# Page Size Aware Cache Prefetching

Georgios Vavouliotis[§†]
*georgios.vavouliotis@bsc.es*

Gino Chacon[‡]
*ginochacon@tamu.edu*

Lluc Alvarez[§†]
*lluc.alvarez@bsc.es*

Paul V. Gratz[‡]
*pgratz@tamu.edu*

Daniel A. Jiménez[‡]
*djimenez@acm.org*

Marc Casas[§†]
*marc.casas@bsc.es*

[§]Barcelona Supercomputing Center [†]Universitat Politècnica de Catalunya [‡]Texas A&M University

***Abstract*—The increase in working set sizes of contemporary applications outpaces the growth in cache sizes, resulting in frequent main memory accesses that deteriorate system performance due to the disparity between processor and memory speeds. Prefetching data blocks into the cache hierarchy ahead of demand accesses has proven successful at attenuating this bottleneck. However, spatial cache prefetchers operating in the physical address space leave significant performance on the table by limiting their pattern detection within 4KB physical page boundaries when modern systems use page sizes larger than 4KB to mitigate the address translation overheads.**

**This paper exploits the high usage of *large pages* in modern systems to increase the effectiveness of spatial cache prefetching. We design and propose the *Page-size Propagation Module (PPM)*, a $\mu$architectural scheme that propagates the page size information to the lower-level cache prefetchers, enabling *safe* prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages, at the cost of augmenting the first-level caches' Miss Status Holding Register (MSHR) entries with one additional bit. PPM is compatible with any cache prefetcher without implying design modifications. We capitalize on PPM's benefits by designing a module that consists of two page size aware prefetchers that inherently use different page sizes to drive prefetching. The composite module uses adaptive logic to dynamically enable the most appropriate page size aware prefetcher. Finally, we show that the proposed designs are transparent to which cache prefetcher is used.**

**We apply the proposed page size exploitation techniques to four state-of-the-art spatial cache prefetchers. Our evaluation shows that our proposals improve single-core geomean performance by up to 8.1% (2.1% at minimum) over the original implementation of the considered prefetchers, across 80 memory-intensive workloads. In multi-core contexts, we report geomean speedups up to 7.7% across different cache prefetchers and core configurations.**

***Keywords*-cache hierarchy, prefetching, spatial correlation, microarchitecture, hardware, virtual memory, address translation, large pages, memory management, memory wall**

## I. Introduction

System performance continues to be limited by the Memory Wall [1], [2], *i.e.*, the discrepancy between high memory access latencies and high processor speeds. Low-latency caches can attenuate this bottleneck by exploiting applications' locality to reduce the latency cost of demand memory accesses but are limited in capacity due to the overhead of implementing large SRAMs near cores.

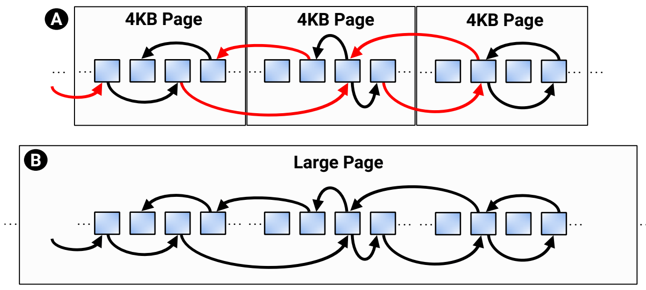


Figure 1. Data structure residing on multiple 4KB pages (up) and one large page (down). Arrows (black and red) illustrate the memory access patterns. Red arrows represent patterns across 4KB pages that a cache prefetcher operating on the physical address space is not allowed to prefetch for (even if it correctly identifies the patterns). There are no red arrows when the data structure resides in a large page.

Prefetching is a technique that hides the latency of memory accesses by proactively fetching data blocks into the cache hierarchy before a core explicitly demands them—alleviating the pressure placed on the memory subsystem by applications with large working sets that the caches cannot fully contain [3]. Effective prefetching has proven successful in attenuating the Memory Wall bottleneck; this is the reason why modern high-performance computing chips employ various cache prefetchers [4]–[13].

Numerous cache prefetchers have been proposed in recent literature [14]–[27]. These prefetchers generally fall into two categories; spatial prefetchers and temporal prefetchers. Spatial prefetchers [14]–[21] exploit the similarity of access patterns across different memory regions to drive prefetching decisions. In contrast, temporal prefetchers [22]–[26] do so by recording sequences of past cache misses. Although effective, temporal prefetchers have drawbacks compared to spatial prefetchers: (i) spatial prefetchers require orders of magnitude less metadata storage compared to temporal prefetchers [21], (ii) spatial prefetchers can save compulsory misses [28] whereas temporal prefetchers are fundamentally limited to prefetch for compulsory misses, and (iii) spatial prefetchers not only save long-latency cache misses but also improve the overall system energy consumption since they increase the DRAM row buffer hit ratio [20], [21], [29].

Previously proposed spatial cache prefetchers operating in the physical address space preserve one key property: they do not permit prefetching beyond 4KB physical page boundaries as physical address contiguity is not guaranteed, *i.e.*, addresses that are contiguous in the virtual address space may be very distant in the physical address space. In addition, crossing 4KB physical page boundaries for prefetching is susceptible to security issues since an adversary could exploit it to create a side-channel [30]–[32]. Prefetchers are unaware of the access permissions of specific pages, thus page-crossing prefetching might allow loading data from pages the prefetcher's cache would not otherwise have access to. Indeed, a recent reverse engineering study [30] demonstrates how to exploit page-crossing prefetching at the lower-level caches to perform a side-channel attack.

Limiting spatial prefetchers to prefetch for intra-4KB physical page patterns limits their ability to speculate long streams of memory accesses [30]. Enabling *safe* prefetching beyond 4KB physical pages requires direct access to the *Translation Lookaside Buffer (TLB)* and a reverse address translation [33]. These requirements pose high latency and energy overheads, hindering safe prefetching across 4KB physical page boundaries in real-world implementations.

The increase in working sets sizes of memory-intensive applications also places tremendous pressure on the TLB hierarchy [34]–[47]. This pressure results in frequent page walks that deteriorate application performance, even in the presence of dedicated software and hardware mechanisms for address translation [35], [48]–[59]. The requirement for small and fast TLBs with low miss rates has led to the advent of large page size support in many operating systems [60], [61], architectures [62]–[65], and virtualization enterprises [66]. For example, x86 architectures support 2MB and 1GB pages alongside standard 4KB pages to increase TLB reach.

This paper demonstrates that exploiting modern prevalence and support for *large pages* can significantly improve a system's overall performance by enabling safe prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages. Figure 1 illustrates this opportunity by considering a generic data structure and showing its memory access patterns when mapped into multiple 4KB pages Ⓐ and a single large page Ⓑ. Although the data structure has predictable patterns across 4KB boundaries, the prefetcher would not prefetch these patterns (red arrows in Figure 1 Ⓐ) due to the limitation of prefetching for intra-4KB patterns. In contrast, when the data structure is mapped to a large page (Figure 1 Ⓑ), the prefetcher *could* safely cross 4KB boundaries and speculate on future access patterns if it was aware that the block resides in a large page. However, the page size information is not available to cache prefetchers operating in the physical address space.

We perform two sets of experiments to highlight the potential of leveraging the presence of large pages for improving spatial cache prefetching effectiveness. First, we demonstrate that modern systems effectively use large pages by executing a set of memory-intensive workloads spanning various contemporary benchmark suites [67]–[69] on a real system [70] and observing that the majority of the considered workloads heavily use large pages throughout their entire execution. Second, we quantify the missed opportunity for safely crossing 4KB physical page boundaries when the block resides in a large page (Figure 1 Ⓑ) by evaluating four state-of-the-art lower-level spatial cache prefetchers (SPP [14], VLDP [15], PPF [16], BOP [17]) using 80 memory-intensive applications from various suites [39], [67]–[69], [71], [72]. Figure 2 depicts the probability that a given prefetch will attempt to cross 4KB page boundaries when the block resides in a large page, but the prefetcher discards it because it is unaware that the block resides in a large page. For most workloads, 1 out of 10 prefetches are discarded due to the restriction of not crossing 4KB boundaries. At the extreme, some workloads see 1 out of 2 prefetches being discarded due to this limitation. Taking into account that cache prefetchers issue multi-million prefetches per executed workload, the probability shown in Figure 2 reveals that enhancing lower-level cache prefetchers with the page size information of the accessed data blocks has the potential to significantly improve cache prefetching performance due to the opportunity of safely crossing 4KB physical page boundaries when data blocks reside in large pages.

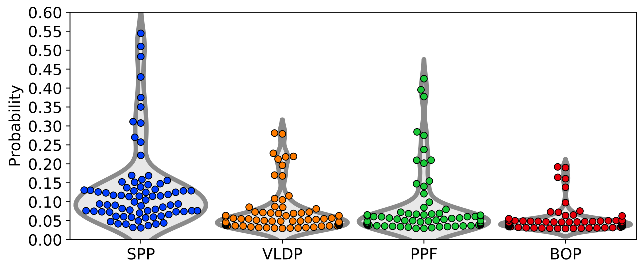


Figure 2. Probability distribution depicted with violin plots showing the probability for a given prefetch to be discarded because it attempts to cross 4KB physical page boundaries when the block resides in a large page, considering four spatial cache prefetchers and 80 applications.

Based on our findings, we propose the *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that propagates the page size information to the lower-level cache prefetchers, enabling safe prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page. PPM exploits the available address translation metadata after a first-level cache miss and directs the page size information to the lower-level cache prefetchers through the first-level caches' Miss Status Holding Registers (MSHRs). PPM operates without requiring any costly TLB lookup or reverse address translation. Moreover, we highlight that PPM does not imply any modification in the design of a lower-level cache prefetcher. For the rest of the paper, we refer to a prefetcher that exploits PPM as *Page Size Aware*

*Prefetcher (Pref-PSA)*.[1] Note that a Pref-PSA inherently uses 4KB pages to drive prefetching decisions since PPM enables prefetching beyond 4KB physical page boundaries (when possible) without modifying the prefetcher's design.

