Instruction-Aware Cooperative TLB and Cache Replacement Policies

Dimitrios Chasapis*
dimitrios.chasapis@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Georgios Vavouliotis*
gvavou5@gmail.com
Zurich, Switzerland

Daniel A. Jiménez
djimenez@acm.org
Texas A&M University
College Station, United States

Marc Casas
marc.casas@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

Modern server and data center applications are characterized not only by large datasets, but also by large instruction footprints that incur frequent cache and Translation Lookaside Buffer (TLB) misses due to instruction accesses. Instruction TLB misses are particularly problematic as they cause pipeline stalls that significantly harm performance.

This paper proposes cooperative last-level TLB (STLB) and L2 cache (L2C) replacement policies targeting workloads with large instruction footprints. We propose *Instruction Translation Prioritization (iTP)*, an STLB replacement policy that maximizes the number of instruction hits in the STLB at the expense of increasing data page walks. To compensate for the increase in data page walks, we propose *extended Page Table Prioritization* (*xPTP*), a new L2C replacement policy that amplifies the benefits of *iTP* by effectively reducing L2C misses due to data page walks. Our proposal, *iTP+xPTP*, combines *iTP* at STLB and *xPTP* at L2C. *iTP+xPTP* employs an adaptive mechanism that switches between *xPTP* and *LRU* policies at L2C based on the pressure placed on the virtual memory subsystem. Our proposal, *iTP+xPTP*, improves single-core geometric mean performance by 18.9% over a baseline that uses the *LRU* replacement policy at both STLB and L2C across a set of contemporary server workloads. Under SMT co-location, the corresponding performance uplift is 11.4%. Finally, we show that *iTP+xPTP* outperforms the state-of-the-art STLB and cache replacement policies.

CCS Concepts: • **Software and its engineering → Virtual memory**; • **Applied computing → Data centers**.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

Keywords: virtual memory, address translation, translation lookaside buffer, TLB management, replacement policy

ACM Reference Format:
Dimitrios Chasapis, Georgios Vavouliotis, Daniel A. Jiménez, and Marc Casas. 2025. Instruction-Aware Cooperative TLB and Cache Replacement Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3669940.3707247

1 Introduction

Virtual memory based on paging is prevalent in contemporary computing systems. Each memory access in page-based virtual memory systems requires translating a virtual address to a physical address. Translation Lookaside Buffers (TLBs) mitigate address translation overheads by storing the recently used virtual-to-physical mappings. Recent literature shows that frequent last-level TLB (STLB) misses incur significant performance and energy overheads [13, 15, 18, 25, 39, 47]. Prior work that reduces address translation overheads can be broadly classified in two categories: i) techniques that reduce the number of STLB misses [16, 17, 36, 37, 62, 65, 66, 69, 70, 81, 87] and ii) techniques that mitigate the latency cost of page walks [42, 53].

Modern server and data center applications are characterized not only by large datasets but also by large instruction footprints [14, 39, 61, 88]. These large footprints incur frequent cache and STLB misses due to instruction accesses, which constitute a major performance painpoint because they cause frequent pipeline stalls. This problem is likely to be exacerbated in the future since the instruction footprint of server applications increases yearly by up to 30% [39, 52, 74]. To make matters worse, frequent STLB misses (for both data and instruction accesses) also increase cache pressure since

*Both authors contributed equally to this research work.

they trigger page walks that insert in the cache hierarchy memory blocks containing page table entries.

A fundamental aspect of STLB performance is its replacement policy. Processor vendors typically implement simple replacement policies for the STLB (*e.g.*, *LRU* variations [10, 66, 70]). To improve TLB performance for address translation bound applications, previous work [55] proposes a predictive replacement policy for the STLB. However, none of the previously proposed STLB replacement policies specifically targets reducing instruction STLB misses, which can potentially stall the pipeline and are more harmful for performance than data STLB misses whose latency can be partially hidden in out-of-order cores. Although neglecting STLB instruction misses does not cause performance degradation for desktop and High-Performance Computing (HPC) workloads, which have small instruction footprints that typically fit in the first-level TLB, it leaves on the table many opportunities for improving the performance of big-code workloads.

When it comes to replacement policies for the cache hierarchy, prior work is classified in two categories: translation-oblivious replacement policies [21, 23, 32, 35, 63, 71, 72, 77, 79] and translation-aware replacement policies [63, 79]. The fundamental difference between these two categories is that the latter differentiate their replacement decisions between cache blocks storing instruction/data payload and cache blocks storing Page Table Entries (PTEs). Prior translation-aware cache replacement policies have two limitations: (i) they do not distinguish between cache blocks accommodating instruction PTEs and data PTEs and (ii) they do not work synergistically with the STLB replacement policy.

