Exploiting Page Table Locality for Agile TLB Prefetching

Georgios Vavouliotis*† georgios.vavouliotis@bsc.es
Lluc Alvarez*† lluc.alvarez@bsc.es
Vasileios Karakostas‡ vkarakos@cslab.ece.ntua.gr
Konstantinos Nikas‡ knikas@cslab.ece.ntua.gr
Nectarios Koziris‡ nkoziris@cslab.ece.ntua.gr
Daniel A. Jiménez§ djimenez@acm.org
Marc Casas*† marc.casas@bsc.es

*Barcelona Supercomputing Center
‡ National Technical University of Athens
†Universitat Politècnica de Catalunya
§Texas A&M University

***Abstract*— Frequent Translation Lookaside Buffer (TLB) misses incur high performance and energy costs due to page walks required for fetching the corresponding address translations. Prefetching page table entries (PTEs) ahead of demand TLB accesses can mitigate the address translation performance bottleneck, but each prefetch requires traversing the page table, triggering additional accesses to the memory hierarchy. Therefore, TLB prefetching is a costly technique that may undermine performance when the prefetches are not accurate.**

In this paper we exploit the locality in the last level of the page table to reduce the cost and enhance the effectiveness of TLB prefetching by fetching cache-line adjacent PTEs "for free". We propose *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that predicts the usefulness of these "free" PTEs and prefetches only the ones most likely to prevent TLB misses. We demonstrate that combining SBFP with novel and state-of-the-art TLB prefetchers significantly improves miss coverage and reduces most memory accesses due to page walks.

Moreover, we propose *Agile TLB Prefetcher (ATP)*, a novel composite TLB prefetcher particularly designed to maximize the benefits of SBFP. ATP efficiently combines three low-cost TLB prefetchers and disables TLB prefetching for those execution phases that do not benefit from it. Unlike state-of-the-art TLB prefetchers that correlate patterns with only one feature (*e.g.,* strides, PC, distances), ATP correlates patterns with multiple features and dynamically enables the most appropriate TLB prefetcher per TLB miss.

To alleviate the address translation performance bottleneck, we propose a unified solution that combines ATP and SBFP. Across an extensive set of industrial workloads provided by Qualcomm, ATP coupled with SBFP improves geometric speedup by 16.2%, and eliminates on average 37% of the memory references due to page walks. Considering the SPEC CPU 2006 and SPEC CPU 2017 benchmark suites, ATP with SBFP increases geometric speedup by 11.1%, and eliminates page walk memory references by 26%. Applied to big data workloads (GAP suite, XSBench), ATP with SBFP yields a geometric speedup of 11.8% while reducing page walk memory references by 5%. Over the best state-of-the-art TLB prefetcher for each benchmark suite, ATP with SBFP achieves speedups of 8.7%, 3.4%, and 4.2% for the Qualcomm, SPEC, and GAP+XSBench workloads, respectively.

I. Introduction

In paging-based virtual memory systems each memory access requires a translation from the virtual to the physical address space. To do so, the processor looks for the translation in the Translation Lookaside Buffer (TLB). On a TLB miss, the processor traverses the page table to find the missing translation, incurring significant latency and additional memory accesses. Hence, address translation significantly contributes to the total number of memory accesses, especially in workloads with large memory footprints and low locality [1]–[5].

Prior work has quantified the cost of TLB performance [6]–[9] and has proposed approaches to mitigate the overheads of address translation. These approaches mainly fall into three categories: (i) increasing TLB reach by introducing hardware and OS support [6], [10]–[15], (ii) reducing the latency of TLB misses [7], [16]–[21], and (iii) reducing TLB misses by prefetching Page Table Entries (PTEs) [22]–[25]. In this paper we focus on the last category, TLB prefetching, that operates at the microarchitecture level, is independent of the system state, relies only on the memory access pattern of the application, and does not disrupt the existing virtual memory subsystem.

Prior TLB prefetching mechanisms [22]–[24] always trigger a page walk in the background to prefetch a PTE. Since page walks incur additional memory references, TLB prefetching might hurt performance if the prefetched PTEs are not consumed by future TLB accesses. Although TLB prefetching can reduce the number of TLB misses, the large number of additional memory references it triggers can undermine its potential for performance improvement and increase the energy consumption of the system.

This paper exploits the locality of PTEs in the last level of the page table to improve the performance of TLB prefetching while reducing its cost in terms of memory accesses and energy consumption. Thanks to page table locality, contiguous PTEs are stored within the same cache line at the end of each page walk. Prior work leverages page table locality [12], [13], [26], [27] to increase TLB reach and reduce page walks, but does not exploit it to enhance the performance and reduce the cost of TLB prefetching. We demonstrate that naively prefetching all neighboring PTEs into a TLB prefetch queue after a page walk results in sub-optimal performance gains. In response, we propose *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that predicts through sampling which of these neighboring PTEs are more likely to prevent future TLB misses and fetches them into a TLB prefetch

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00016

queue. We highlight that SBFP can be combined with any TLB prefetcher to achieve notable performance enhancements while reducing the memory footprint of page walks and the energy consumption of address translation.

Moreover, this paper proposes *Agile TLB Prefetcher (ATP)*, a composite TLB prefetcher particularly designed to exploit the benefits of SBFP. The design of ATP is driven by our analysis findings which indicate that no single state-of-the-art TLB prefetcher performs best among all the applications, and some workloads do not benefit from TLB prefetching due to irregular patterns. Unlike state-of-the-art TLB prefetchers that correlate patterns with only one feature (*e.g.,* strides, PC, distances), ATP combines three low-cost TLB prefetchers and adapts its prefetching strategy depending on the memory access pattern of the application. To do so, ATP relies on two mechanisms: (i) logic that dynamically selects the most appropriate TLB prefetcher in terms of both the accuracy of the prefetched PTE and also the usefulness of its corresponding free prefetches selected by the SBFP scheme, and (ii) an adaptive throttling mechanism that disables TLB prefetching during phases that do not benefit from it.

In summary, this paper makes the following contributions:

- We evaluate the state-of-the-art TLB prefetchers using industrial workloads provided by Qualcomm [28], the SPEC CPU 2006 [29] and SPEC CPU 2017 [30] benchmark suites, the GAP suite [31], and XSBench [32].
- We propose *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that exploits page table locality by predicting which of the adjacent PTEs present in a 64B cache line are most likely to save future TLB misses and prefetch them into a TLB buffer. We demonstrate that combining SBFP with novel and state-of-the-art TLB prefetchers provides great performance benefits.
- We propose *Agile TLB Prefetcher (ATP)*, a composite TLB prefetcher that combines three low-cost TLB prefetchers and maximizes the impact of SBFP. ATP introduces adaptive selection and throttling schemes to enable the most appropriate TLB prefetcher per TLB miss while disabling TLB prefetching when it is not helpful.
- We propose a unified solution that combines ATP and SBFP. This approach yields a geometric speedup of 16.2%, 11.1%, and 11.8% with 37%, 26%, and 5% average reduction of page walk memory references for the Qualcomm, SPEC, and Big Data (GAP+XSBench) workloads over no TLB prefetching, respectively. Over the best state-of-the-art TLB prefetcher for each benchmark suite, ATP coupled with SBFP improves performance by 8.7%, 3.4%, and 4.2% for the Qualcomm, SPEC, and Big Data workloads, respectively.

II. Background

A. Virtual Memory Subsystem

In paging-based virtual memory systems each memory operation requires a translation from virtual to physical address space. Modern systems provide software and hardware architectural support to reduce the address translation overheads.

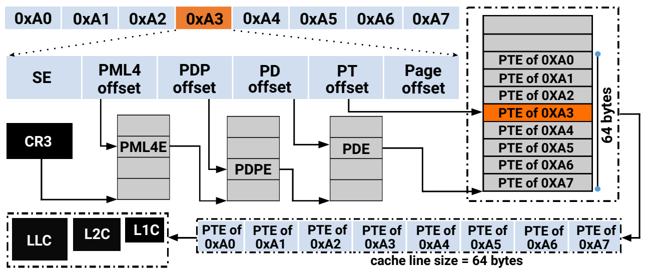

Fig. 1: Page table locality on a x86-64 page table walk.

On the software side, the *Page Table* is an OS-managed structure that contains the virtual-to-physical mappings of all pages loaded to main memory. The x86-64 architecture implements a four-level radix tree page table [33] with levels named PML4, PDP, PD, and PT from root to leaves.[1]

On the hardware side, the *Memory Management Unit (MMU)* accelerates address translation through the *TLB* and the *MMU-Caches*. TLBs cache the most recently used address translations. For each memory access, the TLB is searched for the translation. On a TLB hit, the requested translation is transferred to the CPU. On a TLB miss, a *page walk* traverses the page table to find the requested translation, introducing one reference to the memory hierarchy per page table level. The MMU-Caches (called Page Structure Caches (PSCs) in x86 [35]) avoid some of these references to the memory hierarchy by caching intermediate levels of the page table [7], [18]. The references that miss in the MMU-Caches look for the corresponding translation entries in the memory hierarchy (L1, L2, LLC, DRAM). Hence, the page walk latency depends on the locality in the MMU-Caches and the cache hierarchy.

Modern processors implement multi-level TLBs. Last level TLB misses account for the majority of cycles spent in the TLB miss handler [6], [7], [36]. For the rest of the paper, we use TLB to refer to the last level TLB unless stated otherwise.