We capitalize on PPM's benefits by transparently integrating the notion of large pages into the design of any cache prefetcher.[1] We observe that doing so may positively or negatively impact performance as some workloads enjoy great benefits by making the prefetcher inherently use large pages while others experience performance degradation because large pages provide a coarser representation of the memory access patterns than standard 4KB pages. To avoid harming performance while enjoying the benefits of integrating large pages in the prefetcher's design, we implement a composite scheme that consists of two identical versions of the same Pref-PSA[1] that differ in only one aspect; one Pref-PSA inherently uses 4KB pages to drive prefetching while the other Pref-PSA uses large pages. Finally, the composite scheme uses adaptive logic based on Set-Dueling [73] to dynamically enable the most appropriate of the two competing prefetchers. We refer to this composite prefetcher as *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*.[1]

In summary, this paper makes the following contributions:

- This is the first study to reveal that leveraging modern prevalence and support for *large pages* can improve the effectiveness of spatial cache prefetchers operating in the physical address space due to the arising opportunity for crossing 4KB physical page boundaries when the accessed blocks reside in large pages.
- We propose *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that enables *safe* prefetching beyond 4KB physical page boundaries. Coupling prior spatial cache prefetchers SPP [14], VLDP [15], PPF [16], and BOP [17] with PPM provides single-core geomean speedups of 5.5%, 2.1%, 4.7%, and 2.1% over their original versions that stop prefetching at 4KB boundaries no matter the size of the page where the blocks reside, respectively, across 80 workloads. PPM does not imply modifications in the prefetcher's design.
- We capitalize on PPM by transparently integrating large pages into any prefetcher's implementation and designing a composite prefetcher, named *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*,[1] that selects between two identical Pref-PSAs that only differ in the page size that they use to drive prefetching. Our single-core evaluation shows that SPP-PSA-SD, VLDP-PSA-SD, PPF-PSA-SD, and BOP-PSA-SD outperform their original version by 8.1%, 4.0%, 5.1%, and 2.1%, respectively, across 80 workloads. In multi-core contexts, we report geomean speedups up to 7.7% across different prefetchers and core configurations.

[1]It can be any cache prefetcher operating in the physical address space.

## II. Background

### A. Data Cache Prefetching

Data cache prefetching is a technique that hides the latency cost of memory accesses by proactively fetching data blocks into the caches before a core explicitly requests them. Prior work in the domain can be generally classified into spatial cache prefetchers and temporal cache prefetchers. Spatial prefetchers [14], [15], [17]–[21] rely on the similarity of access patterns across different memory regions to drive prefetching, while temporal prefetchers [22]–[26] do so by recording the sequence of past cache misses, assuming that there will be a recurrence of those misses in the near future.

Spatial cache prefetchers are considered effective and are widely used in real-world implementations [5], [7], [70], [74] due to their intrinsic properties that provide unique benefits. Specifically, spatial prefetchers require orders of magnitude less metadata storage than temporal prefetchers [21] because the former store only deltas/offsets between accessed blocks within pages, whereas the latter records the complete sequence of the addresses that cause cache misses. In addition, spatial prefetchers can save compulsory cache misses, which constitute a critical bottleneck in applications [28], since they leverage observed deltas on already seen pages to prefetch for unobserved pages. Contrarily, temporal prefetchers record sequences of past misses, fundamentally limiting their ability to save compulsory misses. Finally, prior work demonstrates that spatial prefetches that reach DRAM typically enjoy row buffer hits as opposed to temporal prefetches, downplaying the fetch order impact while reducing the overall energy consumption of the system [20], [21], [29].

### B. Architectural Support for Address Translation

Each memory access on paging-based virtual memory systems requires an address translation from the virtual to the physical address space. Recent work [34]–[47] demonstrates that memory-intensive applications experience significant performance degradation (even up to 50%) due to tremendous pressure placed on the virtual memory subsystem. Modern systems provide a combination of both software and hardware mechanisms to reduce address translation overheads [35], [48]–[59]. The *page table* is an OS-managed and architecturally visible structure that contains the virtual-to-physical mappings of all pages loaded to main memory. In x86-64 architectures, the page table is implemented as a multi-level radix tree structure [55], [75]. The *Translation Lookaside Buffer (TLB)* is a small and fast buffer that stores the most recently used address translations entries and is typically implemented as a multi-level structure [33]. Finally, the MMU Caches [48] (referred as Page Structure Caches in x86 architectures) are $\mu$architectural structures that reduce some of the page walk references to the memory hierarchy (L1D$\rightarrow$L2C$\rightarrow$LLC$\rightarrow$DRAM) by caching intermediate levels of the radix tree page table [35], [56], [57].

*1) Large Pages:* Even in the presence of dedicated schemes for the virtual memory subsystem (Section II-B), address translation is still a major performance obstacle for contemporary applications [43], [76]. To alleviate this bottleneck, modern systems have introduced *large pages*, *i.e.*, pages larger than a standard 4KB page. For instance, x86-64 processors support 2MB and 1GB pages alongside standard 4KB pages. Effectively using large pages provides unique performance and energy gains. A large TLB entry accommodates the translation of a much larger contiguous memory region (*e.g.*, a 2MB page covers 512 times more memory space than a single 4KB page), increasing TLB reach. Large pages also reduce the number of page table levels that must be traversed upon a last-level TLB miss (4 traversals with 4KB pages, 3 traversals with 2MB pages).

Modern OSes provide two mechanisms to allocate large pages. The first approach requires the user to reserve physical memory for large pages and use the *hugetlbfs* library [77] to map specific memory segments of the application onto large pages. This approach is static and limits the usage of large pages. In the second approach, the OS transparently allocates large pages without requiring any user involvement. Specifically, the Linux Transparent Huge Pages (THP) mechanism [60] provides automatic and transparent support for 2MB pages. However, this is not the case for pages larger than 2MB (*e.g.*, 1GB pages for x86 architectures), which still require manual allocation using the *hugetlbfs* library.[2]

## C. Spatial Cache Prefetching and Page Boundaries

*1) Spatial Prefetching at L1D:* Modern L1D prefetchers use virtual addresses to drive prefetching and exploit the consistent streams exposed by the accessibility of the virtual addresses in the form of streaming and stride prefetchers [55], [78]–[82], identifying key access behaviors [25], [83], and correlated instruction-pointer prefetchers [55], [84]. Conceptually, these prefetchers can cross 4KB page boundaries [85] because they have direct access to the address translation module to extract the virtual-to-physical mappings of the pages where the prefetched blocks reside. In practice, it is not that straightforward. 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 fetch the corresponding translation from memory? Doing so would impact (positively or negatively) performance, bandwidth, and energy consumption depending on the accuracy of the page-crossing prefetching. Notably, crossing page boundaries greatly impacts the timeliness of the L1D prefetching [86]; even if page-crossing prefetching is accurate, it might negatively impact prefetching timeliness (and the system's overall performance) when it goes all the way down to DRAM to find the address translations since L1D prefetchers [83] require quick turnaround times on memory accesses due to the sheer amount of requests seen at the first-level caches. For these reasons, state-of-the-art L1D prefetchers [83] typically permit prefetching within standard 4KB page boundaries.[3]

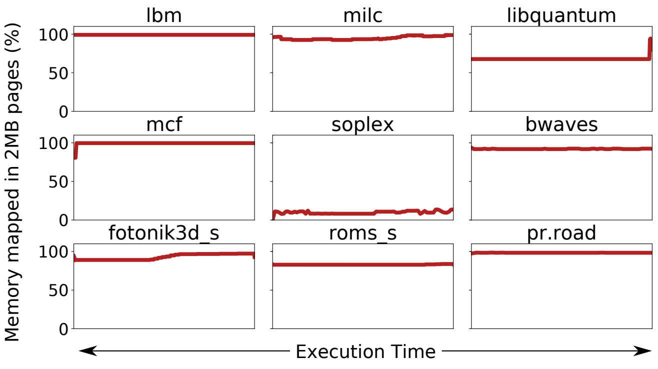


Figure 3. Percentage of allocated memory mapped to 2MB pages across the entire execution of nine representative memory-intensive benchmarks.

*2) Spatial Prefetching at L2C/LLC:* Prefetchers for the lower-level caches drive prefetching using physical addresses because the virtual addresses are not propagated to the lower-level caches since these caches are implemented as physically indexed physically tagged (PIPT) structures. In addition, lower-level cache prefetchers stop prefetching at 4KB physical boundaries since physical address contiguity is not guaranteed, *i.e.*, contiguous virtual addresses might not be contiguous in the physical address space, thus permitting prefetching beyond 4KB physical boundaries might introduce new security vulnerabilities. Allowing L2C/LLC prefetchers to speculatively cross 4KB physical page boundaries may result in prefetching data from pages that a given process does not have access to, thereby preloading data into the cache hierarchy. This behavior results in a side-channel in which an adversary can detect if the victim has accessed a page despite the adversary not having permissions to that page [30]–[32]. Indeed, a recent reverse engineering study [30] shows how to perform a side-channel attack on recent Apple processors (*e.g.*, M1 Max and M1 Pro) by exploiting page-crossing prefetching at the lower-level caches.

## III. Motivation

This section reveals that exploiting the presence of large pages in modern systems can improve cache prefetching effectiveness by enabling *safe* prefetching across 4KB page boundaries. Our analysis focuses on x86 architectures with 4KB and 2MB pages since modern OSes provide automatic and transparent support for only these page sizes (Section II-B1). In addition, we target cache prefetchers operating in the physical address space (L2C/LLC prefetchers) and not

[2]There is a recent work [44] that aims at automatically and transparently allocating all page sizes in x86 systems (including 1GB pages).

[3]Designing a synergistic TLB prefetcher that improves the timeliness of L1D page-crossing prefetching is a promising research direction.

cache prefetchers driving prefetching with virtual addresses (L1D prefetchers) for the reasons explained in Section II-C1 and for two other reasons. First, L1D prefetchers issue prefetch requests using virtual addresses on every L1D access. However, the page size information is part of the address translation metadata available after the TLB access, thus waiting for the page size information upon TLB misses might harm the timeliness of L1D prefetching. Second, first-level caches necessitate simple and fast prefetchers due to their sizes and access latencies as opposed to lower-level caches that permit the implementation of sophisticated prefetchers. Finally, we focus on spatial prefetchers for the reasons outlined in Section II-A.

## *A. Limitations of Existing Cache Prefetchers*

Previously proposed lower-level spatial cache prefetchers preserve one key property; they assume the use of only standard 4KB pages, limiting their pattern detection to 4KB memory regions. Consequently, they do not permit prefetching beyond 4KB physical page boundaries because physical address contiguity is not guaranteed, *i.e.*, addresses that are contiguous in the virtual address space might not be contiguous in the physical address space. Moreover, prior spatial cache prefetchers stop prefetching at 4KB physical page boundaries because crossing 4KB boundaries might introduce new security vulnerabilities since an adversary could exploit page-crossing prefetching to attack the system, as explained in Section II-C.

Enabling *safe* spatial prefetching beyond 4KB physical page boundaries would ideally require direct access to the TLB to extract the virtual-to-physical mappings of the pages where the prefetched blocks reside. Doing so for spatial prefetchers operating in the physical address space requires allowing direct access from the lower-level caches to TLB to perform a reverse translation from the physical to the virtual address space. This reverse translation incurs high overheads since the reverse mappings are multi-valued functions [33].

***Finding 1.*** *There is no previously proposed μarchitectural scheme that ensures safe spatial prefetching beyond 4KB physical page boundaries for the lower-level caches.*

## *B. Opportunity for Safe Prefetching Across 4KB Boundaries*

As explained in Section II-B1, systems provide support for large page sizes to reduce the address translation overheads. When large pages are used, the corresponding physical mappings (physical pages) are also large, *i.e.*, the virtual and corresponding physical pages are of the same size. Intuitively, limiting the cache prefetcher to a 4KB physical page boundary when the accessed block resides on a large page leads to sub-optimal performance gains due to the missed opportunity for safely prefetching across 4KB boundaries.

