section IV-A). We observe that DRIPPER improves performance over both Permit PGC and Discard PGC for the vast majority of the 8-core mixes. Specifically, DRIPPER improves geomean performance by 3.3% and 2.0% over Permit PGC and Discard PGC, respectively, across 300 8-core mixes. Note that DRIPPER is not tuned in the 8-core context; an exploration can provide even higher speedups.

VI. Related Work

To the best of our knowledge, this is the first work on analyzing and improving prefetching across page boundaries.

Prefetch Filtering. PPF [20] uses perceptrons to predict the usefulness of L2C prefetches. The key differences between PPF and DRIPPER are: (i) PPF uses only program features while DRIPPER uses both program and system features, (ii) PPF's design is not versatile since it uses features specialized to the SPP prefetcher [48], *i.e.*, features that use metadata specific to SPP that other prefetchers do not have while DRIPPER uses only prefetcher-independent features, (iii) PPF uses a static activation threshold while DRIPPER uses a scheme that adapts the activation threshold to different phases, and (iv) PPF always discards page-cross prefetches. Lin *et al.* [52] uses density vectors to predict the usefulness of prefetches. Gamoudi *et al.* [31] collects runtime events during sampling intervals which are used to turn on/off prefetching on subsequent non-sampling intervals. Panda *et al.* [66] propose a filter based on the weighted majority algorithm that uses four table-based predictors [95], [96]; the prediction is taken by comparing the confidence between positive and negative predictors. These works discard page-cross prefetches and optimize for in-page prefetches. Section V-A converts the state-of-the-art prefetch filter for in-page prefetches, PPF [20], to predict the usefulness of page-cross prefetches and shows that PPF falls short at forming an effective page-cross filter.

Page Size Aware Prefetching. Prior work [89] reveals how to exploit 2MB large pages to enable safe prefetching across 4KB physical page boundaries. This work does not optimize page-cross prefetching but exposes the page size information to the underlying prefetcher. Therefore, this work (i) does not provide benefits if large pages are not used and (ii) blindly permits the prefetcher to cross 4KB boundaries (assuming the existence of large pages) without filtering useless prefetch requests. Contrarily, our proposal provides consistent benefits with and without 2MB large pages since it accurately filters useless page-cross prefetches and is orthogonal to page-cross prefetching in the physical address space. Finally, Section V-B6 shows that DRIPPER significantly outperforms the proposal of [89] when 2MB large pages are used.

Prefetch Management. Prior art [43], [73], [74], [83], [91] lowers the prefetch-induced cache pollution by making cache policies aware of prefetching. Another line of work [25]–[28], [33], [39], [62], [64], [65], [69], [82], [83] uses heuristics to tune prefetching aggressiveness. Prior work [53], [54], [63] proposes predictors that permit prefetching only for critical loads in many-core systems with constrained bandwidth.

VII. Conclusions

This work demonstrates the importance of optimizing page-cross prefetching for first-level caches. In response, it proposes MOKA, a framework for designing μarchitectural filters that ensure effective and accurate page-cross prefetching. We use the MOKA framework to prototype a page-cross filter, named DRIPPER, and we show that DRIPPER significantly improves the performance of various state-of-the-art L1D prefetchers across an extensive set of workloads.

REFERENCES

[1] "AMD EPYC™ 7003 Processors," https://www.amd.com/en/server-docs/software-optimization-guide-for-amd-epyc-7003-processors-zip-format.

[2] "ARM Cortex-A55 Core Technical Reference Manual r1p0," https://developer.arm.com/documentation/100442/0100/functional-description/level-1-memory-system/data-prefetching?lang=en.

[3] "ARM Neoverse V2," https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-v2#:~:text=The%20Arm%20Neoverse%20V2%20CPU,%2C%20power%2C%20and%20security%20enhancements., accessed: 05-9-2022.

[4] "Championship Value Prediction (CVP)," https://www.microarch.org/cvp1/, accessed: 05-9-2022.

[5] "ChampSim," https://crc2.ece.tamu.edu/, accessed: 05-9-2022.

[6] "Coffee Lake - Microarchitectures - Intel," https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake, accessed: 05-9-2022.

[7] "Geekbench 5," https://www.geekbench.com/blog/2019/09/geekbench-5/.

[8] "Hot Chips 2023: Arm's Neoverse V2," https://chipsandcheese.com/2023/09/11/hot-chips-2023-arms-neoverse-v2/, accessed: 05-9-2022.

[9] "SPEC CPU 2006," https://www.spec.org/cpu2006/, accessed: 05-9-2022.

[10] "SPEC CPU 2017," https://www.spec.org/cpu2017/, accessed: 05-9-2022.

[11] "PARSEC," https://parsec.cs.princeton.edu/, 6666, accessed: 05-9-2022.