B. Page Table Locality

Figure 1 depicts how a page walk is performed in x86-64 architectures and illustrates the locality of the PTEs in the last level of the page table. PTEs are stored contiguously in memory, and each PTE occupies 8 bytes, so a single cache line can store 8 PTEs. When the requested PTE is read from memory at the end of a page walk, it is grouped with 7 neighboring PTEs and they are stored into a single 64-byte cache line. Hence, a cache line holds the requested address translation plus 7 more PTEs that do not require additional memory operations to be prefetched. Note that these neighboring PTEs are contiguous in both virtual and physical address spaces, but they may point to non-contiguous physical pages (depending on the system state, fragmentation).

C. TLB Prefetching

TLB misses trigger page walks that incur significant latency overheads. Prefetching PTEs ahead of demand accesses is an effective approach to mitigate the latency of TLB misses. TLB

[1]Some vendors support a five-level radix tree page table [34].

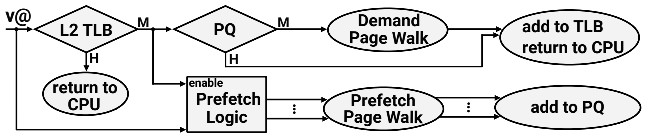

Fig. 2: System with TLB prefetching. Diamonds indicate decision points, circles indicate actions.

prefetching schemes typically use a *Prefetch Queue* (PQ), a small buffer (*e.g.,* 16-64 entries) that holds the prefetched PTEs to avoid polluting the TLB content [23], [37]–[39].

Figure 2 shows the operation of a system that uses a TLB prefetcher. On a memory access, the TLB is initially looked up. On a TLB miss, the requested PTE is looked up in the PQ. If the translation is found in the PQ, the corresponding entry is stored in the TLB, the page walk is avoided, and the processor replays the request. If the PQ does not have the translation, a *demand page walk* is triggered to fetch it from the page table. The TLB prefetching scheme is activated and produces new prefetches when a TLB miss occurs. For each prefetch request, a *prefetch page walk* is triggered in the background. At the end of a prefetch page walk, the prefetched PTE is stored in the PQ. Before issuing new prefetches, TLB prefetchers check whether the corresponding prefetches already reside in the PQ. If so, the corresponding prefetch requests are canceled. Finally, only non-faulting prefetches are permitted.

D. *State-of-the-Art TLB Prefetchers*

Sequential Prefetcher (SP). SP [23], [40] prefetches the PTE located next to the one that triggered the TLB miss.

Arbitrary Stride Prefetcher (ASP). ASP [23] is a table-based TLB prefetcher that captures varying strides patterns. Each prediction table entry has four fields: the PC for indexing, the previous page that caused a TLB miss while accessed by that PC, the stride, and a state describing whether the stride has been unchanged for at least two consecutive table hits.

On a TLB miss, ASP looks up the prediction table for possible hits. On a table miss, the PC is stored in the first field of the corresponding entry, the stride field is invalided, and the counter of the state field is reset. On a table hit, ASP updates the stride field using the current and previous missing pages. If there is no change in the stride field, the counter of the state field is increased; otherwise, it is reset. In case of either table hit or miss, the current page is stored in the second field of the entry. Finally, a prefetch takes place only when the counter of the state field is greater than two.

Distance Prefetcher (DP). DP [23] is a table-based TLB prefetcher that correlates miss patterns with distances between virtual pages that produce consecutive TLB misses. Each prediction table entry has three fields: the corresponding distance for indexing, and two predicted distances.

On a TLB miss, DP computes the distance considering the current and the previous missing virtual pages. On a table hit, DP issues two prefetches using the current missing page and the predicted distances contained in the second and the third fields of the hit entry. Otherwise, a new entry is inserted in the prediction table. In case of either table hit or miss, the entry corresponding to the previous TLB miss is updated by inserting the distance between the current and previous missing pages in the least recently used prediction entry.

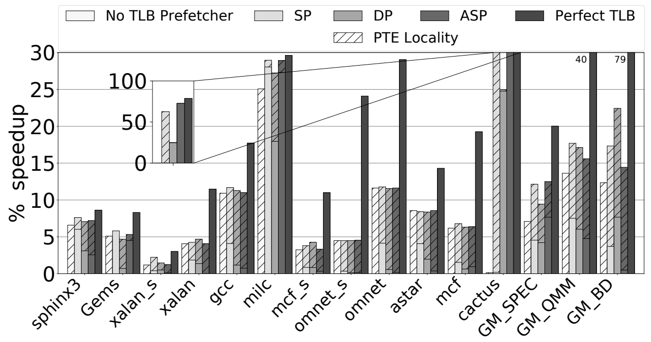

Fig. 3: Performance of SP, ASP, DP and Perfect TLB with and without exploiting PTE locality.

III. Motivation

This section motivates the need for new TLB prefetching approaches and highlights the potential performance improvements when PTE locality is exploited by the state-of-the-art TLB prefetchers[2] (Section II-D), using a 64-entry PQ. To demonstrate the benefits of PTE locality on TLB prefetching, we enhance all the state-of-the-art TLB prefetchers with an unbounded PQ to store all the available PTEs in the cache line returned at the end of each page walk (demand and prefetch). We also consider the case where no TLB prefetcher is used but PTE locality is exploited by storing the available PTEs in the PQ on demand page walks, and an idealized scenario; a Perfect TLB where all the accesses are hits. This evaluation considers industrial workloads provided by Qualcomm (QMM) [28], the SPEC CPU 2006 [29] and SPEC CPU 2017 [30] suites, the GAP [31] suite, and the XSBench [32]. We refer to GAP and XSBench as Big Data (BD) workloads. Section VII explains in detail our experimental setup and the considered workloads.

Our motivational analysis draws the following key findings.

1. TLB prefetching has the potential to improve performance. Figure 3 shows the performance delivered by the state-of-the-art TLB prefetchers, the Perfect TLB, and the scenario that uses an unbounded PQ to exploit PTE locality. The speedups are computed over no TLB prefetching. Without exploiting PTE locality SP, DP, ASP, and Perfect TLB yield a geometric speedup of 4.5%, 4.2%, 7.6%, and 20% for the SPEC, 7.5%, 6.1%, 4.8%, and 40% for the QMM, and 3.7%, 7.6%, 0.5%, and 79% for the BD workloads, respectively. The performance of Perfect TLB reveals large room for improving TLB performance, especially for the QMM and BD workloads.

2. There exists no single TLB prefetcher that performs best across all workloads. Figure 3 also reveals that, for benchmarks showing irregularly distributed stride TLB miss patterns (*e.g.,* cactus), ASP and DP outperform SP. By contrast, for benchmarks with sequential TLB miss patterns (*e.g.,* sphinx3),

[2]We set the configuration parameters of the state-of-the-art TLB prefetchers as proposed in the original papers.

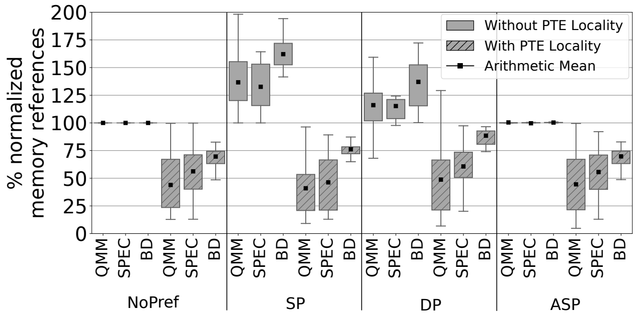

Fig. 4: Normalized memory references due to page walks.

SP outperforms ASP and DP due to conflicts in their prediction tables. These conflicts force ASP and DP to discard the captured stride patterns and, once the execution shows again a regular pattern, they require several TLB misses to identify again the strides. Moreover, SP, ASP, and DP cannot capture highly irregular patterns (*e.g.,* mcf). Although SP, ASP, and DP are ineffective in these irregular scenarios, they trigger a large number of prefetch page walks to serve inaccurate prefetches.

3. Exploiting PTE locality for TLB prefetching purposes has the potential to significantly improve performance. Figure 3 depicts that all state-of-the-art TLB prefetchers and the scenario without TLB prefetcher provide large performance gains when exploiting PTE locality. Greater performance is reported when using TLB prefetchers because the scenario without TLB prefetcher exploits PTE locality only on demand page walks, while TLB prefetchers also issue prefetch page walks, thus they further exploit PTE locality on prefetch page walks.

4. TLB prefetching induces additional references to the memory hierarchy. Figure 4 presents the number of memory references caused by page walks (demand plus prefetch) for SP, DP, ASP, and the scenario without TLB prefetcher (NoPref). The term *memory reference* refers to a page walk reference that is served by the memory hierarchy (L1, L2, LLC, DRAM) since our methodology takes into account cache locality in page walks (Section VII). All scenarios are evaluated with and without exploiting the PTE locality. The normalization factor, 100% in Figure 4, is the total number of memory references without TLB prefetching. Without exploiting PTE locality SP, DP, and ASP burden the system with 63%, 36%, and 1% additional memory references compared to no TLB prefetching, respectively, for the BD workloads. ASP keeps low the memory reference overhead due to the state field of its prediction table which ensures that only accurate prefetches will be issued. We observe similar behavior for the SPEC and the QMM workloads.