The first question we answer is whether modern systems practically use large pages or not. To do so, we execute a set

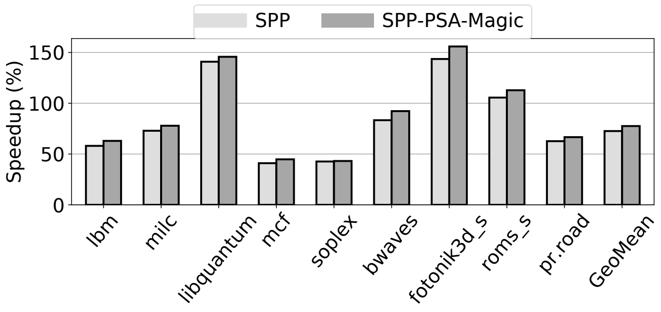


Figure 4. Performance comparison between the original implementation of SPP and the ideal page size aware SPP (SPP-PSA-Magic) across a set of memory-intensive workloads.

of memory-intensive benchmarks from various suites (SPEC 2006 [67], SPEC 2017 [68], and GAP [69]) on an Intel Xeon E5-2687W, collecting the usage of 4KB and 2MB pages with the page-collect tool [87] and Linux's THP mechanism [60] enabled (Section II-B1). Figure 3 presents the percentage of allocated memory mapped in 2MB pages across the execution of nine memory-intensive workloads. The main takeaway of this experiment is that most workloads heavily use 2MB pages, corroborating the conclusions of prior work [42], [44], [45]. Interestingly, most workloads preserve the high usage of 2MB pages during their entire execution.

***Finding 2.*** *Modern systems heavily use 2MB pages when executing memory-intensive applications.*

*1) Quantifying the Potential:* Qualitatively, making lower-level cache prefetchers aware of the size of the page where the accessed blocks reside would enable safe prefetching beyond 4KB physical page boundaries when the corresponding page is 2MB, resulting in better prefetching timeliness and coverage. Removing the restriction of prefetching within 4KB physical boundaries would allow the prefetchers to detect more distinct patterns, increasing their coverage. Furthermore, the prefetchers would be able to timely prefetch patterns that cross 4KB physical page boundaries instead of waiting for an access to the next page to start issuing prefetches for the already captured patterns.

*Underlying Prefetcher.* To quantitatively answer whether or not large page exploitation can improve the effectiveness of spatial prefetching for the lower-level caches, we consider the *Signature Path Prefetcher (SPP)* [14], a confidence-based look-ahead L2C prefetcher that directs prefetched blocks into L2C or LLC depending on its internal confidence mechanism. SPP creates a compressed signature and associates it with the physical page address. To do so, SPP relies on two main structures: (i) the Signature Table, a table indexed with the physical page number that stores the history of previously delta patterns per page as a compressed signature, and (ii) the Pattern Table, a table indexed by the signatures generated by the Signature Table that stores predicted deltas.

We focus on SPP to demonstrate the potential benefits of enabling beyond 4KB physical page boundaries spatial prefetching by leveraging the presence of 2MB pages in modern systems since SPP provides the basis for many L2C prefetcher designs and optimizations [16], [18], and has been deployed in real-world industrial designs [7]. In subsequent sections, we consider additional L2C prefetchers to highlight our proposals' versatility.

*Methodology.* To quantify the benefits that large pages could bring in spatial cache prefetching, we use an enhanced version of ChampSim [88] that supports both 4KB and 2MB pages. Section V describes our simulation infrastructure.

We implement and evaluate two different versions of SPP to demonstrate the benefits of exploiting the presence of 2MB pages for enhancing lower-level cache prefetching performance. The first version corresponds to the original implementation of SPP, which stops speculation at 4KB physical page boundaries, no matter the page size of the accessed block since it does not have any notion of the page size. The second version of SPP differs from the original in that SPP *magically* knows the page size of the accessed blocks, so it stops prefetching at 4KB boundaries when the block resides in a 4KB page or 2MB boundaries when the block resides in a 2MB page. Figure 4 presents the performance of the original SPP and its ideal page size aware version (SPP-PSA-Magic) over a baseline without prefetching at any cache level, similar to prior work [14], [16], [18], across the same set of memory-intensive benchmarks used for the real system measurements, presented in Section III-B. Our evaluation (Section VI) considers additional workloads (Section V) to highlight the benefits of our proposals.

Figure 4 reveals that magically propagating the page size information to SPP significantly improves its performance since SPP-PSA-Magic outperforms SPP for all considered workloads (5.2% in geomean). The only exception is `soplex` where SPP and SPP-PSA-Magic provide similar speedups since this workload mainly uses 4KB pages, limiting the opportunity for further performance gains by exploiting the presence of 2MB pages. SPP-PSA-Magic outperforms SPP original because it experiences better timeliness and coverage than SPP. Specifically, SPP-PSA-Magic issues prefetches that SPP would otherwise discard due to the limitation of prefetching for intra-4KB page patterns or postponing until there is an access to the prefetched block's page. Finally, we emphasize that SPP-PSA-Magic does not imply any modification to SPP's original design since it keeps driving prefetching using 4KB indexes, similar to SPP original, despite its awareness of the page size.

***Finding 3.*** *Making cache prefetchers operating in the physical address space aware of the page size has potential for significantly improving system's performance without any modification in the prefetcher's design.*

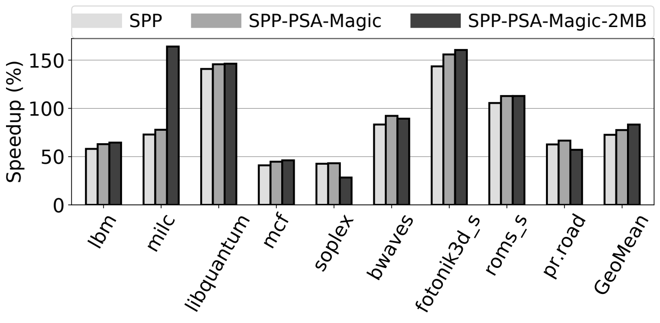


Figure 5. Performance of SPP original, SPP-PSA-Magic from Figure 4, and the ideal page size aware SPP that inherently uses 2MB pages (SPP-PSA-Magic-2MB), across a set of memory-intensive workloads.

## C. Integrating Large Pages into the Design

Section III-B1 reveals that *magically* propagating the page size information to SPP provides large speedups without implying any modification to its original design. Apart from this, what would be the performance impact of integrating 2MB pages into the design of SPP? This section answers this question, using the same baseline as Section III-B1.

The original version of SPP uses multiple structures to drive prefetching (Section III-B1). Only one of these structures, named Signature Table, uses the physical page number for indexing. Therefore, we implement another version of SPP that differs from the original version in only one aspect; it uses 2MB pages, not 4KB pages, to index the Signature Table. Consequently, the new SPP version can store deltas into the structure that stores predicted deltas, named Pattern Table, that are larger than the ones stored in the corresponding structure of SPP original and SPP-PSA-Magic.[4] In addition, the signature stored in the Signature Table depends on the deltas stored in the Pattern Table. Consequently, this new version of SPP has fundamental differences compared to SPP original and SPP-PSA-Magic: (i) it eliminates aliasing in the Pattern Table due to indexing with 2MB pages at the cost of generalizing patterns among all 4KB pages within a 2MB memory block, and (ii) it can discover patterns that SPP original and SPP-PSA-Magic fail at finding due to considering larger deltas for prefetching and/or experiencing less aliasing in the Pattern Table. Note that the new SPP version is magically aware of the page size to adjust its prefetching boundaries accordingly, similar to SPP-PSA-Magic of Section III-B1. We refer to this new SPP version as SPP-PSA-Magic-2MB. Figure 5 presents the speedups of SPP original, SPP-PSA-Magic from Section III-B1, and SPP-PSA-Magic-2MB.

Figure 5 reveals that SPP-PSA-Magic-2MB behaves differently across different benchmarks. For example, it provides huge speedups over both SPP and SPP-PSA-Magic for the `milc` benchmark. We observe such behavior because

[4]Deltas within a 4KB page range between -64 to +64 whereas deltas within a 2MB page range between -32768 to +32768.

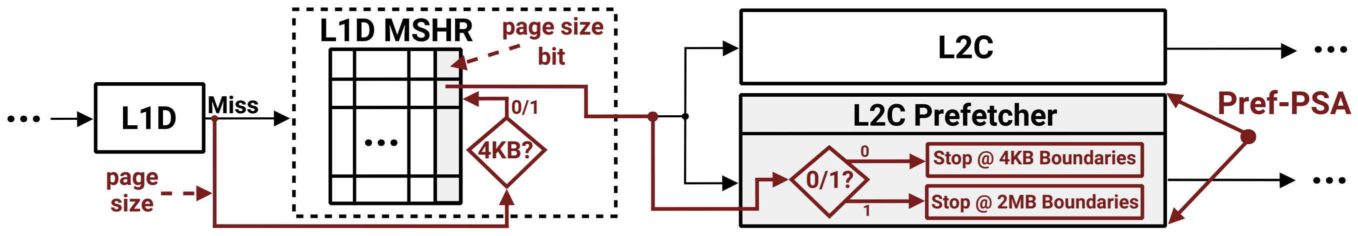


Figure 6. Page-size Propagation Module (PPM).

SPP-PSA-Magic-2MB does not suffer from aliasing in the Pattern Table and it prefetches for patterns that both SPP and SPP-PSA-Magic fail at capturing due to considering smaller deltas.[4] For benchmarks like `libquantum`, SPP-PSA-Magic-2MB performs similar to SPP-PSA-Magic (still greater than SPP original). However, there are benchmarks (*e.g.*, `soplex`) where SPP-PSA-Magic-2MB significantly degrades performance over SPP original. Such behavior is observed because indexing the Signature Table with 2MB pages changes its content and the patterns it captures. Interestingly, results presented in Figure 5 demonstrate that indexing with 4KB pages, regardless of whether the block resides in a 4KB or a 2MB page, is sometimes better than indexing with 2MB pages since SPP-PSA-Magic outperforms SPP-PSA-Magic-2MB for some workloads (*e.g.*, `soplex`, `pr.road`). This is the case for workloads that have fine-grain address patterns (4KB-grain). In other words, when a block resides in a 2MB page it is seldom beneficial to index the prefetcher's structures with 2MB pages; sometimes indexing with 4KB pages, no matter the size of the page where the accessed block resides, provides better prefetches.