This paper proposes cooperative STLB and cache replacement policies that target workloads with large instruction footprints. For STLB, we propose the *Instruction Translation Prioritization (iTP)* replacement policy. The design of *iTP* is based on two observations: (i) workloads with large instruction footprints incur large instruction translation overheads, and (ii) smartly prioritizing instruction translations over data translations in the STLB has the potential to mitigate these overheads. *iTP* maximizes the number of instruction hits in the STLB at the expense of increasing data page walks. To compensate for the increase in data page walks, we propose the *extended Page Table Prioritization* (*xPTP*), a new L2C replacement policy that amplifies the benefits of *iTP*. *xPTP* is built on the observation that the number of cache misses triggered by data page walks may increase when using an STLB replacement policy that prioritizes instruction entries over data entries like *iTP*. In this context, *xPTP* effectively reduces cache misses due to data page walk references and, therefore, maximizes the benefits of *iTP*. Our proposal, *iTP* combined with *xPTP* (*iTP+xPTP*) uses an adaptive scheme that disables *xPTP* during phases with low STLB pressure, ensuring high performance across different execution phases.

In summary, this paper makes the following contributions:

- This is the first study to show that prioritizing instructions over data in the STLB has the potential to provide performance gains for server applications with large code footprints. We show that doing so increases the cache pressure due to page walks for data references.
- We propose two cooperative replacement policies, *iTP* and *xPTP*, to drive the STLB and L2C replacement policies, respectively. Both *iTP* and *xPTP* are motivated by an in-depth analysis of the TLB and cache behavior of server workloads with large code footprints. *iTP* prioritizes instruction translations over data translations in the STLB at the cost of increasing data page walks. *xPTP* compensates for the increase in data page walks by improving the locality of data PTEs in the caches.
- We compare *iTP* and *iTP+xPTP* with state-of-the-art translation-aware cache replacement policies (TDRRIP [79], PTP [63]) and the state-of-the STLB replacement policy (CHiRP [55]) across 120 single- and 75 two-hardware thread server workloads. *iTP* and *iTP+xPTP* improve geomean performance by 2.2%, 18.9% and 0.3%, 11.4% with respect to *LRU* (for both STLB and L2C) for the single- and two-hardware thread workloads, respectively. We show that *iTP+xPTP* outperforms the state-of-the-art cache and STLB replacement policies in both isolation and combination.
- We show that our proposals, *iTP* and *iTP+xPTP*, provide consistent performance gains when different well-established LLC replacement policies (*e.g.*, *SHiP* [84], *Mockingjay* [71]) are used.

2 Background

2.1 Virtual Memory Subsystem

Each memory access on page-based virtual memory systems requires a virtual-to-physical address translation. To reduce the address translation costs, modern systems combine software and hardware approaches. The page table is a structure containing virtual-to-physical translations of all pages loaded to memory and it is typically implemented as a multi-level radix tree structure [4]. The TLB is a structure that stores the most recently used virtual-to-physical translations to reduce the latency and energy costs of frequent page walks. CPU vendors typically implement multi-level TLBs with small instruction and data first level TLBs (ITLB and DTLB) and a large last-level TLB (STLB) [10]. ITLB and DTLB cache instruction and data Page Table Entries (PTEs), respectively, while the STLB accommodates both data and instruction PTEs within the same structure. Thus, workloads with large data and code footprints (*e.g.*, server and datacenter applications) frequently contend for STLB capacity [80].

For each memory access, which can be either for instruction or data, the appropriate first-level TLB is looked up and, upon a miss, the STLB is accessed. STLB misses imply that

Section 6.6 evaluates both unified and split STLB designs.

the hardware page table walker traverses the page table to find the required translation. Frequent STLB misses negatively impact performance since page walks might require multiple accesses to the memory hierarchy to find the requested translation [12, 15, 19, 25, 39, 47, 58, 61, 81]. MMU caches [15, 16, 64] significantly reduce the page walk memory references (and the latency cost of page walks) by storing entries from the intermediate levels of the radix tree page table. Finally, PTEs containing intermediate or leaf page table levels are stored in the cache hierarchy. These PTEs may contain data related to virtual-to-physical translations of either instruction or data memory accesses [10].

2.2 Cache Replacement Policies

Previously proposed cache replacement policies can be broadly classified in two categories. The first category consists of *translation-oblivious cache replacement policies*, *i.e.*, policies that do not differentiate between cache blocks accommodating instructions or data payload, and blocks accommodating PTEs. The translation-oblivious replacement policies either consider recency to guide replacement [26, 34, 50, 59, 67, 75, 83], or drive replacement decisions based on previously observed patterns [21, 23, 28, 32, 35, 43, 44, 46, 51, 71, 72, 77, 85]. *Mockingjay* [71] is the state-of-the-art translation-oblivious replacement policy and uses patterns within long histories to accurately predict reuse distances.

The second category contains *translation-aware cache replacement policies*, *i.e.*, policies that distinguish between blocks containing data or instructions payload, and blocks containing PTEs. The most recent translation-aware replacement policies are i) *T-DRRIP+T-SHiP* [79], which prioritizes keeping cache blocks that contain PTEs and favors evicting blocks brought by demand loads that missed in the STLB and ii) *Page Table Prioritization (PTP)* [63], which prioritizes keeping blocks containing PTEs in both L2C and LLC. Note that *T-DRRIP+T-SHiP* and *PTP* do not differentiate between cache blocks containing instruction PTEs and data PTEs.