5. Exploiting PTE locality reduces page walk memory references. Figure 4 shows that, when PTE locality is exploited, the number of memory references due to page walks is significantly reduced. SP achieves a higher reduction than the other TLB prefetchers because it issues prefetch requests using the +1 stride, which are likely to already be fetched in the PQ due to PTE locality. DP and ASP use larger strides, slightly reducing the impact of PTE locality on them.

The reported improvements of exploiting PTE locality for TLB prefetching purposes assume an ideal and indefinitely large PQ. However, TLB prefetching is fundamentally limited by the PQ size due to latency and area overheads. Therefore, combining PTE locality exploitation with TLB prefetching in the context of a properly sized PQ requires a smart method to select the most useful PTEs per page walk.

IV. Sampling-Based Free TLB Prefetching (SBFP)

This section proposes *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that exploits page table locality to reduce the cost and improve the effectiveness of TLB prefetching. SBFP uses sampling to predict which of the cache-line adjacent PTEs are more likely to prevent future TLB misses and fetches them into the TLB Prefetch Queue (PQ). Moreover, SBFP reduces the negative impact of prefetch page walks, it can be combined with any TLB prefetcher, and it can operate on both demand and prefetch page walks.

A. Pushing the Envelope on Free TLB Prefetching

As described in Section II-B, 7 PTEs that are "free", *i.e.*, they do not require any additional memory operations to be prefetched, are stored in the cache hierarchy at the end of each page walk. The naive approach is to prefetch all available free PTEs into the TLB PQ. However, TLB prefetching is limited by the size of the PQ, the cost of PQ lookups, and the PQ area overhead. Thus, naively storing all available free prefetches per page walk into the PQ may limit the performance benefits by evicting useful prefetches and polluting the PQ with inaccurate prefetches (evaluated in Section VIII-A). Therefore, to exploit the benefits of page table locality with a realistic PQ size, a scheme that dynamically identifies and prefetches only the useful free PTEs per page walk is required. To address this need, we design *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that predicts via sampling the usefulness of the free PTEs per page walk and fetches in the PQ only the most likely ones to save future TLB misses.

B. SBFP Design

We define *free distance* as the distance, within the cache line, between the PTE that holds the demand translation and another PTE that can be obtained for free. Depending on the position of the requested PTE in the cache line, there are 14 possible free distances: from -7 to +7, excluding 0.

1) Design Overview: The SBFP scheme associates each free PTE with a free distance and leverages this information to predict the usefulness of the corresponding PTEs. Figure 5 presents the components and the functionality of the SBFP module: the *Sampler*, the *Free Distance Table (FDT)*, and the *Prefetch Queue (PQ)*. The Sampler is a small buffer that is responsible for detecting phases when free distances, which were previously useless, can provide useful prefetches. To do so, each Sampler entry stores the virtual page and its corresponding free distance for every free PTE that is decided not to be placed in the PQ. The decision whether to place a free PTE into the PQ or the Sampler is made by the FDT, a

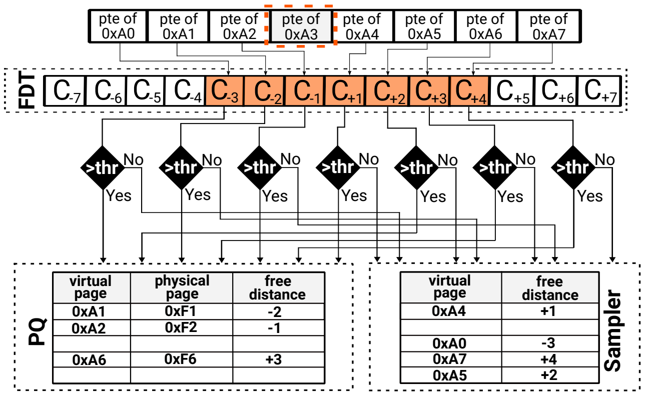

Fig. 5: Sampling-based free TLB prefetching mechanism.

table composed of 14 saturating counters. Each FDT counter monitors the hit ratio of one free distance. Finally, the PQ is a fully associative buffer that stores the virtual page, the physical page and the corresponding free distance of the prefetches.

2) Operation: To explain the operation of SBFP, we consider the example presented in Figure 5 that assumes a page walk triggered for virtual page 0xA3. At the end of the corresponding page walk, we identify the position of the requested PTE inside the cache line by extracting the 3 least significant bits of the page, thus the PTE of 0xA3 resides in position 4 within the cache line. Then we calculate the free distances of all PTEs residing in the same cache line and we associate each PTE with a free distance (*e.g.,* the PTE of 0xA2 is associated with free distance -1).

To determine whether a free prefetch has to be placed in the PQ or the Sampler, we compare the FDT counter corresponding to its free distance with a threshold. If the counter exceeds the threshold, the free prefetch is fetched in the PQ; otherwise, is placed in the Sampler. Specifically, the PTE of 0xA2 has free distance -1, thus we compare the saturating counter of the FDT that corresponds to free distance -1 (C_{-1}) with a threshold to determine if the PTE of 0xA2 should be placed in the PQ or in the Sampler. The same procedure is followed for each free PTE in the cache line. Since PTEs only contain physical addresses, their virtual page numbers must be computed before inserting them in the PQ or the Sampler; the virtual page numbers of the free prefetches are computed as the virtual page number of the demand translation plus their corresponding free distance.

When a PQ or Sampler hit occurs, the FDT counter that corresponds to the free distance of the hit entry is increased. For instance, in case of PQ or Sampler hit caused by a prefetch that was associated with free distance -5, we simply increment the FDT counter C_{-5}. To prevent permanent saturation, we use a decay scheme that shifts right one bit all the FDT counters when one of the FDT counters saturates.

To summarize, SBFP adjusts the values of FDT counters depending on which free distances are frequently producing PQ or Sampler hits, which makes SBFP capable of predicting the most useful free PTEs per page walk. Note that the Sampler is searched only on PQ misses, so its lookup is not

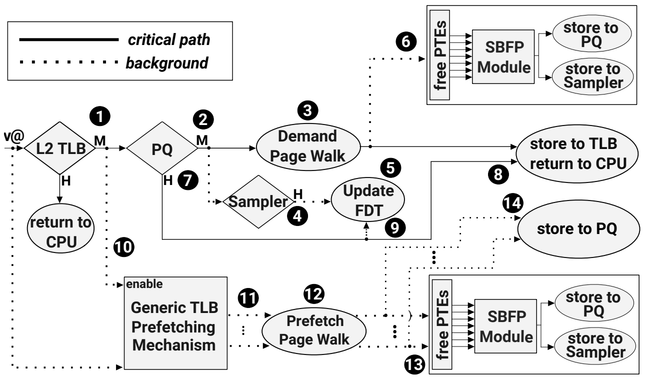

Fig. 6: Combining SBFP with a generic TLB prefetcher Diamonds indicate decision points, circles indicate actions.

placed in the critical path. Our experiments indicate that a FDT composed of 10-bit counters and a 64-entry fully associative Sampler using the FIFO replacement policy are good design points. Finally, the threshold used for determining if a free PTE should be placed in the PQ or the Sampler is 100.

3) Insights on the Effectiveness of SBFP: Modern workloads typically operate on multiple data structures that might favor different sets of free distances. Having perfect knowledge of the most useful free distances per data structure would require a separate FDT per each structure, incurring high complexity and area costs. Alternatively, we propose a generalized SBFP that learns from any stream of accesses and uses a decay mechanism to ensure that only useful free distances will be used. We evaluated the ideal scenario that uses a different FDT per PC that produces at least one TLB miss, and we observed modest performance gains over the generalized FDT that are not worth the required complexity. Once the counters of the generalized FDT saturate, the decay mechanism lowers their values to increase their sensitivity to new data structures. Finally, our analysis indicates that 10-bit counters saturate fast enough to capture the transitions across data structures.

C. Combining SBFP with TLB Prefetching Schemes

This section demonstrates that SBFP can be combined with any TLB prefetching scheme to exploit the benefits of page table locality on both demand and prefetch page walks. To do so, we consider a system that uses a generic TLB prefetching module as well as SBFP. Figure 6 shows in steps the operation of this system, pointing out the interaction between SBFP and the TLB prefetcher. Note that the TLB prefetcher and the SBFP use a shared PQ to store the prefetch requests.

On a TLB miss ❶, the requested translation is looked up in the PQ. On PQ misses ❷, a demand page walk is initiated to fetch the translation ❸. In the background, we look up in the Sampler for possible hits. On Sampler hits ❹, we increment the FDT counter that corresponds to the free distance of the hit entry ❺. When the demand page walk finishes, the SBFP scheme operates and decides which free PTEs should be placed in the PQ and the Sampler, as explained in Section IV-B ❻. On PQ hits ❼, the demand page walk is avoided, the translation is transferred to TLB ❽, and if the hit was produced by a free prefetch we increment the FDT counter that corresponds to the free distance of the hit entry ❾. In either case of PQ hit or miss, the TLB prefetcher is activated ❿, producing new prefetches ⓫. Each prefetch triggers a prefetch page walk to fetch the corresponding translation ⓬. At the end of each prefetch page walk the prefetched PTE is grouped with 7 PTEs that can be prefetched for free due to page table locality. At this point, SBFP is again activated to decide which of the free prefetches should be placed in the PQ or the Sampler, essentially applying lookahead prefetching ⓭. Finally, the prefetched PTEs are stored in the PQ ⓮.