***Finding 4.*** *Integrating 2MB pages into the design of a cache prefetcher may positively or negatively impact performance, depending on the workload. A scheme that dynamically selects between two page size aware versions of a prefetcher that drive speculation considering different page sizes has the potential to deliver outstanding benefits.*

### D. Putting Everything Together

Sections III-B1 and III-C highlight that leveraging the presence of 2MB pages in modern systems for lower-level cache prefetching has the potential to provide significant benefits. However, the reported gains assume *magically* propagating the page size information to the lower-level cache prefetchers. Realistically leveraging 2MB pages for enhancing cache prefetching performance requires (i) a scheme that propagates the page size information to the lower-level cache prefetchers, and (ii) a smart mechanism that enables the page-size aware version of the prefetcher that inherently uses 2MB pages only when it is confident that doing so would positively impact performance.

## IV. Design

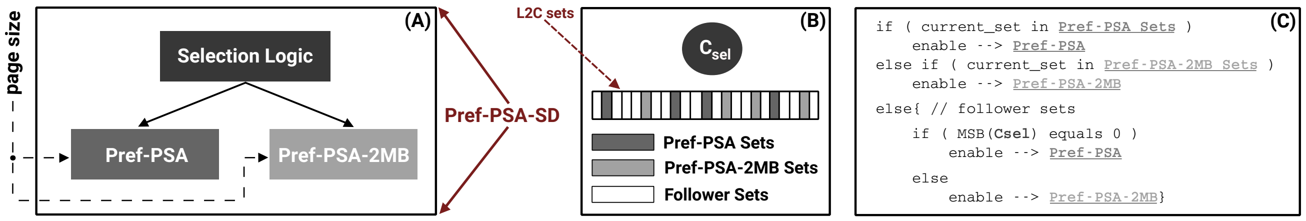
We design and propose the *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that propagates the page size information to lower-level cache prefetchers and enables *safe* prefetching beyond 4KB physical page boundaries when an accessed block resides in a large page (Section IV-A). We show that PPM is compatible with any lower-level cache prefetcher without implying any modification to the underlying prefetcher's implementation. For the rest of the paper, we use *Page Size Aware Prefetcher (Pref-PSA)* to refer to a prefetcher that exploits the PPM module.

We further capitalize on PPM's benefits through the design of a composite scheme that transparently integrates large pages into the prefetcher's design, providing additional performance benefits at modest storage and logic costs. Section IV-B presents in detail this composite scheme.

### A. Page-size Propagation Module (PPM)

To address our analysis findings (Section III), we design the *Page-size Propagation Module (PPM)*, an easily implemented $\mu$architectural scheme that makes lower-level cache prefetchers aware of the page size of the accessed blocks, enabling *safe* prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page (Findings 2 and 3, Section III-B). Practically, PPM augments the cache MSHRs with one additional bit indicating the page size of the corresponding accessed block. PPM does not imply any modification to the underlying prefetcher's implementation nor any costly reverse virtual to physical address translation (Finding 1, Section III-A).

We focus on prefetching applied at the L2C to describe the design and the operation of PPM while presenting the modifications required to propagate the page size information to LLC prefetchers. We target L2C prefetching because contemporary L2C prefetchers (i) store prefetched blocks into the L2C or LLC depending on their internal confidence mechanisms, and (ii) a prefetcher placed in the L2C has a clearer view of the miss stream than an LLC prefetcher. We do not target L1D prefetchers because (i) waiting for the page size information upon TLB misses might harm their timeliness since these prefetchers operate on L1D accesses, as explained in Section III, and (ii) the L1D opts for low-latency accesses, hindering the implementation of sophisticated prefetchers (Sections II-C1 and III.)



```
if ( current_set in Pref-PSA Sets )
    enable --> Pref-PSA
else if ( current_set in Pref-PSA-2MB Sets )
    enable --> Pref-PSA-2MB
else{ // follower sets
    if ( MSB(Csel) equals 0 )
        enable --> Pref-PSA
    else
        enable --> Pref-PSA-2MB}
```

Figure 7. (A) L2C prefetching module comprised of two generic page size aware (PSA) prefetchers and adaptive logic that dynamically selects between them, (B) selection logic implementation, (C) operation in pseudo-code. Pref-PSA (Pref-PSA-2MB) drives prefetching assuming 4KB (2MB) pages.

*1) Implementation and Operation:* The key idea behind PPM is that first-level caches are typically implemented as virtually indexed physically tagged (VIPT), thus upon an L1D miss the page size of the missed block is available as part of the address translation metadata.

In practice, on L1D misses PPM extracts the page size information from the address translation metadata and propagates it to the L1D MSHR. To do so, we augment each L1D MSHR entry with one additional bit indicating the page size of the missed block. Since L2C prefetchers are engaged on L2C accesses, *i.e.*, L1 misses, PPM propagates the page size bit from the L1D MSHR to the L2C prefetcher via the corresponding request's stream, making the L2C prefetcher aware of the page size (Pref-PSA).

Figure 6 illustrates the design and operation of a cache hierarchy enhanced with PPM. In practice, upon an L1D miss, PPM records the corresponding miss to the L1D MSHR coupled with one more bit annotating the page size of the corresponding block from the address translation metadata. The page size bit is either 0 or 1, indicating whether the corresponding missed block resides in a 4KB or 2MB page, respectively. Then, Pref-PSA takes the page size bit as input and adjusts its prefetching strategy accordingly. If the page size bit is 0, Pref-PSA stops prefetching at 4KB boundaries since the block resides in a 4KB page. However, if the page size bit is 1, Pref-PSA *safely* crosses 4KB boundaries since it is aware that the block resides in a 2MB page and stops prefetching at 2MB boundaries. Although Pref-PSA may continue prefetching across 4KB boundaries when the accessed blocks reside in 2MB pages, it continues to index its internal prefetching structures using 4KB pages, regardless of the page size, since PPM does not imply any modification in the underlying prefetcher's design.

*Storage Overhead.* PPM's implementation requires just one bit per L1D MSHR entry, assuming two concurrently supported page sizes (4KB pages and 2MB pages).

*Additional Page Sizes.* Although architectural support for address translation is similar between different architectures (x86, ARM, RISC-V), some implementations support more than two page sizes. PPM is compatible with any number of concurrently supported page sizes but would require more bits stored in the L1D MSHR entries. Assuming $N$ concurrently supported page sizes, PPM needs to additionally store $\lceil \log_2 N \rceil$ bits on each L1D MSHR entry.

*Operation on L1I Misses.* Today, Linux transparently supports 2MB pages only for data, not for code. In addition, mapping code into large pages using the *hugetlbfs* library [77] might introduce security vulnerabilities [86], [89], [90]. For these reasons, this work considers that all instruction pages are 4KB, and do not enhance the L1I MSHR with the page size bit. We emphasize that this is not a limitation of our design, but rather an implementation choice based on the policies followed by modern systems. PPM can also be used, without any modification, to propagate the page size information to the L2C prefetching module upon L1I misses.

*Applicability on LLC Prefetching.* The procedure for propagating the page size information to an LLC prefetcher is similar to the one explained in Section IV-A1, but with another propagation level. First, the L2C MSHR entries should also store a bit annotating the page size. Second, upon an L2C miss, the page size bit should be propagated from the L1D MSHR to the L2C MSHR. Finally, the L2C MSHR routes the page size bit to the LLC prefetcher.

## *B. Integrating Large Pages in the Design*

This section builds on top of the PPM (Section IV-A) and presumes that the page size information propagates to the L2C prefetching module. In other words, it assumes the placement of a generic Pref-PSA alongside L2C.

In this section, we show how to couple an existing Pref-PSA with another page size aware version of the same prefetcher that uses 2MB pages to drive prefetching; we refer to this prefetcher as *Pref-PSA-2MB*. As explained in Section III-C, integrating 2MB pages into the design of an L2C prefetcher may positively or negatively impact performance. To address this finding, we also implement a smart and low-cost scheme that enables Pref-PSA-2MB only when it is confident that doing so will positively impact performance.

Figure 7 (A) depicts a high-level overview of our composite design. The L2C prefetching module consists of (i) two generic page size aware prefetchers, one inherently using 4KB pages (Pref-PSA), similar to Section IV-A, and another inherently using 2MB pages (Pref-PSA-2MB), and (ii) adaptive selection logic, based on Set-Dueling [73], that dynamically selects between Pref-PSA and Pref-PSA-2MB. We refer to this composite design as *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*.

*1) Design of Pref-PSA-2MB:* We transparently integrate the notion of 2MB pages into any L2C prefetcher design by targeting its internal prefetching structures indexed with the physical page number (if any). The only modification necessary is that we require these structures to be indexed using 2MB pages, no matter the size of the page where the accessed block resides. Although Pref-PSA-2MB assumes 2MB pages for indexing its internal structures, prefetching is permitted within the page where the trigger block resides to avoid opening side-channels (Section II-C). If the prefetcher has no structure indexed with the physical page number, Pref-PSA-2MB is equivalent to Pref-PSA. Note that Pref-PSA-2MB uses predicted deltas that range between -32768 and +32768 since it assumes only 2MB pages. Therefore, Pref-PSA-2MB may, or may not, capture patterns that Pref-PSA captures, as explained in Section III-C.

*2) Selection Logic:* To address our last analysis finding and agilely select between Pref-PSA and Pref-PSA-2MB, we implement a scheme based on Set Dueling [73], a technique originally invented to select between different replacement policies within a cache. Our selection logic (Figure 7 (B)) consists of a single saturating counter, $C_{sel}$, that reflects which prefetcher to enable for the current access. We train both Pref-PSA and Pref-PSA-2MB on all L2C accesses since training each prefetcher only when it is selected results in insufficient training.

In practice, the selection logic clusters the L2C sets into three categories: sets dedicated to Pref-PSA, sets dedicated to Pref-PSA-2MB, and follower sets dynamically assigned to the most accurate prefetcher between Pref-PSA and Pref-PSA-2MB. We dedicate a small fraction of the total L2C sets to the two competing prefetchers to avoid negatively impacting performance when one of the prefetchers harms performance. Empirically, we find that 32 sets are adequate for each prefetcher, similar to prior work [73]. To make our Set Dueling based scheme work for prefetching, we use one bit per L2C block to annotate which prefetcher (Pref-PSA or Pref-PSA-2MB) issued the prefetch to ensure correct updating of $C_{sel}$. This bit is required because the prefetched block may not be stored in the same set as the trigger block.[5] The annotation bit implies 1KB extra storage for a 512KB L2C which is affordable for realistic implementations.

*3) Pref-PSA-SD Operation:* Pref-PSA-SD monitors the efficacy of each prefetcher by marking prefetched blocks based on the issuing prefetcher. Upon an L2C access, Pref-PSA or Pref-PSA-2MB issues prefetches for the current access based on whether or not the accessed block belongs to either prefetchers' sample set. If the corresponding block does not belong to either prefetcher's sample sets, $C_{sel}$ selects which prefetcher should be enabled. If the Most Significant Bit (MSB) of $C_{sel}$ is 0, Pref-PSA issues prefetches for the current access. Otherwise, Pref-PSA-2MB generates prefetches for the current access. The operation of Pref-PSA-SD is also illustrated with pseudo-code in Figure 7 (C).