2.3 TLB Replacement Policies

Processor vendors typically implement simple replacement policies for the TLB hierarchy (*e.g.*, random for first-level TLBs, and LRU variants for STLB [10, 66, 70]). To improve TLB performance for applications that put high pressure on the virtual memory subsystem, previous work [55] proposes CHiRP, a table-based replacement policy for STLBs. CHiRP requires augmenting each STLB entry with metadata bits to ensure the correct update of its prediction table. Upon STLB misses, CHiRP generates a signature for indexing the prediction table. If the corresponding confidence counter exceeds a predefined threshold, CHiRP predicts that the translation will be reused soon and drives the STLB insertion policy accordingly. Notably, CHiRP does not distinguish between data and instruction PTEs residing in the same STLB structure.

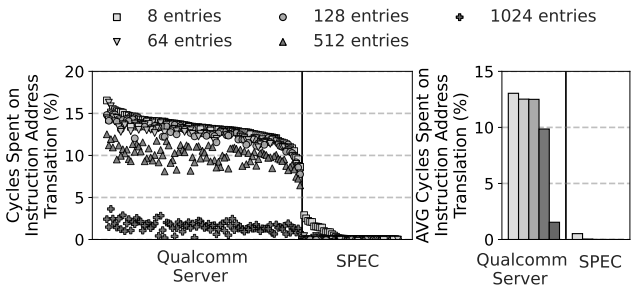


Figure 1. Address translation overhead as a function of different ITLB sizes across *SPEC CPU 2006*, *SPEC CPU 2017* and *Qualcomm Server* workloads.

3 Motivation

This section motivates the need for synergistic STLB and cache replacement policies to accelerate server workloads with large instruction and data footprints. Section 3.1 shows that the large code sizes of server applications cause significant STLB pressure, increasing the cost of instruction address translation. Section 3.2 demonstrates the potential benefits of STLB replacement policies that prioritize instruction translations over data translations when dealing with workloads with large instruction footprints. Section 3.3 highlights the need to consider the impact of instruction-aware STLB replacement policies (Section 3.2) on the locality of data address translations stored in the cache hierarchy.

3.1 Instruction Address Translation Cost

Contemporary server applications have large code footprints that reach millions of instructions [14, 40, 41, 60, 61, 80] as opposed to desktop and HPC applications that have small (tens of KBs) instruction footprints [61, 80]. This variety of instruction footprints across different application domains results in different behaviors at the STLB level, which is typically shared between instruction and data translations.

To quantify the impact of the code size on the TLB behavior and the overall performance of a system across different application types, Figure 1 presents the total CPU cycles spent on instruction address translation as a function of different first-level instruction TLB (ITLB) sizes when considering the SPEC CPU 2006 and SPEC CPU 2017 (*SPEC*) [7, 8] benchmarks and the *Qualcomm Server* workloads used in the CVP-1 [1] and IPC-1 [9] contests. Section 5 describes in detail the experimental setup we use to generate these data.

Figure 1 reveals that increasing the number of ITLB entries reduces the instruction address translation overheads. Focusing on a realistic ITLB sizes (*e.g.*, 64 or 128 entries), we measure that *Qualcomm Server* and *SPEC* workloads spend on average 12.52% and 0.03% of their total execution cycles serving instruction address translation, respectively. We observe such behavior because *SPEC* workloads have small code footprints that generally fit in reasonable-sized ITLBs

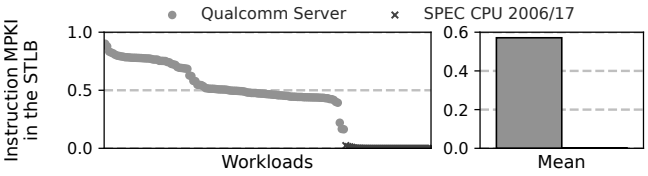


Figure 2. STLB MPKI for instruction references of the*SPEC 2006*, *SPEC 2017*, and *Qualcomm Server* workloads.

[80] while *Qualcomm Server* workloads spend a large fraction of their cycles serving instruction address translation due to their large code footprints; Figure 1 also shows that an ITLB of more than 1024 entries is needed to minimize the instruction address translation overheads of the *Qualcomm Server* workloads. The main takeaway of this study is that server workloads incur significantly higher instruction translation costs than HPC and desktop workloads.

We complement the results of Figure 1 by quantifying the STLB MPKI due to instruction accesses across the same set of workloads used in Figure 1. This study considers a 64-entry ITLB and a 1536-entry STLB (Section 5). Figure 2 shows that the STLB MPKI for instructions is negligible for the *SPEC* workloads since most of the instruction translations fit in the ITLB, as indicated while explaining the results of Figure 1. In contrast, the STLB MPKI for instruction references is up to 0.9 for the *Qualcomm Server* workloads, revealing that these applications trigger a significant number of page walks for instruction accesses, explaining why these applications face significant instruction address translation overheads.

Finding 1. *Applications with large code footprints amplify the address translation overheads.*

3.2 Prioritizing Instruction Translations in the STLB

Despite the substantial pressure on the virtual memory subsystem stemming from the large instruction footprints of server workloads, which will worsen significantly in the coming years [40, 52, 61, 74], fundamental TLB design concepts have been conceived taking into account the properties of desktop and HPC workloads and thus neglect the impact of instruction address translation on the system performance. As a result, current STLB replacement policies do not differentiate between data and instruction entries and they use access recency to drive replacement decisions.