To elaborate more on the operation of SBFP when combined with a TLB prefetcher, we consider the following example: the system experiences a TLB miss on virtual page 0xA3 which also misses in the PQ. As a result, a demand page walk is initiated to fetch the corresponding PTE. When the page walk finishes, SBFP compares the FDT counters with a threshold to identify the most useful free distances for the current miss. Assuming that only free distance -1 exceeds the threshold, SBFP fetches in the PQ the PTE of page 0xA2 (0xA3-1) while the other free PTEs are stored in the Sampler. Next, we further assume that the TLB prefetcher issues a prefetch request for page 0xB7. Similarly, SBFP places the PTE of 0xB6 (0xB7-1) in the PQ while the other free PTEs are stored in the Sampler.

The bottom line is that SBFP can be combined with any TLB prefetcher. Section VIII-A highlights that enhancing state-of-the-art TLB prefetchers with SBFP significantly improves their effectiveness. Finally, in Section V we design a TLB prefetcher aimed at maximizing the benefits of SBFP.

V. Agile TLB Prefetcher (ATP)

This section introduces *Agile TLB Prefetcher (ATP)*, a novel composite TLB prefetcher, implemented as a decision tree. Unlike state-of-the-art TLB prefetchers (Section II-D) that correlate patterns with one feature (*e.g.,* constant strides, PC, distances between pages that produce consecutive TLB misses), ATP captures patterns that correlate well with different features by combining three low-cost TLB prefetchers. To do so, ATP utilizes adaptive selection and throttling schemes to dynamically enable the most appropriate TLB prefetcher and disable TLB prefetching when it is not helpful.

A. ATP Design

Design overview: Figure 7 (left) depicts the hardware components of ATP; three TLB prefetchers (P0, P1, and P2) and a single Prefetch Queue (PQ) shared among them. Moreover, ATP requires modest additional logic for the selection and throttling mechanisms: (i) a saturating counter *'enable_pref'* for the throttling mechanism, (ii) two saturating counters *'select_1'*, *'select_2'* that dynamically select the most accurate TLB prefetcher, and (iii) a Fake Prefetch Queue (FPQ) per constituent prefetcher, which monitors its accuracy to update the values of 'select_1', 'select_2', and 'enable_pref' accordingly. Each FPQ holds only predicted virtual pages and not the corresponding address translations; hence, the term *fake*.

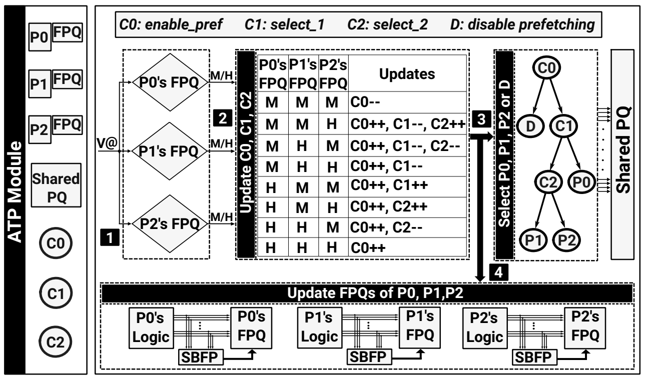

Fig. 7: Design and functionality flowchart of ATP.

Operation: Figure 7 (right) shows step-by-step the operation of ATP. First, ATP looks for the translation of the missing virtual page in the FPQs of all constituent TLB prefetchers $\boxed{1}$. Depending on the search outcome, the saturating counters are updated in step $\boxed{2}$. For example, in case of hit in at least one FPQ, 'enable_pref' increases its value for issuing new prefetch requests. Otherwise, 'enable-pref' is decreased, *i.e.*, increases its confidence for disabling TLB prefetching.

Next, ATP uses the updated values of the saturating counters to make a decision for the current miss. The 'enable_pref' counter is responsible for choosing whether to enable or disable TLB prefetching. If the most significant bit of 'enable_pref' is one, the decision is to issue new prefetch requests. In this case, 'select_1' is probed to select the individual TLB prefetcher that will generate prefetches for the current TLB miss $\boxed{3}$. If then the most significant bit of 'select_1' is one, the prefetcher residing in the right leaf (P0) is selected; otherwise 'select_2' is responsible for selecting which prefetcher should operate. Likewise, if the most significant bit of 'select_2' is one, prefetcher P2 is selected; otherwise prefetcher P1 issues prefetches. Finally, if the most significant bit of 'enable pref' is zero no prefetch request will be issued.

Subsequently, we update the content of all FPQs $\boxed{4}$. All the constituent TLB prefetchers store in their own FPQ the virtual pages corresponding to the prefetches that they would issue if they were allowed to individually produce prefetches plus the free prefetches that SBFP would select after the completion of each fake page walk. Using these "fake prefetches" we track the usefulness of each TLB prefetcher for future TLB misses.

B. Building Blocks of ATP

ATP is composed of three easily implementable TLB prefetchers that are presented below.

Stride Prefetcher (STP). STP is a more aggressive version of SP (Section II-D). STP uses the strides $\{-2,-1,+1,+2\}$ for prefetching. On a TLB miss for virtual page A, STP will prefetch the PTEs of the pages $\{\text{A}-2,\text{A}-1,\text{A}+1,\text{A}+2\}$.

H2 Prefetcher (H2P). H2P keeps track of the last two observed distances between virtual pages that cause a TLB miss. Assuming that $d(X,Y)$ represents the signed distance between the pages X and Y, and A, B, and E are the last three virtual pages that caused a TLB miss, H2P will prefetch the PTEs of the virtual pages: $E + d(E, B)$ and $E + d(B, A)$.

Modified Arbitrary Stride Prefetcher (MASP). MASP is an evolution of ASP [23], [41]. To issue prefetch requests, ASP requires two consecutive hits in a certain prediction table entry to display the same stride. While this policy increases the accuracy of ASP, it misses prefetching opportunities. Thus, we design MASP, implementing two modifications to ASP: (i) the requirement of observing the same stride for at least twice consecutively is removed, and (ii) a second prefetch takes place for each TLB miss, using the virtual page and its corresponding stride. Each entry of the prediction table of MASP has three fields: the PC for indexing, the previous virtual page that caused a TLB miss while being accessed by that PC, and the corresponding stride. Assume a TLB miss for virtual page A that hits in the prediction table of MASP, where the respective entry has in the second field the virtual page E and in the third field the stride +5. MASP will prefetch the PTEs of the pages $A + 5$ and $A + d(A, E)$, where $d(X, Y)$ computes the signed distance between pages X and Y.

Insights on the effectiveness of ATP. ATP enables STP, H2P, and MASP when the TLB miss stream correlates well with small strides, the distance between virtual pages that produce TLB misses, and the PC, respectively. The aggressiveness of STP and H2P may negatively impact both performance and number of triggered page walks, as Section VIII-A highlights. ATP minimizes these negative effects by enabling STP or H2P only when they are likely to produce accurate prefetches. When the TLB miss stream exhibits irregular patterns, the throttling scheme of ATP disables prefetching until it observes again patterns that are predictable by at least one of its constituent prefetchers, leveraging the operation of the FPQs. We assign leaf nodes P0, P1, and P2 to H2P, MASP, and STP, respectively. Finally, our experiments indicate that 8-bit, 6-bit, and 2-bit counters are good design points for 'enable pref', 'select_1', and 'select_2', respectively. Each FPQ is a 16-entry fully associative buffer and uses the FIFO replacement policy.

VI. Important Considerations for ATP and SBFP

This section elaborates on important aspects that both ATP and SBFP take into account when operating in a system.

Impact on Page Replacement Policy. Since prefetches are speculative events, they should ideally not influence the access bits of the prefetched pages. However, the memory consistency model of x86 architectures dictates that TLBs are allowed to accommodate translations that have their status bits on, *i.e.*, all TLB prefetches are obliged to set the access bits. As a consequence, inaccurate prefetches can negatively affect the page replacement policy and lead to sub-optimal decisions. Prior TLB prefetching works do not consider the impact on the page replacement policy due to the growing memory capacities. However, with the advent of heterogeneous memories, the OS has to migrate data between fast and slow memories, so accurately setting the access bit is very important today. Section VIII-E shows that our proposal, ATP coupled with SBFP, has a negligible impact on the page replacement policy.

Component	Description
L1 ITLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle, 4-entry MSHR, 1 page walk / cycle
Page Structure Caches	3-level Split PSC, 2-cycle. PML4: 2-entry, fully; PDP: 4-entry, fully; PD: 32-entry, 4-way.
Prefetch Queue	(16-64)-entry, fully assoc, 2-cycle
Sampler	64-entry, fully assoc, 2-cycle
L1 ICache	32KB, 8-way, 1-cycle, 8-entry MSHR
L1 DCache	32KB, 8-way, 4-cycle, 8-entry MSHR, next line prefetcher
L2 Cache	256KB, 8-way, 8-cycle, 16-entry MSHR, ip stride prefetcher
LLC	2MB, 16-way, 20-cycle, 32-entry MSHR
DRAM	4GB, tRP=tRCD=tCAS=11

TABLE I: System simulation parameters.

Multiple Page Sizes. Neither ATP nor SBFP require any modifications to support multiple page sizes. Since the page size is known after address translation, ATP issues two prefetch requests per prefetch candidate assuming 4KB and 2MB pages and, when the page granularity is known, one of the prefetch page walks is discarded. This approach does not imply additional complexity since modern architectures support speculative page walks [35]. SBFP checks whether the free prefetches are valid translation entries before adding them in the PQ or the Sampler (either valid PT entries or PD entries), even when PD entries that map 2MB pages are next to PD entries that point to PT entries. Finally, the PQ is fully associative, which avoids page size indexing implications [35].