[5]This is not the case for cache replacement policies [73] because the cache replacement function domain is a single set.

| Component | Configuration |
|---|---|
| **CPU Core** | 1-8 cores, 4GHz, 352-entry ROB, 4-wide |
| **L1 ITLB/DTLB** | 64-entry, 4-way, 1-cycle, 8-entry MSHR |
| **L2 TLB** | 1536-entry, 12-way, 8-cycle, 16-entry MSHR |
| **L1 ICache** | 32KB, 8-way, 4-cycle, 8-entry MSHR |
| **L1 DCache** | 48KB, 12-way, 5-cycle, 16-entry MSHR |
| **L2 Cache** | 512KB, 8-way, 10-cycle, 32-entry MSHR<br>32 sets for Pref-PSA/Pref-PSA-2MB, 3-bit $C_{sel}$ |
| **LLC (per core)** | 2MB, 16-way, 20-cycle, 64-entry MSHR |
| **DRAM** | 8GB (single-core), 32GB (multi-core), 3200MT/s |
| **Branch Predictor** | hashed perceptron [91] |

Table I
SYSTEM CONFIGURATION.

To update $C_{sel}$, Pref-PSA-SD takes into account the useful prefetches of the two competing prefetchers by looking at the annotation bit (Section IV-B2). Specifically, a cache hit due to a prefetch issued by Pref-PSA (Pref-PSA-2MB) decrements (increments) $C_{sel}$ by one. Empirically, we found that three bits for $C_{sel}$ are adequate to identify the most useful cache prefetcher per execution phase dynamically.

Finally, no matter which prefetcher is activated, we let both Pref-PSA and Pref-PSA-2MB train on all L2C accesses and adjust their prefetching strategy accordingly. Training each prefetcher only when it is selected, as Set Dueling implies when used for cache replacement policies [73], provides poor performance gains due to insufficient training and false pattern observation, as we show in Section VI-B3.

## V. METHODOLOGY

### A. Performance Model

We evaluate our proposals using the ChampSim simulator [88], modeling a 4-wide out-of-order processor and a three-level cache hierarchy, similar to prior work [14], [16]. We model 1-core, 4-core, and 8-core out-of-order machines with 64-byte blocks. Table I presents our experimental setup.

Prefetching is applied upon L2C accesses with prefetched blocks placed into the L2C or LLC, depending on the L2C prefetcher's confidence. There is no prefetching at the L1 caches, and all cache levels use the LRU replacement policy.

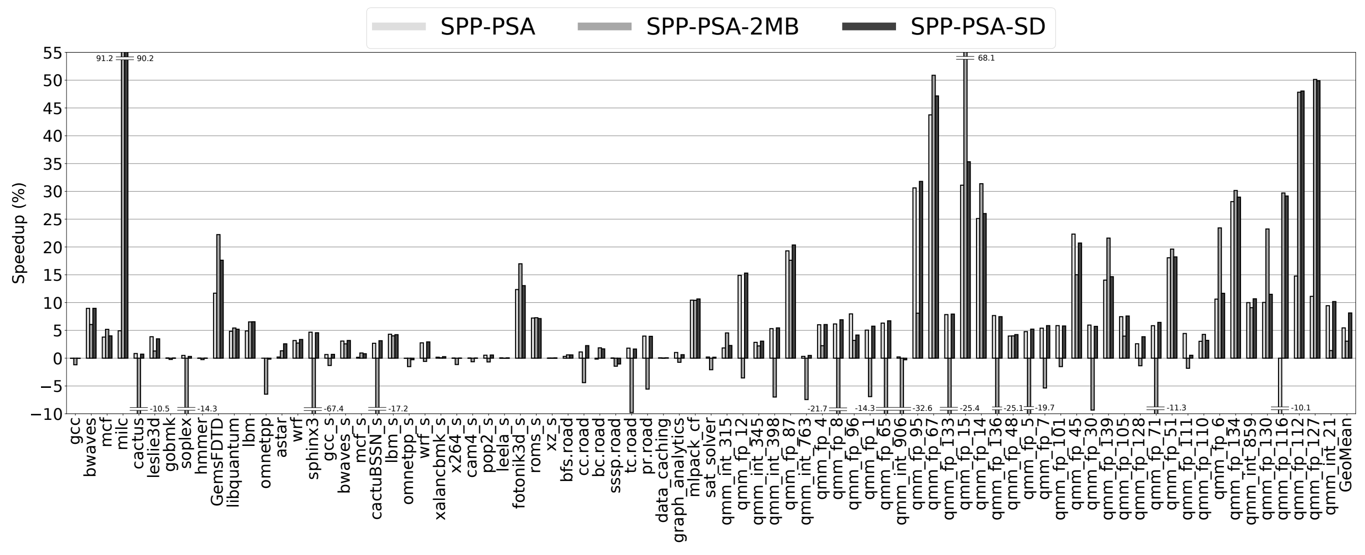


Figure 8. Performance comparison between different page size aware (PSA) versions of the SPP prefetcher. The speedups are computed over the original implementation of SPP that considers only 4KB pages and stops prefetching at 4KB physical page boundaries [14].

Prior work on spatial prefetching for the lower-level caches is limited to 4KB physical page boundaries [3]. However, modern OSes provide support for large pages, as explained in Sections II-B1 and III-B. Thus, we extend ChampSim to concurrently support both 4KB and 2MB pages since the Linux Transparent Huge Pages (THP) mechanism [60] provides automatic and transparent support only for 2MB large pages (Section II-B1). Mapping data into 1GB large pages requires manually using the *libhugetlbfs* [77] since THP does not transparently support 1GB pages. For these reasons, our evaluation considers a system that concurrently supports 4KB and 2MB pages.

We verify that our infrastructure accurately simulates multiple page sizes by measuring the usage of 4KB and 2MB pages for all SPEC CPU 2006 [67], SPEC CPU 2017 [68], and GAP [69] benchmarks, presented in Section V-B, using the page-collect tool [87] on an Intel Xeon E5-2687W machine and compare them with the corresponding usages on our simulation infrastructure. Our experiments reveal that real systems heavily use 2MB pages (on average 85% of the total allocated memory is mapped to 2MB pages across the considered workloads in our system), as also shown in Section III-B. Additionally, our infrastructure simulates multiple page sizes within only 1.8% error compared to the real system measurements for the considered workloads.

*1) Constrained Evaluation:* We test our proposals under different DRAM bandwidth configurations that roughly correspond to three commercial processors (Intel Xeon Gold [70], AMD EPYC [92], and AMD Threadripper [93]), similar to [27]. Moreover, we evaluate scenarios that consider different entries in the L2C MSHR and different LLC sizes. Section VI-B4 evaluates these scenarios. The multi-core evaluation uses the default configuration (Table I).

## *B. Workloads*

We consider an extensive and diverse set of workloads to evaluate our proposals. Specifically, we use all workloads from SPEC CPU 2006 [67] and SPEC CPU 2017 [68] suites, big data workloads included in the GAP suite [69] using the `road` input graph, scale-out applications from CloudSuite [39], a machine learning workload (mlpack [71]), and industrial workloads provided by Qualcomm (QMM) for CVP1 [72]. Workloads with an LLC MPKI of at least 1 are considered memory-intensive and thus considered in our evaluation. Overall, our evaluation considers 195 different traces spanning 80 workloads. All traces were obtained using the SimPoint [94] methodology, and our evaluation reports the weighted mean speedups achieved per application.

*Single-core Performance.* All SPEC, GAP, CloudSuite, and machine learning workloads run the first 250M instructions to warm up the $\mu$architectural structures and 250M instructions are executed to obtain the experimental results. For the QMM workloads, we use 50M warm-up instructions and 100M instructions for gathering results [95].

*Multi-core Experiments.* We randomly generate 100 mixes from our workload set for multi-core evaluation. Both 4-core and 8-core evaluations use the same number of warm-up and simulation instructions as the single-core experiments. We report the weighted speedup over the baseline to obviate speedup overestimation due to applications with high IPC [16], [96]. For each application running on a core, we compute the IPC on the multi-core context and the IPC in isolation on a system with the multi-core specs. Then, 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.

## VI. Evaluation

### A. Considered Prefetchers

This section highlights our proposals' versatility by applying the proposed page size exploitation techniques (Section IV) on a set of four state-of-the-art spatial L2C prefetchers: Signature Path Prefetcher (SPP) [14], Variable Length Delta Prefetcher (VLDP) [15], Perceptron-based Prefetch Filtering (PPF) [16], and Best Offset Prefetcher (BOP) [17]. We also consider the Instruction Pointer Classifier Prefetcher (IPCP) [83] in Section VI-B5 to compare against state-of-the-art L1D prefetching.[6]

### B. Single Core Experiments

*1) Performance:* This section quantifies the single-core performance benefits of making the L2C prefetching module aware of the page size by using the PPM scheme (Section IV-A), while illustrating the source of these benefits. Specifically, we quantify the performance gains of (i) the page size aware (PSA) versions of the considered L2C prefetchers (Section IV-A), (ii) the page size aware versions of the L2C prefetchers that inherently use 2MB pages to index their structures (PSA-2MB), presented in Section IV-B, and (iii) the composite scheme (PSA-SD) that dynamically enables the most appropriate between the PSA and PSA-2MB versions of the L2C prefetcher (Section IV-B).

Starting with the SPP prefetcher, Figure 8 reports the speedups of SPP-PSA, SPP-PSA-2MB, and SPP-PSA-SD over the original SPP implementation that is not aware of the page size, thus it stops prefetching at 4KB page boundaries, across all considered workloads. Overall, SPP-PSA, SPP-PSA-2MB, and SPP-PSA-SD improve geomean performance over the original SPP by 5.5%, 3.0%, and 8.1%, respectively. The main takeaways of this experiment are:

- SPP-PSA greatly improves performance over SPP original across the vast majority of the considered workloads (*e.g.*, `GemsFDTD`, `fotonik3d_s`, `qmm_fp_95`). This happens because SPP-PSA exploits PPM to safely cross 4KB boundaries when a block resides in a 2MB page, resulting in better coverage and timeliness than SPP original.
- SPP-PSA-2MB behaves differently across different applications; for some workloads it greatly outperforms SPP original (*e.g.*, `milc`, `qmm_fp_67`) while for others it degrades performance (*e.g.*, `cactus`, `tc.road`), corroborating our last analysis finding (Finding 4, Section III-C). We observe such behavior due to the SPP-PSA-2MB's intrinsic properties (Section III-C): (i) SPP-PSA-2MB indexes the internal prefetching structures with 2MB pages that provides a coarser representation of the access patterns than indexing with 4KB pages and less aliasing in the prediction tables, and (ii) SPP-PSA-2MB considers more strides for prefetching than SPP-Pref-PSA (SPP-PSA-2MB uses strides ranging between -32768 and +32768 while SPP-PSA uses strides ranging between -64 and +64, as explained in Section III-C). Therefore, for workloads like `milc`, SPP-PSA-2MB outperforms SPP-PSA because it (i) does not suffer from aliasing in the prediction table, and (ii) uses strides larger than 64 that manage to detect patterns that SPP-PSA fails at finding due to only considering deltas smaller than 64. However, for benchmarks like `tc.road`, SPP-PSA-Magic-2MB degrades performance over SPP original and SPP-PSA because indexing its internal structure with 2MB pages erroneously generalizes different access patterns experienced by different 4KB pages residing in the same 2MB memory block into the same prefetching structure entry. In other words, indexing with 4KB pages, regardless of whether the block resides in a 2MB page, is sometimes better than indexing with 2MB pages; this is the case for workloads with 4KB-grain address patterns. The main takeaway is that the SPP-PSA-2MB may positively or negatively impact performance, depending on the workload, motivating the design of the SPP-Pref-PSA-SD module.
- SPP-PSA-SD provides the overall best performance gains over SPP original because it accurately enables the most appropriate prefetcher between SPP-PSA and SPP-PSA-2MB. For benchmarks like `milc` and `qmm_fp_67`, SPP-PSA-SD identifies that SPP-PSA-2MB is more effective than SPP-PSA and primarily enables SPP-PSA-2MB. In contrast, SPP-PSA-SD consistently enables SPP-PSA for benchmarks like `sphinx3`, `pr_road`, and `qmm_fp_12` since it identifies that SPP-PSA-2MB does not provide useful prefetches for these benchmarks. Notably, we observe that for some workloads (*e.g.*, `qmm_fp_87`, `cactuBSSN_s`) SPP-PSA-SD offers better performance than its best performing prefetcher as these workloads benefit from enabling different prefetchers across different execution phases.