This section motivates the potential benefits of (i) making the STLB replacement policies instruction-aware and (ii) prioritizing instruction translations over data translations in the STLB because instruction references are on the critical path of the pipeline execution [61, 80], thus their latency cost can be sometimes (partially) hidden.

To illustrate the potential improvements of prioritizing instructions over data translation entries in the STLB, we consider several modified versions of the *LRU* replacement policy.[3] Specifically, these modified replacement policies decide to evict an instruction STLB entry depending on a certain probability P. For example, if $P = 0.8$, the replacement policy will decide to evict a data translation entry for 80% of the evictions and an instruction entry for the other 20%. If there are several data (instruction) translation entries in the STLB, the modified *LRU* will evict the least recently used data (instruction) translation. If only data (instruction) entries are present in the STLB, the least recently used data (instruction) translation will get evicted, independently from the decision made based on the probability P.

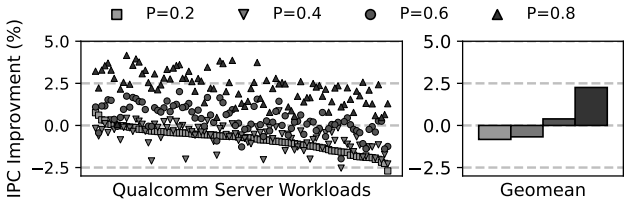


Figure 3. IPC improvement when the STLB replacement policy prioritizes instruction translations over data translations by a certain probability P.

Figure 3 presents the performance comparison between the different modified LRU policies that consider a probability P to decide whether to victimize a data or an instruction translation entry in the STLB over a baseline that considers LRU as STLB replacement policy. Section 5 describes the methodology we use to generate the data shown in Figure 3.

We observe that the highest probabilities for keeping instruction translations in the STLB provide performance gains over standard LRU. The opposite behavior is observed for the lower probabilities since keeping data over instruction translations in the STLB incurs performance degradation. Regarding the *SPEC* workloads, our experiments indicate that the modified LRU policies perform similar to standard LRU since the instruction footprint of *SPEC* workloads fits in the ITLB, thus STLB accommodates mostly data translations.

Figure 3 reveals that trading instruction for data entries in the STLB brings performance benefits for applications with large instruction footprints that exacerbate STLB pressure, as it is the case for the *Qualcomm Server* workloads, while applying the opposite strategy results in performance slowdowns. This happens because instruction references are on the critical path of pipeline execution, potentially causing pipeline stalls while data references partially hide their latency costs thanks to out-of-order execution [61, 80].

Finding 2. *Prioritizing instructions over data in the STLB can provide significant performance enhancements for workloads with large instruction footprints.*

Finding 2 motivates our STLB replacement policy, presented in Section 4.1, that selectively trades instructions for data in the STLB for big-code applications.

[3] Processor vendors typically use an LRU variant as STLB replacement policy [10, 66, 70], as explained in Section 2.3.

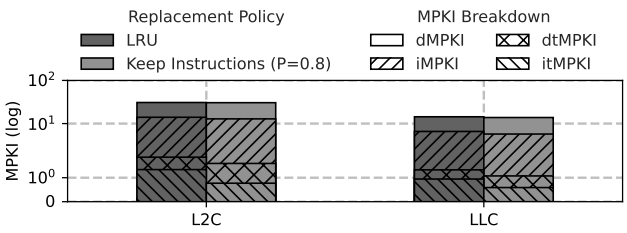


Figure 4. MPKI breakdown across all cache levels comparing *LRU* and an *LRU* variant that favours keeping instruction translations at the STLB (Keep Instructions ($P = 0.8$)).

3.3 Impact on the Cache Hierarchy of Prioritizing Instructions in the STLB

Section 3.2 demonstrates that prioritizing instructions over data in the STLB can provide great performance benefits. However, doing so comes at the cost of increasing the number of data page walks that would impact the locality of PTEs stored in the cache hierarchy since page walks are served via memory accesses. This section quantifies the impact of prioritizing instructions over data in the STLB on the locality of the different cache levels.

Figure 4 shows the MPKI breakdown for the L2C and LLC when the STLB uses the *LRU* replacement policy and the modified *LRU* replacement policy that favors keeping instructions translations with $P = 0.8$ probability across all *Qualcomm Server* workloads. The MPKI breakdown consists of four categories: (i) MPKI due to data accesses (*dMPKI*), (ii) MPKI due to instruction accesses (*iMPKI*), (iii) MPKI due to page walks triggered by data STLB misses (*dtMPKI*), *i.e.*, memory requests looking for PTEs that contain data translations, and (iv) MPKI due to instruction STLB misses (*itMPKI*). Figure 4 indicates that using an STLB replacement policy that prioritizes instructions over data, increases the number of misses due to page walk references triggered by data STLB misses (*dtMPKI*) with respect to *LRU*. This MPKI increase in data page walk references has a negative impact on the latency cost of data STLB misses.

> ***Finding 3.*** *Prioritizing instructions over data in the STLB increases the cache misses triggered by page walks serving data STLB misses.*

Finding 3 motivates the cache replacement policy that we propose in Section 4.2, which works cooperatively with the new STLB replacement policy that we present in Section 4.1 and provides significant benefits when the cache hierarchy faces a heavy load due to frequent data page walks.