Context Switches. ATP and SBFP leverage small structures that quickly warm up and are flushed at context switches, so they do not need to be tagged with address space identifiers.

VII. Methodology

Workloads. We consider a large set of workloads provided by Qualcomm (QMM) for CVP1 [28], all the benchmarks from SPEC CPU 2006 [29] and SPEC CPU 2017 [30] suites, and big data workloads included in GAP [31] suite and the XSBench [32]. The QMM set includes industrial workloads. The GAP suite includes graph processing kernels using five different input graphs. We report results for the two input graphs that produce the most TLB intensive combinations per kernel. XSBench is evaluated using all different grid types, and we present results for the two most TLB intensive ones. We refer to GAP and XSBench workloads as Big Data (BD) workloads because they have massive memory footprints [5]. Workloads with a TLB MPKI rate of at least 1 are considered TLB intensive and thus taken into account in our evaluation. After the MPKI selection, our set of workloads includes 125 QMM workloads, 12 SPEC CPU workloads, and 13 BD workloads. All traces were obtained using the SimPoint [42] methodology. Each SPEC CPU and BD workload runs 250 million warmup instructions and one billion instructions are executed to measure the experimental results. For the QMM workloads we use 50 million warmup instructions and 100 million instructions for measuring the results [43].

Simulation Infrastructure. For evaluation we use Champ-Sim [44], a detailed simulator that models a 4-wide out-of-

Prefetcher	Description
SP	Static Free Distances: $\{+1, +3, +5, +7\}$
DP	Distance-table: 64-entry, 4-way. Static Free Distances: $\{-2, -1, +1, +2\}$
ASP	PC-table: 64-entry, 4-way. Static Free Distances: $\{-1, +1, +2\}$
STP	Static Free Distances: $\{+1, +2\}$
H2P	Static Free Distances: $\{+1, +2, +7\}$
MASP	PC-table: 64-entry, 4-way. Static Free Distances: $\{+1, +2\}$
ATP	MASP & STP & H2P prefetchers. Fake PQ: 16-entry, fully assoc. Static Free Distances: $\{+1, +2\}$

TABLE II: Configuration of all TLB prefetchers.

order processor. We extend ChampSim with a realistic page table walker used in x86 architectures, modeling (i) the variant latency cost of page walks, (ii) the page walk references to memory hierarchy, and (iii) the cache locality in page walks, similar to prior works [25]. Specifically, we simulate a 4-level page table, a page table walker, and a 3-level split PSC. The page table walker supports up to 4 concurrent TLB misses, similar to Skylake microarchitecture [37], while one page walk can be initiated per cycle. Table I summarizes our setup.

We implement and evaluate the TLB prefetchers explained in Sections II-D and V. Table II presents their configuration. Our evaluation focuses on 4KB pages but we also evaluate large pages in Section VIII-B4. The energy consumption is measured using CACTI 6.5 [45] with 22nm technology.

VIII. Evaluation

A. Impact of SBFP

To highlight the benefits of SBFP we compare it against the following scenarios: (i) free prefetching is not exploited (NoFP), *i.e.*, free prefetches are not stored in the PQ; (ii) all free prefetches are naively placed in the PQ (NaiveFP); (iii) each prefetcher uses its own optimal set of free distances based on a static offline exploration that identifies the most useful free distances per TLB prefetcher (StaticFP). To do so, we explore all possible sets of free distances and measure the performance of each TLB prefetcher with them. Table II presents the optimal set of statically selected free distances for the state-of-the-art and the new TLB prefetchers.

1) Performance: The impact of free TLB prefetching on system's performance regarding the above explained scenarios for the state-of-the-art and the new TLB prefetchers is presented in Figure 8, assuming a 64-entry PQ. We also comment on different PQ sizes in the end of this section. The speedup results are computed over no TLB prefetching.

Figure 8 reveals that all evaluated prefetchers achieve high performance gains for all scenarios considering free prefetching (NaiveFP, StaticFP, SBFP) than when free prefetching is not exploited (NoFP). We observe this behavior because (i) the free prefetches provide PQ hits that reduce demand page walks, and (ii) most of the prefetch requests have already been prefetched for free, avoiding prefetch page walks. Overall, ATP with SBFP yields a geometric speedup of 16.2%, 11.1%, and 11.8% for the QMM, SPEC, and BD workloads, respectively. Regarding the TLB MPKI rates, ATP with SBFP reduces the TLB MPKI from 13.9 to 8.2 (41% reduction) for the QMM, from 3.4 to 1.46 (56% reduction) for the SPEC, and from 38.9 to 29.2 (25% reduction) for the BD workloads.

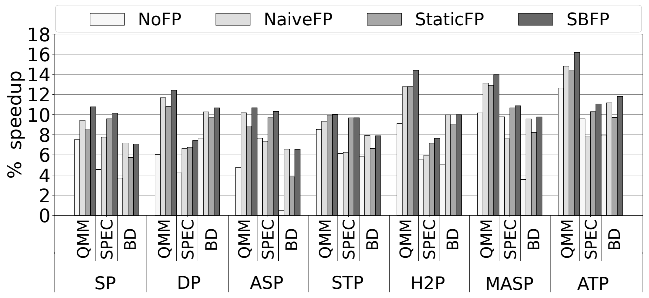

Fig. 8: Performance impact of free TLB prefetching scenarios.

Next, we compare ATP with SBFP, with the best state-of-the-art TLB prefetcher (it is not always the same across benchmark suites and scenarios that exploit free prefetching). ATP with SBFP outperforms the best state-of-the-art TLB prefetcher with NoFP by 8.7%, 3.4%, and 4.2% for the QMM, SPEC, and BD workloads, respectively. In addition, ATP with SBFP improves performance over the best state-of-the-art TLB prefetcher with NaiveFP by 4.6%, 3.4%, and 1.6% for the QMM, SPEC, and BD workloads, respectively. Finally, ATP with SBFP outperforms the best state-of-the-art TLB prefetcher with StaticFP by 5.4%, 1.4%, and 2.1% for the QMM, SPEC, and BD workloads, respectively.

Figure 8 shows that NaiveFP outperforms StaticFP for the QMM and BD workloads. For the SPEC workloads the opposite behavior is observed. This happens because the static selection always uses the overall most useful free distances based on static exploration, but cannot use the non-selected free distances that are seldom beneficial in specific execution phases. In these cases, NaiveFP outperforms StaticFP because it fetches all available free PTEs in the PQ. The main disadvantage of NaiveFP over StaticFP is that it does not examine the usefulness of the free prefetches, resulting in PQ thrashing. However, SBFP identifies the useful free PTEs per execution phase, combining the advantages of NaiveFP and StaticFP.

Our evaluation shows that using a PQ with 16 and 32 entries provides an average performance reduction of 56% and 32% with respect to a 64-entry PQ, respectively. Larger PQs provide negligible performance improvements over a 64-entry PQ. Thus, a 64-entry PQ is optimal design point for this work.

2) Cost of TLB prefetching: To highlight that free prefetching reduces the cost of TLB prefetching, we present in Figure 9 the normalized number of memory references triggered by page walks (demand plus prefetch) for all considered scenarios that exploit free prefetching and TLB prefetchers. The term *memory reference* refers to a page walk reference that is served by the memory hierarchy (L1, L2, LLC, DRAM); note that we take into account cache locality in page walk memory references (Section VII). The normalization factor, 100% in Figure 9, is the total number of memory references for demand page walks without TLB prefetching. We observe a large increase in memory references when free prefetching is not exploited (NoFP) because all prefetches require a

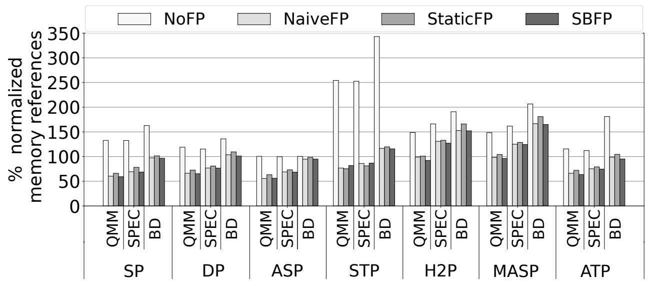

Fig. 9: Normalized memory references due to page walks.

prefetch page walk, and the reduction of the demand page walks is smaller than the number of prefetch page walks introduced. Focusing on the BD workloads, SP, DP, ASP, STP, H2P, MASP, and ATP require 63%, 36%, 1%, 250%, 90%, 106%, and 81% additional memory references compared to the scenario without TLB prefetching, respectively.

All techniques that exploit free prefetching significantly reduce the number of page walk memory references because (i) the majority of the prefetches that require a prefetch page walk are proactively fetched in the PQ as free prefetches, and (ii) free prefetches provide PQ hits that save demand page walks. All considered prefetchers experience their highest reduction in terms of memory references with SBFP. This behavior occurs because SBFP saves more TLB misses than NaiveFP and StaticFP, *i.e.*, it eliminates more demand page walks, as presented in Figures 8. Overall, ATP with SBFP eliminates by 37%, 26%, and 5% the number of memory references due to page walks compared to the scenario without TLB prefetching for the QMM, SPEC, and BD workloads respectively. For readability, Figure 9 shows the normalized number of memory references triggered by both demand and prefetch page walks. However, memory references that hit in the cache hierarchy incur lower latency than the ones going to DRAM, and the latency cost of a demand page walk is more critical than the latency of a prefetch page since the former takes place in the critical path while the latter performs in the background. Section VIII-B2 elaborates on these implications.