[6]We evaluate two IPCP versions; IPCP original that prefetches for intra-4KB patterns and another IPCP version that crosses 4KB page boundaries.

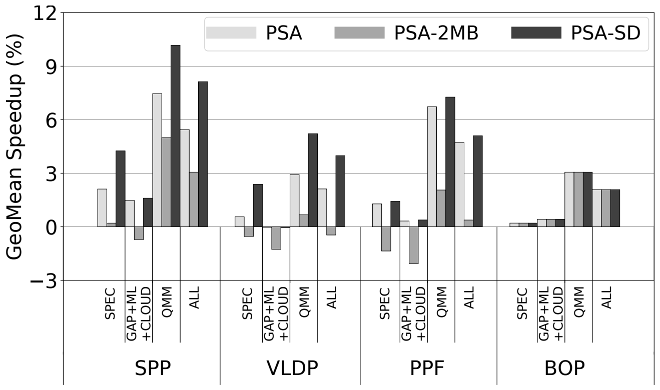


Figure 9. Performance comparison between the PSA, PSA-2MB, and PSA-SD versions of state-of-the-art L2C prefetchers, across all considered benchmark suites. The speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 8.

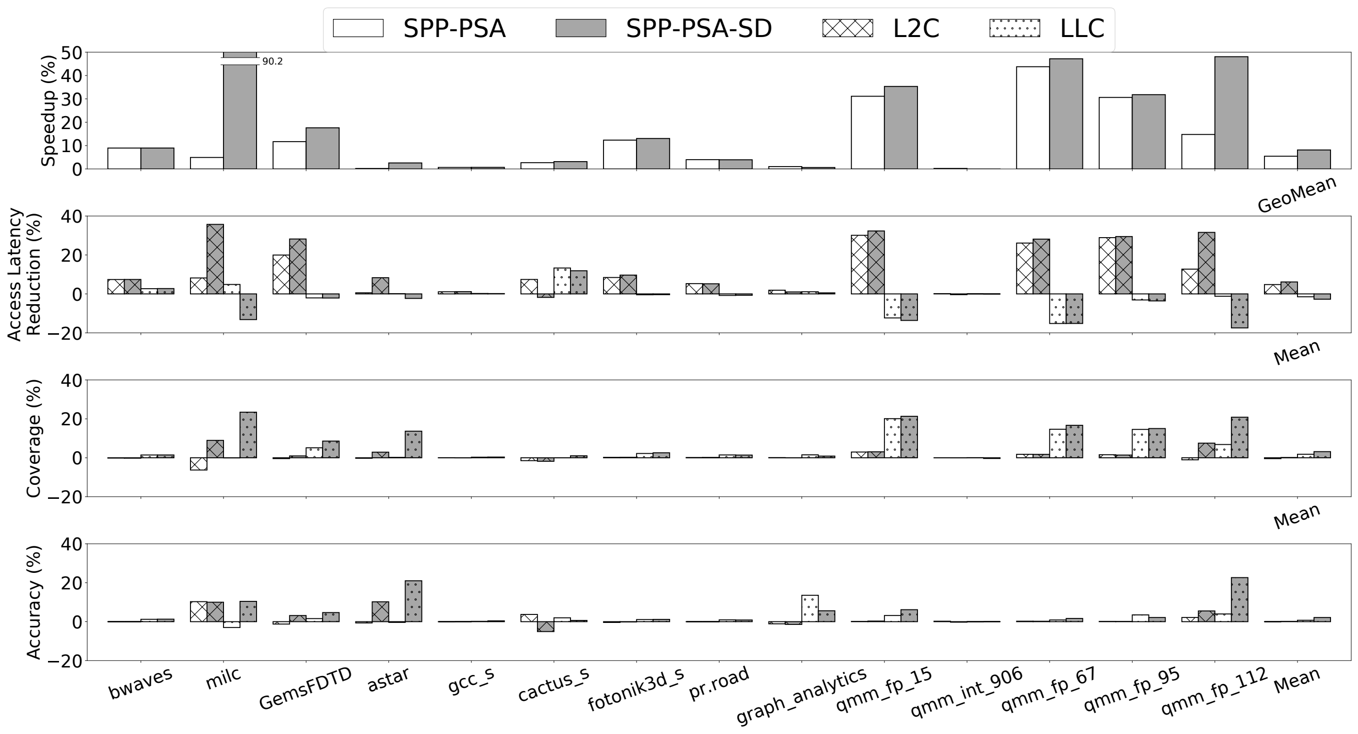


Figure 10. Impact of PSA and PSA-SD versions of SPP on performance, L2C/LLC access latency, L2C/LLC miss coverage, and L2C/LLC prefetching accuracy. All results are computed over the original implementation of SPP. For all considered metrics higher is better.

For workloads operating mainly on 4KB pages (*e.g.*, `soplex`, `hmmer`, `omnetpp`, `gcc_s`, `graph_analytics`) SPP-PSA and SPP-PSA-SD merely improve performance over SPP original because there exist only a few opportunities for safely crossing 4KB pages since these workloads do not use many 2MB pages. Interestingly, SPP-PSA-2MB harms performance for these workloads by erroneously generalizing access patterns to different 4KB pages within the same 2MB memory block into the same prefetching structure entry, thus using the same prefetch deltas.

*Additional Prefetchers.* To demonstrate that our page size exploitation techniques benefit any spatial lower-level cache prefetcher, we consider the VLDP, PPF, and BOP L2C prefetchers (Section VI-A) and evaluate their original implementation as well as their PSA, PSA-2MB, and PSA-SD versions. Figure 9 presents the geomean speedups of the PSA, PSA-2MB, and PSA-SD versions of SPP, VLDP, PPF, and BOP across the considered benchmark suites coupled with a geomean across all workloads. The speedups are computed over the original versions of the considered prefetchers, similar to Figure 8. For example, the speedups of the VLDP-PSA/VLDP-PSA-2MB/VLDP-SD are computed over VLDP original that considers only 4KB pages and stops prefetching at 4KB physical page boundaries.

Overall, the results reported in Figure 9 drive conclusions consistent with the ones reported for the SPP prefetcher in Figure 8: (i) the PSA version of all prefetchers greatly improves performance over the original versions across all considered benchmark suites (*e.g.*, VLDP-PSA improves geomean performance over VLDP original by 3.0% for the QMM workloads), (ii) the PSA-2MB version provides modest performance gains because it improves or degrades performance depending on the workload (*e.g.*, PPF-PSA-2MB improves (degrades) performance for the QMM (SPEC) workloads by 2.1% (1.3%)), and (iii) the PSA-SD version provides the best speedups since the selection logic enables the most appropriate prefetcher between PSA and PSA-2MB (*e.g.*, PPF-PSA-SD outperforms PPF original by 5.1% across all workloads). Finally, all BOP versions (PSA, PSA-2MB, PSA-SD) provide the same speedups because BOP does not use any structure indexed with the page size (Section IV-B), causing BOP-PSA-2MB to degenerate to BOP-PSA. Thus BOP's PSA, PSA-2MB, and PSA-SD versions are identical.

*Non-Intensive Workloads.* To quantify the impact of our proposals on less memory-intensive workloads, we temporarily augment our workload set with all SPEC 2006 and SPEC 2017 workloads no matter their cache MPKI rates, and evaluate all considered prefetchers coupled with all page size exploitation techniques. Across all considered workloads (intensive plus non-intensive), the (PSA, PSA-2MB, PSA-SD) versions of SPP, VLDP, PPF, and BOP improve geomean performance over their original versions by (4.1%, 2.2%, 6.1%), (1.7%, -0.3%, 3.3%), (3.4%, 0.2%,

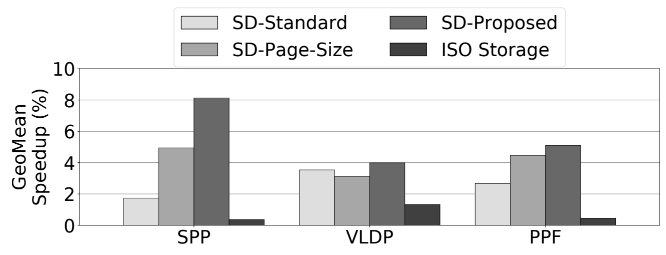


Figure 11. Performance comparison between different implementations of the selection logic of the considered prefetchers' PSA-SD versions.

3.8%), and (1.6%, 1.6%, 1.6%), respectively. All BOP versions provide the same speedups because BOP does not use any structure indexed with the page size (Section IV-B). The speedups are slightly lower than the ones reported when considering only the memory-intensive workloads (Figure 9) because some non-intensive SPEC workloads lower the reported geomean speedups. In addition, we observe that our proposals do not harm the performance of the non-intensive workloads. The main takeaway of this experiment is that page size aware lower-level prefetching provides significant benefits for memory-intensive workloads without negatively impacting the performance of non-intensive workloads.

*2) Sources of Performance Enhancements:* This section justifies the benefits delivered by the proposed page size exploitation techniques (PSA, PSA-SD). This section does not analyze the PSA-2MB version of the prefetchers since it is part of the PSA-SD design that dynamically selects between the PSA and PSA-2MB versions of the prefetchers.

Figure 10 considers different metrics to explain the performance gains of our proposals on the SPP prefetcher. Specifically, we use the following metrics: (i) cache access latency (in cycles) to quantify the impact of our proposals on prefetching timeliness (better prefetching timeliness reduces cache access latency), (ii) miss coverage, and (iii) prefetching accuracy. Moreover, we compute these metrics for the L2C and LLC since SPP directs prefetches in both caches, depending on its internal confidence mechanism. Finally, for this set of experiments, we consider some representative workloads across all considered benchmark suites plus an average across all considered workloads (Mean in Figure 10) for readability. Note that all metrics are computed over the original SPP version, similar to Section VI-B1.