4 Instruction-aware Cooperative Cache/STLB Replacement Policies

This section proposes new replacement policies for the STLB and the cache hierarchy that cooperatively accelerate server applications. Section 4.1 presents the *Instruction Translation Prioritization* (*iTP*), an instruction-aware STLB replacement policy. Section 4.2 presents the *extended Page Table Prioritization* (*xPTP*), an L2C replacement policy that amplifies the benefits of *iTP*. Section 4.3 describes the operation of a system that incorporates both *iTP* and *xPTP* policies.

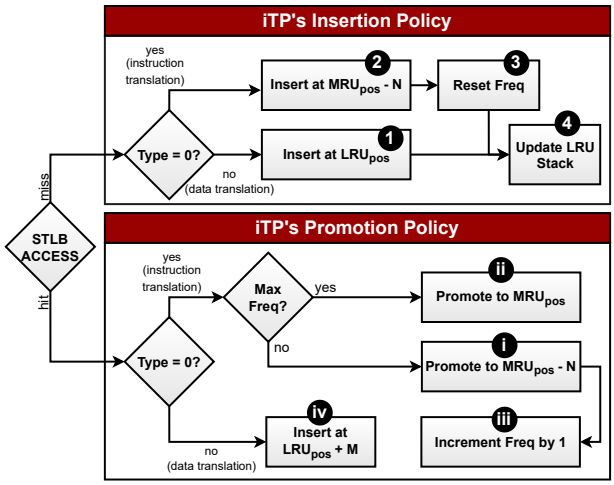


Figure 5. Insertion (up) and promotion (down) policies of *iTP*. Variables N and M represent distances from the top and the bottom of the recency stack, respectively.

4.1 Instruction Translation Prioritization (iTP)

This section proposes the *Instruction Translation Prioritization* (*iTP*), a new STLB replacement policy. *iTP* is motivated by Finding 1 and Finding 2, presented in Sections 3.1 and 3.2, respectively. These two findings reveal that workloads with large instruction footprints suffer from remarkable instruction translation costs, and that prioritizing instruction translations over data translations in the STLB has the potential to mitigate these costs. To address our analysis findings, the *iTP* policy maximizes the number of instruction hits in the STLB at the expense of increasing data STLB misses. The goal of *iTP* is not to reduce the overall STLB MPKI but selectively trade data translations for instruction translations to save critical for performance instruction page walks.

iTP uses smart insertion and promotion policies that favor keeping instruction translation PTEs at the top positions of the STLB recency stack, which we assume to use the *LRU* policy without loss of generality since vendors typically use an LRU variant as STLB replacement policy (Section 2.3). The top of the *LRU* recency stack corresponds to the most recently used entry while the bottom position of this stack corresponds to the least recently used entry. We refer to these positions as MRU_{pos} and LRU_{pos}, respectively. *iTP* uses the same eviction policy as *LRU*-based policies, *i.e.*, it evicts the entry located at the LRU_{pos} position. However, *iTP* uses insertion and promotion policies that consider whether STLB entries contain instruction or data translations, which is a fundamental difference from other STLB replacement

policies like *LRU* and *CHiRP* [55]. The implementation of *iTP* requires two additional metadata fields per STLB entry. The first field contains one bit annotating whether an entry stores a data translation or an instruction translation; we refer to this field as `Type`. The second field contains a frequency counter that optimizes the insertion and promotion policies of *iTP*; we refer to this field as `Freq`.

Figure 5 shows the insertion and promotion policies of *iTP* with a flowchart. The former takes place at the end of a page walk when the requested translation needs to be inserted in the STLB. The latter takes place upon STLB hits and moves the hit entry to the appropriate position in the recency stack.

4.1.1 iTP Insertion Policy. *iTP* inserts the new entry in the STLB by considering whether it contains a data or an instruction PTE. *iTP* inserts data translation entries (`Type=1`) at the LRU_{pos} position ❶, thus this new entry has the highest priority for eviction. If it is an instruction translation entry, *iTP* inserts it at a high position in the recency stack, but not at the MRU_{pos} position. Instead, *iTP* places the new instruction entry N positions below MRU_{pos} ❷ in the recency stack. The rationale behind this decision is that the MRU_{pos} is reserved for instruction translation entries that are frequently referenced. An entry can only reach the MRU_{pos} position when its `Freq` counter is saturated. For new instruction translation entries, the `Freq` counter is set to 0 ❸. Once the new entry is inserted, *iTP* updates the recency stack of all the other entries by moving them down one position. This update takes place for the insertion of both instruction or data translations ❹. Thus, useless instruction translation entries can reach the LRU_{pos} if they are not frequently accessed and are eventually evicted from the STLB.