B. *Evaluation of ATP coupled with SBFP*

This section compares ATP coupled with SBFP against the state-of-the-art TLB prefetchers and other approaches that improve TLB performance. All prefetchers use a 64-entry PQ.

1) Performance: Figure 10 presents the performance of SP, DP, ASP, and ATP with SBFP for all evaluated workloads. For the SPEC workloads we also show the geometric mean for the whole suite (GM_All), including also the non TLB intensive workloads. For the TLB intensive workloads, ATP with SBFP outperforms all state-of-the-art prefetchers, achieving geometric speedups over the best already proposed TLB prefetcher of 8.7%, 3.4%, and 4.2% for the QMM (up), SPEC (down), and BD (middle) workloads, respectively. Focusing on the BD workloads, we observe that ATP coupled with SBFP provides the overall best improvement. Moreover, for workloads such as xs.nuclide and sssp.twitter DP provides high performance since they exhibit great distance correlation. For the xs.nuclide,

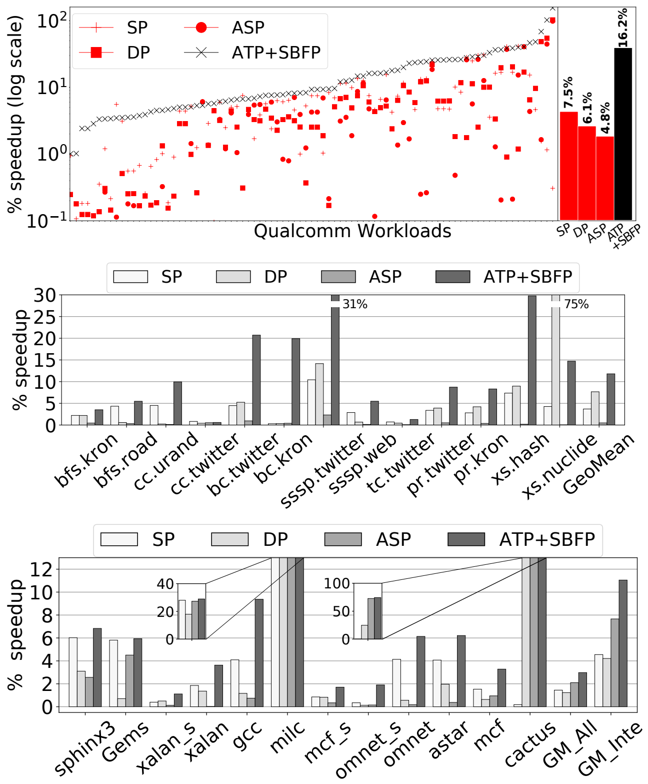

Fig. 10: Performance comparison between ATP coupled with SBFP and state-of-the-art TLB prefetchers.

DP outperforms ATP since ATP enables H2P when distance correlation is observed, although DP is capable of detecting more complex distance patterns compared to H2P.

As Figure 8 highlights, ATP significantly outperforms its constituent prefetchers, STP, H2P, and MASP, highlighting that ATP efficiently combines these prefetchers by selecting the most appropriate prefetcher per TLB miss and disabling prefetching when required. To validate these statements, Figure 11 shows the fraction of time that ATP enables each constituent prefetcher. When STP, H2P, or MASP cannot capture the access patterns of the SPEC workloads, ATP disables prefetching (*e.g.,* xalan_s, mcf). For benchmarks with strided patterns (*e.g.,* milc), ATP enables mostly STP. MASP is enabled only when the TLB miss stream correlates well with the PC (*e.g.,* cactus, mcf_s). As explained in Section V, ATP enables H2P only when it is confident that H2P will produce useful prefetches because H2P uses large distances that may pollute the PQ content in case of inaccurate prefetches. Figure 11 reveals that the SPEC workloads are not benefited by the observed distance correlation, thus ATP never enables H2P. Contrarily, a large number of the QMM and BD workloads (*e.g.,* sssp.twitter, xs.nuclide), exhibit distance correlation among TLB misses. Hence, ATP enables H2P 12% and 34% of the time for the QMM and BD workloads, respectively. Specifically, for xs.nuclide ATP selects H2P for prefetching 61% of the time, explaining why DP outperforms

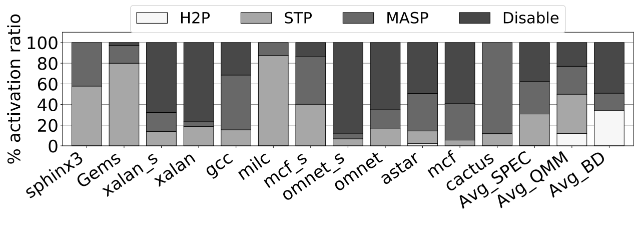

Fig. 11: Fraction of time that ATP selects MASP, STP, H2P or disables TLB prefetching.

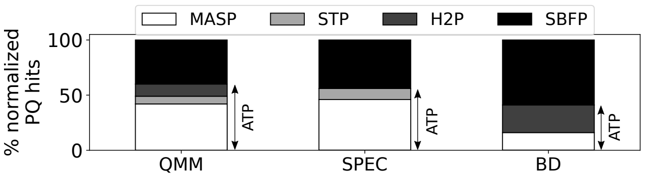

Fig. 12: Percentage of PQ hits provided by ATP (its constituent prefetchers) and SBFP.

ATP for this workload. This evaluation verifies that combining ATP with SBFP addresses all the findings of Section III.

Furthermore, Figure 12 presents a breakdown of the normalized number of PQ hits provided by our proposal across all considered workloads, *i.e.*, it shows the fraction of PQ hits provided by the ATP and the SBFP modules. Moreover, it breaks down the PQ hits provided by ATP into three subcategories; PQ hits provided by MASP, STP, and H2P which are the constituent prefetchers of ATP. On average, prefetch requests issued by the constituent prefetchers of ATP are responsible for 60%, 56%, and 41% of the PQ hits regarding the QMM, SPEC, and BD, respectively. Notably, SBFP also saves a big fraction of the TLB misses, since it is responsible for 40%, 44%, and 59% of the total PQ hits for the QMM, SPEC, and BD workloads, respectively. The bottom line is that both ATP and SBFP play an equally significant role at achieving significant performance enhancements.

2) Cost of TLB Prefetching: Figure 13 presents the average normalized number of memory references due to page walks (demand plus prefetch) out of the total number of memory references for demand page walks without TLB prefetching. Moreover, it shows (i) a breakdown of the memory references caused by demand and prefetch page walks, and (ii) a breakdown of the level of the memory hierarchy that serves the memory references of both demand and prefetch page walks. For the QMM workloads, ATP with SBFP reduces memory references by 37%, while SP, DP, and ASP burden the system with 33%, 19%, and 1% additional memory references. We report similar results for the SPEC workloads. For the BD workloads, we observe lower reduction on memory references than SPEC and QMM workloads because all prefetchers are unable to accurately detect highly irregular TLB miss patterns.

Notably, ATP with SBFP provides the highest reduction in demand page walks for all considered workloads because it provides more PQ hits. Our proposal also reduces the number of memory references of prefetch page walks compared to the other prefetchers because (i) it exploits SBFP to prefetch

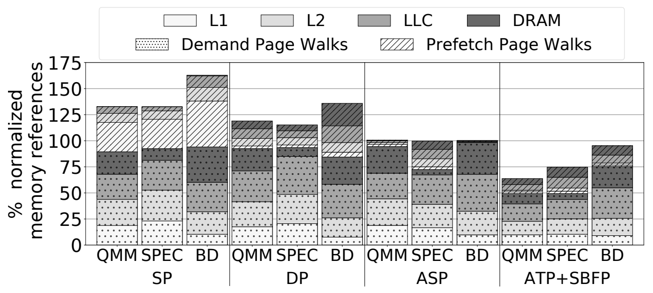

Fig. 13: Normalized memory references due to page walks.

PTEs that otherwise would need a separate prefetch page walk to be fetched, (ii) the throttling mechanism of ATP disables TLB prefetching when it is not helpful, and (iii) for strided patterns the selection mechanism of ATP enables STP, which uses small strides that are mostly served by free prefetches. In addition, our proposal provides higher reduction in the memory references that require a DRAM access than the other prefetchers among all workloads. ATP with SBFP drastically reduces the DRAM accesses of demand page walks, which provides great performance benefits, at the cost of introducing some DRAM accesses for prefetch page walks, which are not in the critical path. The takeaway of this experiment is that our proposal reduces the number of page walk memory references, their cost, and the performance penalties they cause.

3) Hardware Cost: Each PQ entry requires 36 bits for the virtual page, 36 bits for the physical page plus 5 attribute bits. Each prediction table entry of MASP stores 60 bits for the PC, 36 bits for the virtual page, and 15 bits for the stride. Each FPQ entry stores 36 bits for the virtual page. Considering a 64-entry PQ, SP, DP, ASP, and ATP require in total 0.60KB, 0.95KB, 1.47KB, and 1.68KB, respectively. ATP is slightly more expensive than the state-of-the-art TLB prefetchers. Each Sampler entry of the SBFP requires 36 bits for the virtual page plus 4 bits for the free distance. The FDT uses 10-bit counters. Hence, SBFP requires 0.31KB to be implemented.