Looking at Figure 10 we observe that the performance gains of our proposals (PSA, PSA-SD) do not have a single root (*e.g.*, higher coverage). The speedups of SPP-PSA and SPP-PSA-SD are caused by positively impacting different metrics, depending on the workload. This is the reason why looking only at the reported averages across 80 workloads does not provide a clear understanding.

For example, SPP-PSA provides modest speedups for the `milc` benchmark, whereas SPP-PSA-SD provides massive speedups for this benchmark due to SPP-PSA-SD enabling the SPP-PSA-2MB prefetcher for this workload, as explained in Section VI-B1. SPP-PSA provides modest speedups for `milc` because it improves prefetching timeliness by significantly reducing the L2C and LLC access latency costs while providing higher prefetching accuracy, at the cost of slightly reducing L2C prefetching coverage. Regarding the speedup of SPP-PSA-SD for `milc` benchmark, we observe that it provides a large coverage increase ($\sim$10% for L2C and $\sim$22% for LLC) coupled with higher prefetching accuracy ($\sim$10% for L2C and $\sim$10% LLC) while reducing the L2C access latency by almost 40%. We observe a slight increase in LLC access latency because most of the LLC misses have been eliminated, and the remaining misses result in cold DRAM accesses. Similar behavior is observed for other workloads like `GemsFDTD` and `qmm_fp_112`.

For workloads like `bwaves`, `fotonik3d_s`, and `pr_road`, SPP-PSA and SPP-PSA-SD provide similar speedups because SPP-PSA-SD mainly enables SPP-PSA since SPP-PSA-2MB is not helpful for these workloads. As a result, SPP-PSA and SPP-PSA-SD have almost the same impact in the metrics presented in Figure 10. For this group of workloads both SPP-PSA and SPP-PSA-SD significantly reduce L2C and LLC access latencies because they experience better prefetching timeliness than SPP original while providing a slight increase in coverage and accuracy.

Looking at the impact of SPP-PSA and SPP-PSA-SD on workloads like `qmm_fp_15`, `qmm_fp_67`, and `qmm_fp_95`, we observe large speedups over SPP original. In addition, SPP-PSA-SD outperforms SPP-PSA since it enables SPP-PSA-2MB in specific execution phases. For these workloads, both SPP-PSA and SPP-PSA-SD experience an accuracy increase up to 10% for both L2C and LLC, a slight L2C coverage increase (<10%), a massive LLC coverage increase (>13%), a massive reduction in L2C access latency due to better prefetching timeliness, and a small increase in LLC latency, because most LLC misses have been eliminated thus the remaining misses result in cold DRAM accesses.

For workloads like `gcc_s`, `graph_analytics`, and `qmm_int_906` both SPP-PSA and SPP-PSA-SD merely improve performance over SPP original because these workloads mainly operate on 4KB pages, thus there is no potential for high performance gains. Consequently, they have a small impact on the metrics of Figure 10, and the proposed page size exploitation techniques merely improve their performance, as shown in Figure 8.

Finally, Figure 10 focuses on SPP to illustrate the sources of performance enhancements of the proposed page size exploitation schemes. We observe similar behavior for the rest of the evaluated prefetchers (VLDP, PPF, BOP).

*3) Different Selection Logic Implementations:* This section highlights the benefits of the proposed selection logic

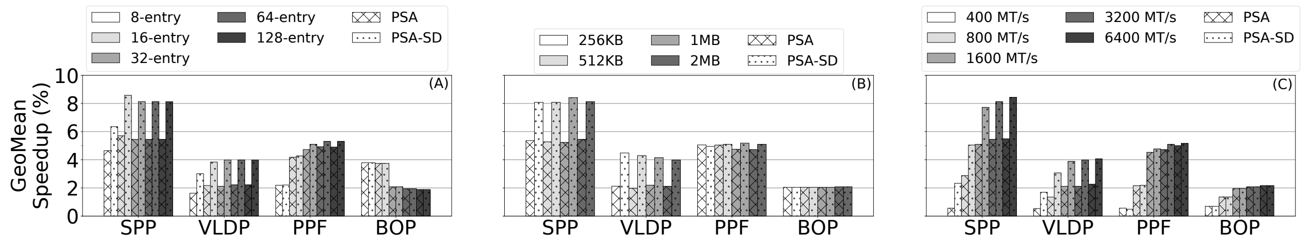


Figure 12. Impact of L2C MSHR size (A), LLC size (B), and DRAM bandwidth (C) on the performance of the PSA and PSA-SD versions of the considered L2C prefetchers. Results are computed over the original implementations of the considered L2C prefetchers.

(Section IV-B) by comparing it against alternative implementations. Specifically, Figure 11 presents the geomean speedups of the PSA-SD versions of all considered L2C prefetchers, across three different selection logic implementations: (i) original implementation of Set-Dueling [73] that trains the PSA and PSA-2MB only when they are selected (SD-Standard), (ii) page size based selection scheme (SD-Page-Size) where the selection logic blindly enables the PSA (PSA-2MB) version of the prefetcher when the accessed block resides in a 4KB page (2MB page), and (iii) the proposed selection logic implementation (SD-Proposed). Moreover, we consider an ISO storage scenario that doubles the storage budget of the original prefetchers' implementations to match the budget of the PSA-SD versions and the cost of the annotation bit (Section IV-B2). Prefetcher BOP is excluded from this experiment since BOP-PSA and BOP-PSA-SD are the same, as shown in Section VI-B1. Finally, the speedups are computed over the original versions of the considered L2C prefetchers, similar to Figure 9.

Figure 11 reveals that SD-Proposed provides the overall highest speedups across all prefetchers (*e.g.*, SD-Proposed outperforms the other selection logic implementations by up to 6.4% for SPP). In addition, we observe that SD-Standard provides lower speedups than SD-Proposed; this happens because SD-Standard trains the PSA and PSA-2MB versions of the prefetchers only when they are selected, whereas SD-Proposed trains both prefetchers across all accesses. Moreover, we find that SD-Page-Size provides good speedups but still performs worse than SD-Proposed. We observe such behavior because indexing the internal structures of the prefetchers with 2MB pages sometimes losses important information due to coarser representation of patterns, leading to sub-optimal prefetching decisions. In other words, blindly considering the page size to enable one of the PSA and PSA-2MB versions is seldom beneficial. Sometimes it is better to assume 4KB (2MB) pages for indexing the internal prefetching structures even if the accessed block resides in a 2MB (4KB) page.[7] This happens when 2MB pages accommodate data structures with orthogonal memory access patterns; in these cases, the prefetcher is more effective at capturing

[7]No matter which page size is considered for indexing, prefetching is permitted within the page where the accessed block resides.

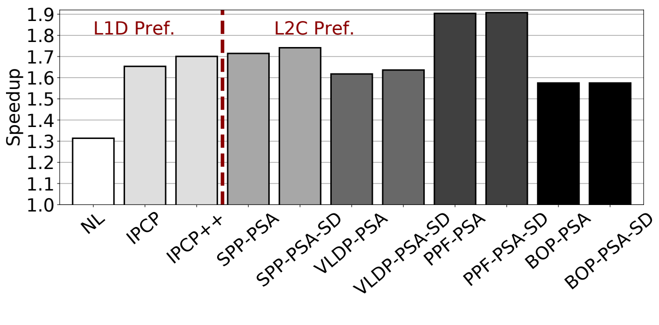


Figure 13. Comparison with state-of-the-art L1D prefetching.

the memory access patterns of the different structures by internally considering 4KB pages since fewer data structures are clustered within one 4KB page than a 2MB page. Finally, the ISO storage scenario's speedups reveal that doubling the prefetchers' size merely improves performance.

*4) Constrained Evaluation:* This section quantifies the impact of various constraints on the performance of the PSA and PSA-SD versions of all considered prefetchers. Figure 12 presents the impact on geomean performance for various L2C MSHR sizes, LLC sizes, and DRAM bandwidths roughly corresponding to various commercial processors (Section V-A1). The speedups are computed over the prefetchers' original versions, similar to Section VI-B1.

Results presented in Figure 12 (A) and (B) reveal that no matter the L2C MSHR size and the LLC capacity the PSA and the PSA-SD versions of all considered prefetchers consistently provide large speedups over the original versions of the prefetchers. For instance, even when the L2C MSHR has 8 entries, SPP-PSA and SPP-PSA-SD improve geomean speedup over SPP original by 4.6% and 6.4%, respectively.

Regarding the impact of DRAM bandwidth (Figure 12, C) on the speedups of our proposals, we observe that the PSA and PSA-SD versions of the prefetchers consistently improve performance over their original versions, even when DRAM bandwidth is 400 MT/s. The main takeaway of this evaluation is that exploiting the page size information to safely prefetch across 4KB physical page boundaries provides large gains even in bandwidth-constrained scenarios.

*5) Comparison with L1D Prefetching:* This section compares the PSA and PSA-SD versions of all considered

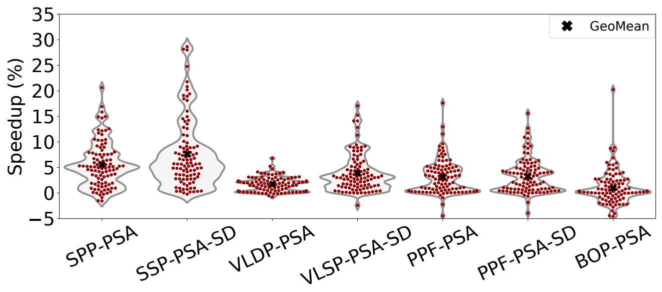


Figure 14. Distribution of 4-core speedups of the PSA and PSA-SD versions of the considered prefetchers across 100 mixes.

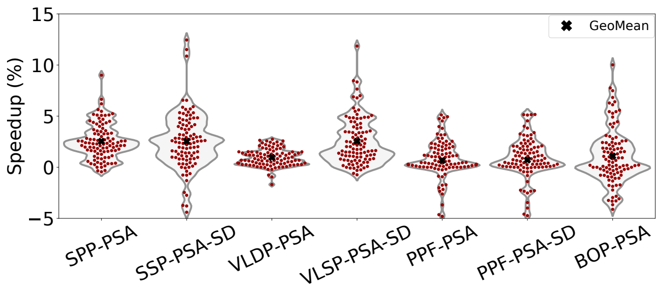


Figure 15. Distribution of 8-core speedups of the PSA and PSA-SD versions of the considered prefetchers across 100 mixes.