4.1.2 iTP Promotion Policy. *iTP* applies different promotion policies for entries containing data or instruction translations. If the hit entry contains an instruction translation (`Type=0`), *iTP* promotes it by considering the value of its `Freq` counter. If this counter is not saturated, *iTP* moves the hit entry to the same position as if it was a new instruction translation entry, *i.e.*, $MRU_{pos} - N$ ❶. If the `Freq` counter is saturated, the entry is moved to MRU_{pos} ❷. The intuition behind this mechanism is that instruction translation entries that experience high access frequency rates are more likely to be useful in the future, thus *iTP* promotes them to the MRU_{pos} position. *iTP* increments the frequency of the hit entry if it is not saturated ❸. If the entry that produced the hit contains a data translation, it is promoted M positions higher than LRU_{pos} ❹. The new position can be expressed with the following formula: $e_{pos} = LRU_{pos} + M$, where e_{pos} is the new entry's position and M is an integer number smaller than the STLB associativity and larger than N.

4.1.3 *iTP's* Storage and Latency Overheads. The proposed implementation of *iTP* requires 4 additional bits to be stored per STLB entry: one bit for the `Type` field and 3 bits for the `Freq` field. For a 1536-entry STLB [10], *iTP* requires 768 bytes of extra storage. *iTP* does not increase the STLB access latency with respect to *LRU* for STLBs with 1536 entries or more. Regarding smaller STLBs, *iTP* may increase the STLB access latency. Approaches keeping the `Type` and `Freq` counters in a small hardware structure decoupled from the STLB can reduce the STLB access latency with small energy overheads. Finally, *iTP* requires one additional bit per STLB MSHR entry to store the `Type` of the corresponding miss.

Having a 3-bit counter per STLB entry to store the `Freq` field constitutes a similar overhead as other well-established replacement approaches. For example, implementing a LRU replacement for a N-set associative cache would require $N \cdot log(N)$ bits per set. Since *iTP* requires $3 \cdot N$ bits per set, LRU incurs the same overhead or more than having 3 bits per entry when $N \geq 8$. Tree-base pseudo-LRU policies require $O(n)$ bits per set for a N-set associative cache, similar to *iTP*.

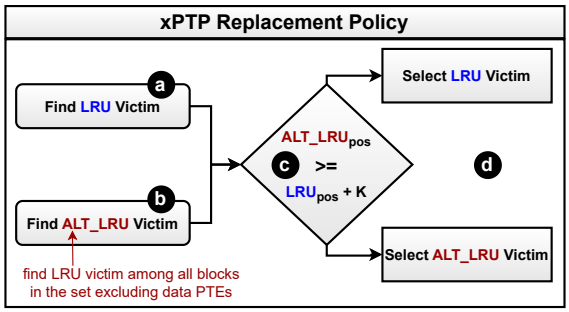


Figure 6. Flowchart diagram depicting the *xPTP* replacement policy. ALT_LRU victim refers to the block with the lowest LRU value excluding blocks that accommodate data PTEs.

4.2 Extended Page Table Prioritization (xPTP)

The *extended Page Table Prioritization* (*xPTP*) is an L2C replacement policy that amplifies the benefits of *iTP*. The design of *xPTP* is motivated by Finding 3 of Section 3.3, showing that when the STLB uses a replacement policy that prioritizes instruction entries over data entries (*e.g.*, *iTP* or the one evaluated in Figure 4), there is a notable increase in cache misses triggered by data page walks. *xPTP* effectively mitigates the performance impact of memory accesses triggered by page walks coming from data STLB misses. To do so, *xPTP* requires an additional bit per L2C block to indicate whether the block contains a data PTE or not. We refer to this bit as `Type`, similarly to the one used for *iTP*. The insertion and promotion policies of *xPTP* follow the *LRU* approach. The only modification is that *xPTP* requires setting the `Type` bit on insertion. The novel aspect of *xPTP* is its eviction policy that favors keeping cache blocks containing data PTEs in the cache hierarchy to accelerate data page walks and eventually reduce the overheads of *iTP* that increases data STLB misses.

Figure 6 shows the *xPTP* eviction policy (assuming an L2C miss). The initial step ⓐ identifies a potential victim block by looking at the bottom of the recency stack. In parallel,

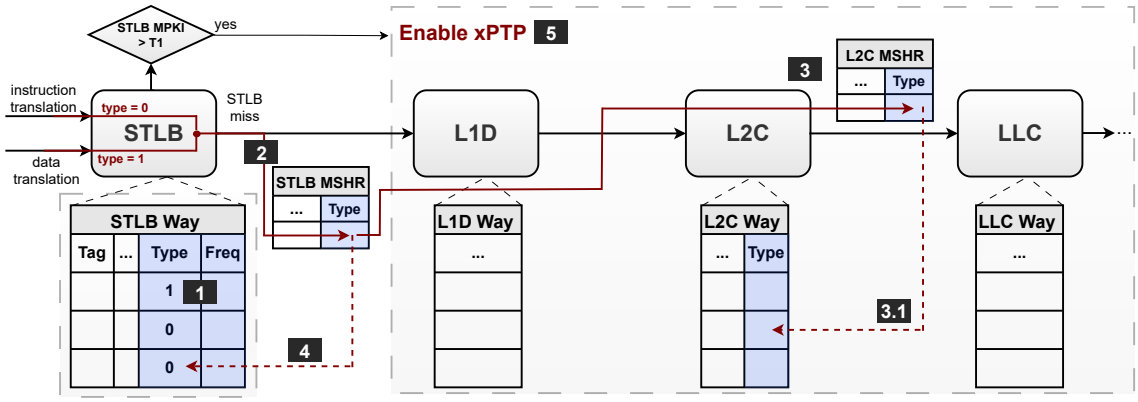


Figure 7. Operation of a system that implements both *iTP* and *xPTP* replacement policies.

xPTP identifies an alternative victim which is the cache block closest to the recency stack not accommodating a data PTE ⓑ. We refer to this position as ALT_VICTIM_{pos}. Step ⓒ determines which cache block will be evicted by evaluating the inequality $ALT\_VICTIM_{pos} \geq LRU_{pos} + K$, where K is an integer smaller than the cache associativity that indicates below which position in the recency stack we consider an entry to be a good candidate for eviction. The final step ⓓ returns the position of the victim; LRU_{pos} if the inequality holds or ALT_VICTIM_{pos} otherwise.