4) Large Pages: To examine the impact of large pages, we evaluate ATP coupled with SBFP and the state-of-the-art TLB prefetchers using 2MB pages, similar to prior work [25], [46]. We observe significant MPKI reduction for most workloads when 2MB pages are used, although many of them still experience high TLB MPKI rates. For these workloads, ATP with SBFP reduces by 88% on average the TLB misses that 2MB pages cannot eliminate. Figure 14 shows the speedup impact of SP, DP, ASP, and ATP with SBFP for the 2MB scenario. The baseline implies using 2MB pages without TLB prefetching. ATP with SBFP provides a geometric speedup of 5.1%, 4.3%, and 9.9% for the QMM, SPEC, and BD workloads, respectively. For SP, DP, and ASP we observe negligible speedup results. Note that the SPEC set of workloads contains only one benchmark, mcf, since for the other workloads the 2MB pages eliminate the observed TLB misses. Finally, most of the PQ hits (89% on average) are produced by free prefetches because free prefetching with 2MB pages covers a larger amount of memory compared to 4KB pages.

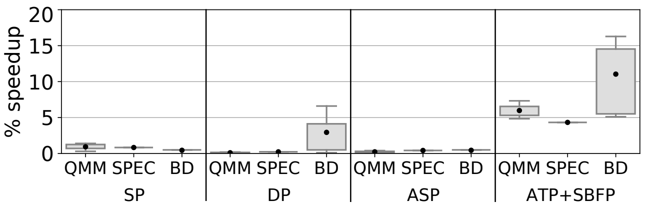

Fig. 14: Performance comparison with 2MB pages.

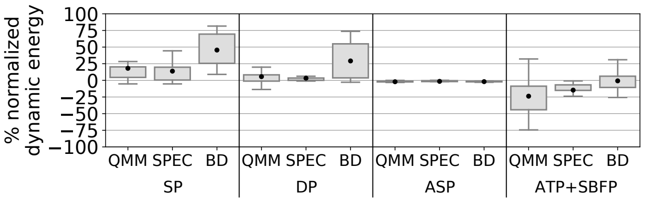

Fig. 15: Normalized dynamic energy consumption.

5) Energy Consumption: To measure the baseline dynamic energy consumption of address translation we take into account all accesses into the ITLB, DTLB, L2-TLB, PSC as well as all page walk memory references. When a TLB prefetcher is used the dynamic energy is reduced by saving demand page walks due to PQ hits but it is also increased by the accesses in the PQ, the Sampler, the FDT, and the triggered references to memory hierarchy for prefetch page walks. Figure 15 presents the dynamic energy consumed by address translation when ATP coupled with SBFP and the state-of-the-art TLB prefetchers operate in the system. ATP with SBFP lowers dynamic energy by 24%, 14.6%, and 1% for the QMM, SPEC, and BD workloads, respectively. SP, DP, and ASP increase the dynamic energy usage, especially for the BD workloads. We remark this behavior because our proposal saves demand page walks by hitting in the PQ and also decreases the number of prefetch page walks by leveraging the SBFP module. Regarding the static energy consumption, negligible results are observed.

C. Comparison with Other Approaches

Figure 16 compares ATP coupled with SBFP against other state-of-the-art techniques that improve TLB performance.

ISO Storage. We compare our proposal against a system without TLB prefetching that, for fairness, has an enlarged TLB. Specifically, the TLB is augmented with 265 entries without affecting its access time, matching the storage of ATP plus SBFP (1.68KB + 0.31KB). Figure 16 shows that ATP with SBFP outperforms this scenario by 14.7%, 9.8%, and 11.5% for the QMM, SPEC, and BD workloads, respectively.

Free prefetching into the TLB. Prior work [26] leverages PTE locality to fetch all free PTEs directly into the TLB on demand page walks. This approach does not use TLB prefetchers nor PQs, and it does not consider selecting only the useful free PTEs. Figure 16 shows that this approach (FP-TLB) reduces performance by 10.2% and 7.8% for the QMM and SPEC workloads, respectively, due to the eviction of useful PTEs from the TLB. Our results are consistent with prior work [22], [23] stating that placing all the free PTEs directly to TLB may pollute its content. There is no previous work that stores prefetches directly into the TLB using big data workloads. Our evaluation reveals that placing all free PTEs into the TLB increases performance by 5.2% for the BD workloads. These workloads experience massive TLB MPKI rates and thrash TLB, thus storing useful free PTEs into the TLB improves performance. Still, ATP coupled with SBFP outperforms this scenario. We observe similar behavior when our proposal places all the prefetches directly into the TLB.

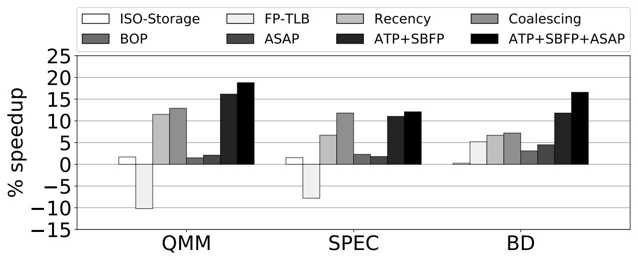

Fig. 16: Performance comparison with other approaches.

Recency-based TLB Preloading [24]. This is a software approach that modifies the page table so that each PTE stores the virtual page that is subsequently accessed. A fundamentally similar approach that only requires microarchitectural modifications is Markov prefetchers [47], that consist of a prediction table indexed with the virtual page where each entry contains a virtual page that is predicted to be accessed next [23]. To approximate the behavior of Recency-based TLB Preloading, we enhance a Markov prefetcher with a 64K-entry prediction table. Figure 16 reveals that our proposal outperforms this approach by 4.7%, 4.4%, 4.3% for the QMM, SPEC, and BD workloads, respectively. Finally, this approach requires very large hardware budget, infeasible for a realistic design.

TLB Coalescing. Coalescing approaches [12], [13], [48] rely on the contiguity of both virtual and physical memory and provide limited benefits when contiguity is absent (*e.g.,* due to fragmentation). Contrarily, SBFP exploits virtual address space contiguity and ATP relies only on the memory access patterns of the application. Therefore, our proposal is orthogonal to TLB coalescing. We compare ATP coupled with SBFP against a scenario with perfect contiguity in the virtual and physical memory where each TLB entry stores 8 adjacent PTEs. Figure 16 shows that this scenario delivers great performance gains since it increases the TLB reach. Still, our proposal outperforms this scenario for the QMM and BD workloads, while the difference for the SPEC workloads is negligible.

Cache Prefetching. Cache prefetchers typically try to learn strides within 4KB physical pages [49], [50]. Hence, the number of observed strides is limited. TLB prefetchers try to capture varying stride patterns since a prefetched page might be far from the page that triggered the TLB miss. Thus, using data cache prefetchers for TLB prefetching intuitively limits the TLB miss patterns that can be captured. To justify this observation, we convert Best-Offset-Prefetcher (BOP) [49], a state-of-the-art data cache prefetcher, to prefetch for the TLB miss stream. We select BOP because it bears some similarity to ATP coupled with SBFP as they both try to identify the most useful strides per execution phase. We enrich the set of deltas that BOP uses with negative ones (the original version of BOP uses only positive deltas) to make sure that we do not underestimate its potential for TLB prefetching.

Figure 16 shows that when BOP is used as a TLB prefetcher, it improves performance by 2.3%, 1.5%, and 3.1% for the QMM, SPEC, and BD workloads, respectively. ATP with SBFP significantly outperforms BOP for all considered benchmark suites because BOP examines the effectiveness of a predefined set of deltas, thus it is unable to capture the varying stride TLB miss patterns, while our proposal captures more generic patterns; ATP activates the right TLB prefetcher per TLB miss and SBFP selects the most useful free PTEs per page walk. Moreover, BOP checks one offset per learning round, thus it requires several rounds to gain confidence for prefetching. In contrast, our proposal identifies faster the useful offsets as ATP leverages the operation of the Fake Prefetch Queues to enable the most appropriate TLB prefetcher, and SBFP learns the usefulness of all free PTEs concurrently. Finally, our proposal is more aggressive than BOP; ATP enables one of its constituent prefetchers per TLB miss, and SBFP uses all offsets that exceed a confidence threshold while BOP uses only the offset with the highest score.

Prefetched Address Translation (ASAP) [25]. ASAP is a hardware scheme that lowers the page walk latency using direct indexing to prefetch deeper levels of the radix tree page table. Figure 16 shows that ASAP improves performance by 2.1%, 1.8%, and 4.5% for the QMM, SPEC, and BD workloads, respectively. ASAP provides important benefits when the PSCs display low hit rates, but for workloads like SPEC and QMM which experience high PSC hit rates, its potential is limited. For the BD workloads, which have lower PSC hit rates, ASAP provides significant performance benefits.

Combining ATP, SBFP, and ASAP. TLB prefetching is orthogonal to techniques that lower page walk latency. Since ASAP lowers the latency cost of page walks, it can be also used to accelerate the prefetch page walks of ATP. Figure 16 shows that combining ATP with SBFP and ASAP improves performance by 18.8%, 12.1%, and 16.6% for the QMM, SPEC, and BD workloads, respectively. ASAP increases the speedup of our proposal since it reduces the latency cost of page walks, thus the prefetched PTEs are fetched faster in the PQ that improves the timeliness of prefetching.