[12] Abishek Bhattacharjee, "Advanced Concepts on Address Translation, Appendix L in "Computer Architecture: A Quantitative Approach" by Hennessy and Patterson," http://www.cs.yale.edu/homes/abhishek/abhishek-appendix-l.pdf.

[13] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 513–526. [Online]. Available: https://doi.org/10.1145/3373376.3378498

[14] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture*, ser. HPCA '19, Feb 2019, pp. 399–411.

[15] A. Basu, M. D. Hill, and M. M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 297–308.

[16] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[17] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2022, pp. 1–18.

[18] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning," in *Proceedings of the 54th International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1121–1137. [Online]. Available: https://doi.org/10.1145/3466752.3480114

[19] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual Spatial Pattern Prefetcher," in *Proceedings of the 52nd International Symposium on Microarchitecture*, ser. MICRO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–544. [Online]. Available: https://doi.org/10.1145/3352460.3358325

[20] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-Based Prefetch Filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: https://doi.org/10.1145/3307650.3322207

[21] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th International Symposium on Microarchitecture*, ser. MICRO '13. New York, NY, USA: ACM, 2013, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540741

[22] P. Braun and H. Litz, "Understanding memory access patterns for prefetching," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:210184295

[23] Y. Chen, L. Pei, and T. E. Carlson, "Leaking Control Flow Information via the Hardware Prefetcher," *CoRR*, vol. abs/2109.00474, 2021.

[24] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[25] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared-resource management for multi-core systems," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 141–152.

[26] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, apr 2012. [Online]. Available: https://doi.org/10.1145/2166879.2166881

[27] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 316–326.

[28] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, ser. HPCA '09, 2009, pp. 7–17.

[29] M. Evers, L. Barnes, and M. Clark, "The amd next-generation "zen 3" core," *IEEE Micro*, vol. 42, no. 3, pp. 7–12, 2022.

[30] J. Feliu, A. Perais, D. Jimenez, and A. Ros, "Rebasing microarchitectural research with industry traces," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*. Ghent, Belgium: IEEE Computer Society, Oct. 2023, pp. 100–114. [Online]. Available: http://webs.um.es/aros/papers/pdfs/jfeliu-iiswc23.pdf

[31] O. Gamoudi, N. Drach-Temam, and K. Heydemann, "Using runtime activity to dynamically filter out inefficient data prefetches," in *European Conference on Parallel Processing*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:29391346

[32] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, "Bit-level perceptron prediction for indirect branches," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 27–38.

[33] A. Gendler, A. Mendelson, and Y. Birk, "A pab-based multi-prefetcher mechanism," *Int. J. Parallel Program.*, vol. 34, no. 2, p. 171–188, apr 2006. [Online]. Available: https://doi.org/10.1007/s10766-006-0006-1

[34] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," 2022. [Online]. Available: https://arxiv.org/abs/2210.14324

[35] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the Samsung Exynos CPU Microarchitecture," in *Proceedings of the 47th International Symposium on Computer Architecture*, ser. ISCA '20, 2020, pp. 40–51.

[36] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 368–379. [Online]. Available: https://doi.org/10.1145/2976749.2978356

[37] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[38] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning Memory Access Patterns," 2018. [Online]. Available: https://arxiv.org/abs/1803.02329

[39] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243181

[40] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[41] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 172–182.

[42] A. Jain and C. Lin, "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching," in *Proceedings of the 46th International Symposium on Microarchitecture*, ser. MICRO '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 247–259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[43] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 110–123.

[44] D. Jimenez, "Fast path-based neural branch prediction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 243–252.

[45] D. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01, 2001, pp. 197–206.

[46] D. A. Jiménez, "Piecewise linear branch prediction," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 382–393.

[47] D. A. Jiménez and E. Teran, "Multiperspective Reuse Prediction," in *Proceedings of the 50th International Symposium on Microarchitecture*, ser. MICRO '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 436–448. [Online]. Available: https://doi.org/10.1145/3123939.3123942

[48] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proceedings of the 49th International Symposium on Microarchitecture*, ser. MICRO '16, 2016, pp. 1–12.

[49] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints," in *Proceedings of the 25th International Symposium on Computer Architecture*, ser. ISCA '98, 1998, pp. 357–368.

[50] W. Lee, J. Lee, B. K. Park, and R. Y. C. Kim, "Microarchitectural characterization on a mobile workload," *Applied Sciences*, vol. 11, no. 3, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/3/1225

[51] Levinthal D., "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Intel Performance Analysis Guide," https://www.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf.

[52] W.-F. Lin, S. Reinhardt, D. Burger, and T. Puzak, "Filtering superfluous prefetches using density vectors," in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, 2001, pp. 124–132.