The implementation of *xPTP* requires one additional bit per L2C block and L2C MSHR entry to store the Type information, which constitutes a negligible area overhead.

4.3 Combining *iTP* with *xPTP* (*iTP+xPTP*)

This section describes *iTP+xPTP*, a cooperative scheme that uses *iTP* as STLB replacement policy and *xPTP* as L2C replacement policy. Figure 7 shows the operation of *iTP+xPTP* in steps and the microarchitectural modifications required to support its operation. First, *iTP* implies 4 additional bits per STLB entry; 3 bits for the frequency counter (Freq) and 1 bit for the Type that specifies whether an STLB entry stores a data or instruction translation, as Section 4.1 indicates. Second, *xPTP* requires one additional Type bit per entry in the L2C and L2C MSHR that specifies whether or not the corresponding entry contains a data PTE or not.

An address translation request can either correspond to an instruction fetch or a data access. The former/latter searches the ITLB/DTLB for the requested translation. Upon DTLB or ITLB misses, the STLB is looked up for possible hits. If the STLB access is a hit, *iTP* applies its promotion policy (Section 4.1.2) that takes into account the Type of the hit entry to accordingly update the corresponding Freq value 1. Then, the processor replays the request. Upon STLB misses, an STLB MSHR entry is primarily allocated. *iTP* augments each STLB MSHR entry with one bit to annotate the Type of the miss 2; Type is 0 (1) for instruction (data) STLB misses 2. Then, the hardware page table walker is activated to fetch the requested translation (for either data or instruction) from the page table, potentially triggering multiple references to the memory hierarchy (L2C, LLC, DRAM), as explained in Section 2.1. For each page walk reference that misses in the L2C, *xPTP* stores the Type bit in the allocated L2C MSHR entry 3; Type is set to 0 (1) for page walk references serving instruction (data) translation requests. Once a page walk reference that misses in the L2C is served 3.1, the Type bit is written back to the corresponding L2C block. The Type bit is set to one for blocks accommodating data address translation entries. Following this operation, *xPTP* leverages the Type bit in the L2C to drive its replacement decisions, as explained in Section 4.2. Finally, at the end of the page walk the requested translation needs to be inserted into the STLB coupled with the corresponding Type bit that is stored in the STLB MSHR 4. At this point, *iTP* takes as input the Type bit to drive the insertion of the new entry, as explained in Section 4.1.1. We refer to the design that combines *iTP* with *xPTP* as *iTP+xPTP*.

4.3.1 Phase Adaptability.

xPTP accelerates data page walks by favoring the placement of data PTEs in the L2C. However, our analysis indicates that *xPTP* is beneficial for performance during phases with high STLB pressure while it may harm performance for phases with low STLB pressure due to favoring data PTEs in the L2C. To address this issue, we enhance *iTP+xPTP* with a dynamic mechanism that enables *xPTP* during phases with high STLB pressure while disabling *xPTP* for workloads with moderate memory footprints that do not stress the TLB hierarchy.

We propose a mechanism that monitors the STLB MPKI rates and, if the STLB MPKI surpasses a threshold (T1 in Figure 7), it enables *xPTP* (step 5 in Figure 7). Otherwise, standard LRU policy is used for the current access. Note that *xPTP* degenerates to LRU if steps ⓐ, ⓑ, ⓒ, and ⓓ of Figure 6 are omitted, thus there is no need to have a separate implementation of the LRU policy. This selection scheme can be implemented with two counters an a 1-bit status register. The first counter accounts for the STLB misses, the second counter accounts for the number of dynamic instructions executed, and the 1-bit status register specifies which cache

Table 1. System configuration.

CPU Core	4GHz, 352-entry ROB
Fetch Target Queue (FTQ)	128-entry
L1 ITLB	64-entry, 4-way, 1-cycle, 8-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle, 8-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle, 16-entry MSHR, 1 page walk / cycle **iTP: 3-bit Freq counter, 1-bit Type, N=4, M=8**
Page Structure Caches	5-level Split PSC, 2-cycle. PSCL5: 2-entry, fully; PSCL4: 4-entry, fully PSCL3: 8-entry, 2-way; PSCL2: 32-entry, 4-way.
L1I Cache	32KB, 8-way, 4-cycle, 8-entry MSHR, VIPT, FDiP
L1D Cache	32KB, 12-way, 5-cycle, 8-entry MSHR, VIPT, Next-Line prefetcher
L2 Cache	512KB, 8-way, 5-cycle, 32-entry MSHR, PIPT, Stride prefetcher, **xPTP: 1-bit Type, K=8**
LLC (per core)	2MB, 16-way, 10-cycle, 64-entry MSHR, PIPT
DRAM	tRP=tRCD=tCAS=12, 12.8 GB/s
Branch Predictor	hashed perceptron [76]

replacement policy is used for the L2C, either *xPTP* or *LRU*. Initially, the two counters are set to zero. Once the counter counting the executed dynamic instructions reaches 1000, the misses counter is compared to T_1, and the 1-bit status register shifts to *xPTP* if the miss count exceeds T_1. Then, both counters are set to zero and the process starts again. Previous work indicates the feasibility of mechanisms driving the cache replacement policies based on simple counters [63].