L2C prefetchers with the Instruction Pointer Classifier Prefetcher (IPCP) [83] which is a state-of-the-art L1D prefetcher. We evaluate two versions of IPCP: the first (IPCP) stops prefetching at 4KB page boundaries, and the second (IPCP++) is allowed to cross 4KB page boundaries for prefetching only when the page where the prefetched block resides is TLB resident (Section II-C1). Both IPCP and IPCP++ apply prefetching using virtual addresses since they are placed alongside L1D. We also evaluate a next-line prefetcher for reference. Figure 13 presents the speedups of the considered prefetchers across all workloads. The baseline system does not use prefetching at any cache level.

Looking at Figure 13, we observe that the version of IPCP that crosses 4KB page boundaries (IPCP++) delivers higher speedup (4.6% in geomean) than IPCP that stops prefetching at 4KB page boundaries. This happens because IPCP++ experiences higher coverage and better timeliness than IPCP due to 4KB-crossing prefetching. However, the PSA and PSA-SD versions of SPP and PPF outperform both IPCP and IPCP++. For example, SPP-PSA-SD and PPF-PSA-SD provide 9.0% (4.4%) and 24.6% (20.0%) higher speedups than IPCP (IPCP++), respectively. In addition, the versions of VLDP and BOP that exploit the page size information provide speedups slightly lower than IPCP and IPCP++. The main takeaway of this experiment is that page size aware L2C prefetching delivers equal or higher performance enhancement than state-of-the-art L1D prefetching.

### C. *Multi-Core Experiments*

This section presents the performance benefits delivered by our proposals (PSA, PSA-SD) to all considered L2C prefetchers in multi-core contexts. Figures 14 and 15 illustrate the distribution of the speedups of the PSA and PSA-SD versions, across the SPP, VLDP, PPF, and BOP prefetchers, in a 4-core and an 8-core context, respectively. Both 4-core and 8-core experiments use 100 random mixes, as explained in Section V-B. Finally, the speedups reported in Figures 14 and 15 are computed over the original versions of the prefetchers, similar to Section VI-B1.

Our multi-core evaluation reveals that both PSA and PSA-SD versions of all considered L2C prefetchers provide large performance benefits for most of the 4-core and 8-core mixes. For example, SPP-PSA and SPP-PSA-SD provide a geomean speedup of 5.6% and 7.7% over SPP original across 100 randomly generated 4-core mixes. However, in the 8-core context, we observe that PSA and PSA-SD versions of all the prefetchers deliver lower performance enhancements than in the 4-core context. This happens because both 4-core and 8-core evaluations use the same DRAM configuration (Table I in Section V). Therefore, there is less opportunity for improvement by exploiting the page size information for prefetching in the 8-core context due to limited bandwidth compared to the 4-core context.

## VII. Conclusions

This paper proposes the *Page-size Propagation Module (PPM)*, a $\mu$architectural scheme that exploits the prevalence of large pages in systems to enable *safe* prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages. In addition, we propose a module comprised of two page size aware prefetchers that inherently use different page sizes to drive prefetching while using adaptive logic to enable the most appropriate prefetcher per cache access. Across an extensive set of workloads, we show that the proposed page size exploitation techniques provide significant benefits to various state-of-the-art prefetchers.

## VIII. Acknowledgements

The authors are grateful to the MICRO 2022 reviewers for their comments that improved the quality of the paper. This work is supported by the Spanish Ministry of Science and Technology through the PID2019-107255GB project, the Generalitat de Catalunya (contract 2017-SGR-1414), the European Union Horizon 2020 research and innovation program under grant agreement No 955606 (DEEP-SEA EU project), the National Science Foundation through grants CNS-1938064 and CCF-1912617, and the Semiconductor Research Corporation project GRC 2936.001. Georgios Vavouliotis has been supported by the Spanish Ministry of Economy, Industry, and Competitiveness and the European Social Fund under the FPI fellowship No. PRE2018-087046. Marc Casas has been partially supported by the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and ESF 'Investing in your future'.

## References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Computer Architecture News*, vol. 23, 1995.

[2] S. A. McKee, “Reflections on the memory wall,” in *Proceedings of the 1st Conference on Computing Frontiers*, 2004.

[3] S. Mittal, “A survey of recent prefetching techniques for processor caches,” *ACM Computing Surveys*, vol. 49, 2016.

[4] D. Suggs, M. Subramony, and D. Bouvier, “The AMD “Zen 2” Processor,” *IEEE Micro*, vol. 40, 2020.

[5] 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, 2016.

[6] 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, 2012.

[7] 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*, 2020.

[8] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, “The ibm system/360 model 91: Machine philosophy and instruction-handling,” *IBM Journal of Research and Development*, vol. 11, 1967.

[9] R. P. Case and A. Padegs, “Architecture of the ibm system/370,” *Commun. ACM*, vol. 21, 1978.

[10] Levinthal D., “Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors.” https://www.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf.

[11] D. Sager, D. P. Group, and I. Corp, “The microarchitecture of the pentium 4 processor,” *Intel Technology Journal*, vol. 1, 2001.

[12] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner, “Power5 system microarchitecture,” *IBM Journal of Research and Development*, vol. 49, 2005.

[13] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, “Power4 system microarchitecture,” *IBM Journal of Research and Development*, vol. 46, 2002.

[14] 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*, 2016.

[15] 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*, 2015.

[16] 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*, 2019.

[17] P. Michaud, “Best-offset hardware prefetching,” in *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, 2016.

[18] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, “Dspatch: Dual spatial pattern prefetcher,” in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019.

[19] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial memory streaming,” in *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[20] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for data cache prefetch,” in *Proceedings of the 23rd International Conference on Supercomputing*, 2009.

[21] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture*, 2019.

[22] 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*, 2019.

[23] 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*, 2019.

[24] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Domino temporal data prefetcher,” in *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture*, 2018.

[25] S. Kumar and C. Wilkerson, “Exploiting spatial locality in data caches using spatial footprints,” in *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[26] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in *Proceedings of the 46th International Symposium on Microarchitecture*, 2013.

[27] 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*, 2021.

[28] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, “Spatio-temporal memory streaming,” *SIGARCH Computer Architecture News*, vol. 37, 2009.

[29] S. Volos, J. Picorel, B. Falsafi, and B. Grot, “Bump: Bulk memory access prediction and streaming,” in *Proceedings of the 47th International Symposium on Microarchitecture*, 2014.

[30] 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 IEEE Symposium on Security and Privacy*, 2022.

[31] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[32] Y. Chen, L. Pei, and T. E. Carlson, "Leaking control flow information via the hardware prefetcher," *CoRR*, vol. abs/2109.00474, 2021.

[33] 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.

[34] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th International Symposium on Computer Architecture*, 2013.

[35] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th International Symposium on Microarchitecture*, 2013.

[36] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The interaction of architecture and operating system design," *SIGARCH Computer Architecture News*, vol. 19, 1991.

[37] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, 1997.

[38] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd International Symposium on Computer Architecture*, 2015.

[39] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[40] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture*, 2018.

[41] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[42] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[43] 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*, 2021.

[44] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *Proceedings of the 54th International Symposium on Microarchitecture*, 2021.

[45] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *Proceedings of the 2020 47th International Symposium on Computer Architecture*, 2020.

[46] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd International Symposium on Computer Architecture*, 2015.

[47] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, 2016.

[48] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[49] A. Bhattacharjee, D. Lustig, and M. Martonosi, *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers, 2017.

[50] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM SIGSAC Conference on Computer and Communications Security*, 2004.

[51] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Computer Security Applications Conference*, 2006.

[52] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[53] R. W. Carr and J. L. Hennessy, "Wsclock—a simple and effective algorithm for virtual memory management," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, 1981.

[54] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, Mass.: Addison-Wesley, 1997.

[55] "Intel®64 and IA-32 Architectures Optimization Reference Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[56] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the 37th International Symposium on Computer Architecture*, 2010.

[57] Intel Corporation, "TLBs, Paging-Structure Caches, and Their Invalidation," https://composter.com.ua/documents/TLBs_Paging-Structure_Caches_and_Their_Invalidation.pdf, 2008.

[58] "Kernel address space layout randomization," https://lwn.net/Articles/569635/.

[59] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[60] "Transparent Huge Pages," http://lwn.net/Articles/423584/.

[61] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, 2002.

[62] "Intel® 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/en-us/articles/intel-sdm.

[63] "AMD-V™ Nested Paging – White Paper 2008," http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[64] "Database Tuning on Linux OS: Reference Guide for AMD EPYC™ 7002 Series Processors," https://developer.amd.com/wp-content/resources/56783_1.0.pdf.

[65] "Virtual memory support, armv4 and armv5," https://developer.arm.com/documentation/ddi0406/b/Appendices/ARMv4-and-ARMv5-Differences/System-level-memory-model/Virtual-memory-support?lang=en.

[66] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively breaking large pages to improve memory overcommitment performance in vmware esxi," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[67] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, 2006.

[68] "SPEC CPU 2017," https://www.spec.org/cpu2017/.

[69] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[70] "Intel Xeon Gold," https://en.wikichip.org/wiki/intel/xeon_gold/6258r.

[71] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "Mlpack: A scalable c++ machine learning library," *J. Mach. Learn. Res.*, vol. 14, 2013.

[72] "Championship Value Prediction (CVP)," https://www.microarch.org/cvp1/.

[73] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *SIGARCH Computer Architecture News*, vol. 35, 2007.

[74] P. Conway and B. Hughes, "The amd opteron northbridge architecture," *IEEE Micro*, vol. 27, 2007.

[75] "Intel 5-Level Paging and 5-Level EPT," https://ebin.pub/5-level-paging-and-5-level-ept-white-paper-revision-10nbsped.html.

[76] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019.

[77] "Libhugetlbfs," https://lwn.net/Articles/374424/, 2010.

[78] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st International Symposium on Microarchitecture*, 2008.

[79] A. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, 1978.

[80] F. Dahlgren and P. Stenstrom, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *Proceedings of 1st IEEE Symposium on High Performance Computer Architecture*, 1995.

[81] "Arm Architecture Reference Manual for A-profile Architecture," https://developer.arm.com/documentation/ddi0487/latest.

[82] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 Conference on Supercomputing*, 1991.

[83] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proceedings of the 47th International Symposium on Computer Architecture*, 2020.

[84] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th International Symposium on Microarchitecture*, 2011.

[85] "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.

[86] 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*, 2021.

[87] "Page-collect – Capturing Process Memory Usage Under Linux," https://github.com/cslab-ntua/contiguity-isca2020.

[88] "ChampSim," https://crc2.ece.tamu.edu/.

[89] "CVE-2021-4002 Vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2021-4002.

[90] "CVE-2017-15127 Vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2017-15127.

[91] D. Tarjan and K. Skadron, "Merging analysis and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, 2005.

[92] "AMD Epyc 7702P," https://en.wikichip.org/wiki/amd/epyc/7702p.

[93] "AMD Ryzen Threadripper 3990X," https://en.wikichip.org/wiki/amd/ryzen_threadripper/3990x.

[94] 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, 2003.

[95] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez, "CHiRP: Control-flow history reuse prediction," in *Proceedings of the 53rd International Symposium on Microarchitecture*, 2020.

[96] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th International Symposium on Microarchitecture*, 2017.