[53] H. Litz, G. Ayers, and P. Ranganathan, "Crisp: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[54] R. Manikantan and R. Govindarajan, "Focused prefetching: performance oriented prefetching based on commit stalls," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 339–348. [Online]. Available: https://doi.org/10.1145/1375527.1375576

[55] Matthias Waldhauer, "New AMD Zen core details emerged," http://dresdenboy.blogspot.com/2016/02/new-amd-zen-core-details-emerged.html, accessed: 05-9-2022.

[56] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 162. [Online]. Available: https://doi.org/10.1145/977091.977115

[57] P. Michaud, "Best-offset Hardware Prefetching," in *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, ser. HPCA '16, 2016, pp. 469–480.

[58] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez, "CHiRP: Control-Flow History Reuse Prediction," in *Proceedings of the 2020 53rd International Symposium on Microarchitecture*, ser. MICRO '16, 2020, pp. 131–145.

[59] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1124–1137.

[60] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *Proceedings of the 55th International Symposium on Microarchitecture*, ser. MICRO '22, 2022, pp. 975–991.

[61] S. Pakalapati and B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching," in *Proceedings of the 2020 47th International Symposium on Computer Architecture*, ser. ISCA '20, 2020, pp. 118–131.

[62] B. Panda, "Spac: A synergistic prefetcher aggressiveness controller for multi-core systems," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3740–3753, 2016.

[63] B. Panda, "Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 714–727. [Online]. Available: https://doi.org/10.1145/3613424.3614245

[64] B. Panda and S. Balachandran, "Introducing thread criticality awareness in prefetcher aggressiveness control," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–6.

[65] B. Panda and S. Balachandran, "Caffeine: A utility-driven prefetcher aggressiveness engine for multicores," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, aug 2015. [Online]. Available: https://doi.org/10.1145/2806891

[66] B. Panda and S. Balachandran, "Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 13–16, 2016.

[67] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[68] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/885651.781076

[69] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, ser. HPCA '14, 2014, pp. 626–637.

[70] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. USA: IEEE Computer Society, 1999, p. 234–245. [Online]. Available: https://doi.org/10.1145/300979.300999

[71] A. Ros, "Bl∪e: A timely, ip-based data prefetcher," in *The 1st ML-Based Data Prefetching Competition. ML for Computer Architecture and Systems*, Worldwide event, Jun. 2021.

[72] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[73] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 355–366.

[74] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, jan 2015. [Online]. Available: https://doi.org/10.1145/2677956

[75] A. Seznec, "The FNL+MMA Instruction Cache Prefetcher," https://hal.inria.fr/hal-02884880/document.

[76] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO '15, Dec 2015, pp. 141–152.

[77] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21.

New York, NY, USA: Association for Computing Machinery, 2021, p. 861–873. [Online]. Available: https://doi.org/10.1145/3445814.3446752

[78] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[79] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner, "POWER5 System Microarchitecture," *IBM Journal of Research and Development*, vol. 49, pp. 505 – 521, 08 2005.

[80] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[81] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," in *Proceedings of the 33rd International Symposium on Computer Architecture*, ser. ISCA '06, 2006, pp. 252–263.

[82] A. Sridharan, B. Panda, and A. Seznec, "Band-pass prefetching: An effective prefetch management mechanism using prefetch-fraction metric in multi-core systems," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017. [Online]. Available: https://doi.org/10.1145/3090635

[83] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07, 2007, pp. 63–74.

[84] D. Suggs, M. Subramony, and D. Bouvier, "The AMD "Zen 2" Processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[85] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, p. 280–300, sep 2005. [Online]. Available: https://doi.org/10.1145/1089008.1089011

[86] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[87] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas, "Morrigan: A Composite Instruction TLB Prefetcher," in *Proceedings of the 54th International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1138–1153. [Online]. Available: https://doi.org/10.1145/3466752.3480049

[88] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, "Exploiting Page Table Locality for Agile TLB Prefetching," in *Proceedings of the 48th International Symposium on Computer Architecture*, ser. ISCA '21, 2021, pp. 85–98.

[89] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, "Page Size Aware Cache Prefetching," in *Proceedings of the 55th International Symposium on Microarchitecture*, ser. MICRO '22, 2022, pp. 956–974.

[90] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest," in *Proceedings of the 2022 Symposium on Security and Privacy*, ser. SP '22. IEEE Computer Society, 2022.

[91] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 442–453.

[92] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal Prefetching Without the Off-Chip Metadata," in *Proceedings of the 52nd International Symposium on Microarchitecture*, ser. MICRO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 996–1008. [Online]. Available: https://doi.org/10.1145/3352460.3358300

[93] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient Metadata Management for Irregular Data Prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019, pp. 1–13.

[94] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, p. 20–24, mar 1995. [Online]. Available: https://doi.org/10.1145/216585.216588

[95] X. Zhuang and H.-H. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *2003 International Conference on Parallel Processing, 2003. Proceedings.*, 2003, pp. 286–293.

[96] X. Zhuang and H.-h. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 18–31, 2007.