D. Beyond Page Boundaries Cache Prefetching

Modern data cache prefetchers are allowed to trigger prefetches that cross page boundaries. Such prefetches first check if the translation resides in the TLB. On TLB hits, the cache prefetch proceeds, otherwise a page walk fetches the corresponding translation into the TLB. To quantify the impact of beyond page boundaries cache prefetching on TLB performance, we use the Signature Path Prefetcher (SPP) [50] in the L2 cache, which allows beyond page boundaries prefetching. Figure 17 presents the performance of (i) SPP, and (ii) ATP with SBFP and SPP. The baseline corresponds to

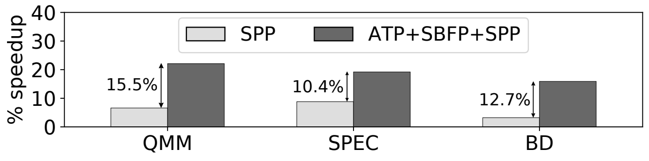

Fig. 17: Performance comparison with SPP.

a system with the IP stride L2 cache prefetcher and without TLB prefetching, similar to all previous sections. We observe that SPP improves performance for all workloads because it captures more patterns than IP stride, but it only saves a small fraction of the TLB misses. Indeed, combining SPP with ATP and SBFP significantly improves performance because, thanks to our proposal, the timeliness of cache prefetches is improved and more TLB misses are avoided. The bottom line is that ATP with SBFP provides large speedups even when a cache prefetcher that goes beyond page boundaries is used.

E. Interaction with OS Page Replacement Policy

As stated in Section VI, inaccurate TLB prefetches might harm the page replacement policy. A prefetch is harmful for the page replacement policy when it sets the access bit of the corresponding PTE, it is evicted from the PQ without providing any hit, and it does not belong to the active footprint of the application. We measure that only 1.7%, 0.9%, and 3.6% of the prefetch requests of ATP with SBFP are harmful for the page replacement policy for the QMM, SPEC, and BD workloads. Thus, we consider negligible the probability of negatively affecting the page replacement policy. We observe a small number of harmful prefetches because SBFP prefetches only the most useful free PTEs, while ATP dynamically enables the right TLB prefetcher and disables prefetching when it is not confident for issuing new prefetches. To avoid harming at all the page replacement policy, when a prefetch is proved to be useless we could trigger a page walk in the background to reset the access bit of the corresponding translation in the page table. With this solution, the number of correcting page walks would be negligible due to the small number of harmful prefetches introduced by our proposal.[3]

IX. Related Work

On the locality of the Page Table. Prior work has identified locality of the PTEs in the page table. Pham *et al.* [12], [13] exploit this locality by modifying TLB to increase its reach. These works require both virtual and physical contiguity, while SBFP solely exploits virtual contiguity. Liu *et al.* [14] use this locality to increase TLB efficiency. Bhattacharjee *et al.* [26] propose a shared last-level TLB organization for multicore systems that fetches directly into the shared TLB the PTEs 1, 2, and 3 pages away from the currently missing page. Their technique only operates at demand page walks, it does not issue TLB prefetches, and it targets multicore systems with a shared last-level TLB. Shin *et al.* [27] exploit this locality by fetching all available PTEs for GPU applications.

[3]The correcting page walks could be issued only when the TLB MSHR is not full to avoid delaying any demand or prefetch page walk.

Wang [39] exploits page table locality to fetch free PTEs into a TLB buffer only for demand page walks, but without providing any concrete implementation. Instead, we propose a practical implementation that dynamically exploits free TLB prefetching through sampling for multiple TLB prefetchers, thus we exploit page table locality for both demand and prefetch page walks, and our proposal improves performance of private per-core TLBs using CPU applications.

Other TLB prefetchers. Bhattacharjee *et al.* [22] propose two TLB prefetchers for multicore systems. The first exploits TLB misses on common virtual pages among cores and pushes TLB entries from leader to other cores. The second is based on DP [23] and exploits distance-predictable TLB misses across cores. ATP could form the base for the latter scheme.

Reducing the TLB miss latency. One way to reduce the cost of TLB misses is by improving the performance of the MMU-Caches [7], [18]. Another approach is the POM-TLB [17], a large L3 TLB stored in main memory that requires only one memory reference per page walk. DVMT [36] reduces the cost of TLB misses by allowing the application to define the appropriate page table format, so that less memory references are needed per page walk. Hashed page tables [16], [20], [51] have been proposed to resolve TLB misses faster than the radix page tables. TLB prefetching is an orthogonal approach.

Speculation. In speculation-based approaches [19], [21], [52], [53], a missing translation is predicted, the processor continues executing instructions speculatively, and a validation page walk is performed in the background. Those approaches are affected by the system state (OS, fragmentation) as they depend on allocating contiguous virtual pages to contiguous physical pages to predict the missing address translations.

Increasing TLB Reach. Processor and OS vendors provide support for large pages [54] to increase TLB reach. Prior work focuses on combining base and large pages by using a single TLB [55]–[57] or separate TLBs [11] per page size. While large pages increase TLB reach and reduce page walks, they are susceptible to performance issues when the OS cannot allocate such mappings (*e.g.,* due to fragmentation) or when the hardware support for large pages is limited with respect to the application needs. Several approaches try to bypass the limitations of large pages [6], [10], [13], [15], [48], [58]–[62]. These schemes are orthogonal to hardware TLB prefetching since they rely on explicit OS and hardware support.

X. Conclusions

This paper provides evidence that exploiting the locality in the last level of the page table for TLB prefetching purposes has the potential to provide large performance benefits. To ameliorate the address translation performance bottleneck, this paper proposes (i) *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic sampling-based scheme that identifies and prefetches only the most useful free page table entries per page walk, and (ii) *Agile TLB Prefetcher (ATP)*, a composite TLB prefetcher comprised of three low-cost TLB prefetchers while introducing selection and throttling schemes to enable the most appropriate TLB prefetcher per TLB miss, and disable TLB prefetching when required. Considering an extensive set of contemporary industrial, academic, and big data workloads, we demonstrate that combining ATP with SBFP provides significant performance enhancements while reducing the vast majority of the page walk references to the memory hierarchy.

XI. Acknowledgements

We are profoundly grateful to Abhishek Bhattacharjee, Dionisios Pnevmatikatos, Elvira Teran, and our anonymous reviewers for their constructive feedback throughout the paper drafting. This work is partially supported by the Spanish Ministry of Science and Technology through the PID2019-107255GB project, the Generalitat de Catalunya (contract 2017-SGR-1414), the NSF grants CCF-1912617 and CNS-1938064, and generous gifts from Intel Labs. 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. Lluc Alvarez has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Juan de la Cierva Formacion fellowship No. FJCI-2016-30984. Marc Casas has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Ramon y Cajal fellowship No. RYC-2017-23269.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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 Annual International Symposium on Computer Architecture*, 2013.

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

[8] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The interaction of architecture and operating system design," *SIGARCH Comput. Archit. News*, Apr. 1991.

[9] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, Jan. 1997.

[10] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," *SIGARCH Comput. Archit. News*, Jun. 2017.

[11] 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 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[12] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[13] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[14] L. Liu, "Multiple-page translation for tlb," in *Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93*, Oct 1993.

[15] 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 Annual International Symposium on Computer Architecture*, 2015.

[16] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016.

[17] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

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

[19] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[20] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[21] 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 ACM/IEEE 47th International Symposium on Computer Architecture (ISCA)*, 2020.

[22] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[23] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[24] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

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

[26] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[27] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular gpu applications," in *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture*, 2018.

[28] "Championship Value Prediction," https://www.microarch.org/cvp1/.

[29] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, Sep. 2006.

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

[31] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, 2015.

[32] "XSBench," https://github.com/ANL-CESAR/XSBench.

[33] "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.

[34] "Intel 5-Level Paging and 5-Level EPT," https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html.

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

[36] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[37] Abishek Bhattacharjee, "Advanced concepts on address translation," http://www.cs.yale.edu/homes/abhishek/abhishek-appendix-l.pdf.

[38] Abhishek Bhattacharjee, Margaret Martonosi, "Inter-core cooperative tlb prefetchers," https://patents.google.com/patent/US8880844B1/en.

[39] James Wang, Zongjian Chen, "Patent on TLB Prefetching," https://patents.google.com/patent/US20110010521.

[40] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, Jun. 2000.

[41] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, May 1995.

[42] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, Jun. 2003.

[43] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jimenez, "Chirp: Control-flow history reuse prediction," in *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*, 2020.

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

[45] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *Proceedings of the International Conference on Computer-Aided Design*, 2011.

[46] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu, "The virtual block interface: A flexible alternative to the conventional virtual memory framework," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020.

[47] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[48] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[49] P. Michaud, "Best-offset hardware prefetching," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[50] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[51] J. Huck and J. Hays, "Architectural support for translation table management in large address space machines," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[52] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[53] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[54] "Transparent Huge Pages in 2.6.38," http://lwn.net/Articles/423584/.

[55] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[56] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[57] A. Seznec, "Concurrent support of multiple page sizes on a skewed associative TLB," *IEEE Transactions on Computers*, July 2004.

[58] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1998.

[59] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *SIGOPS Oper. Syst. Rev.*, Dec. 2003.

[60] M. Swanson, L. Stoller, and J. Carter, "Increasing tlb reach using superpages backed by shadow memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[61] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in supporting two page sizes," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[62] F. Guvenilir and Y. N. Patt, "Tailored page sizes," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020.