5 Experimental Setup

5.1 Evaluation Methodology

To evaluate our proposals, we use the ChampSim simulator [2], a detailed trace-based simulator that models an out-of-order processor and a three-level cache hierarchy with 64B cache blocks. Our baseline implements a decoupled front-end with a fetch width of 6 instructions, similar to prior work [31]. We simulate a 5-level radix tree page table, an x86 hardware page table walker, and 4-level split Page Structure Caches [64]. The simulated hardware page table walker can support up to 4 concurrent page walks [20]. Note that entries from all page table levels are stored in the cache hierarchy. Regarding hardware prefetching, our baseline considers prefetchers for all L1D, L1I, and L2C. Table 1 describes the simulated system configuration. Bold text highlights settings that are relevant to *iTP* and *xPTP* policies. Parameters M, N, and K are determined via parameter space exploration and we keep them constant for all our experiments. Our sensitivity analysis indicates that different values of parameters M and N do not provide significant performance variation. Parameter K has the highest performance impact. We found that values of parameter K that place entries at the middle of the stack (e.g. K=6, K=8) optimize the performance gains.

SMT Core. Workload co-location is a common practice in servers since it offers better CPU and memory utilization [78]. To evaluate our proposals under workload co-location, we extend ChampSim to simulate an SMT core with two hardware threads, similar to prior work [80]. Regarding instruction fetching, a basic block of instructions belonging to a thread is fetched every cycle. We switch the thread we fetch from every cycle. Our SMT model accounts for the contention in all hardware structures shared among threads.

Table 2. List of considered techniques/policies and hardware structures where they are applied.

Technique	L1D	L2	LLC	STLB
TDRRIP [79]	LRU	TDRRIP	LRU	LRU
PTP [63]	LRU	PTP	LRU	LRU
CHiRP [55]	LRU	LRU	LRU	CHiRP
CHiRP+TDRRIP	LRU	TDRRIP	LRU	CHiRP
CHiRP+PTP	LRU	PTP	LRU	CHiRP
iTP (Sec. 4.1)	LRU	LRU	LRU	iTP
iTP+xPTP (Sec. 4.3)	LRU	xPTP/LRU	LRU	iTP
iTP+TDRRIP	LRU	TDRRIP	LRU	iTP
iTP+PTP	LRU	PTP	LRU	iTP
iTP with SHiP [84]	LRU	LRU	SHiP	iTP
iTP+xPTP with SHiP [84]	LRU	xPTP	SHiP	iTP
iTP with Mockingjay [71]	LRU	LRU	Mockingjay	iTP
iTP+xPTP with Mockingjay [71]	LRU	xPTP	Mockingjay	iTP

Simulated Page Sizes. Our evaluation considers two different scenarios: i) the system uses only 4KB pages for both instructions and data (Sections 6.1, 6.2, 6.3) and ii) the system uses both 4KB pages and 2MB pages for data and instructions (Section 6.5). The scenario considering only 4KB pages is relevant since huge pages require memory contiguity and defragmentation to deliver competitive performance, which cannot be guaranteed in server systems due to their large uptimes [30, 56, 86]. In the multi-page size scenario we determine the data footprint portion allocated by 2MB pages using the methodology proposed by previous work [37, 82].

5.2 Workloads

Single Thread Workloads. Our set of single-thread workloads consists of server workloads provided by Qualcomm [24] for the CVP-1 [1] and IPC-1 [9] contests. Prior works on TLB-related optimizations [55, 80, 81] also use this set of workloads. We consider workloads with an STLB MPKI of at least 1.0. In total, our evaluation considers 120 single-core server workloads. For simulation, we use 50 million warmup instructions and 100 million instructions to measure the experimental results. We also consider the SPEC CPU 2006 [27] and SPEC CPU 2017 [8] benchmark suites. We consider all SPEC workloads in Section 3 to motivate our contributions.

SMT Workloads. To evaluate our proposals under workload colocation, we execute two different *Qualcomm Server* workloads in our extended SMT ChampSim version, described in Section 5.1. Our evaluation considers 75 chosen pairs of *Qualcomm Server* workloads, combined in three categories: i) *Intense Load*: combines two randomly chosen workloads with high STLB MPKI (must be over 1.5), ii) *Medium Load*: combines two randomly chosen workloads, one with high and one with medium STLB MPKI, and iii) *Relaxed Load*: combines two randomly chosen workloads, one with high and one with low STLB MPKI. We use the same warmup and simulation instructions as the single-core experiments.

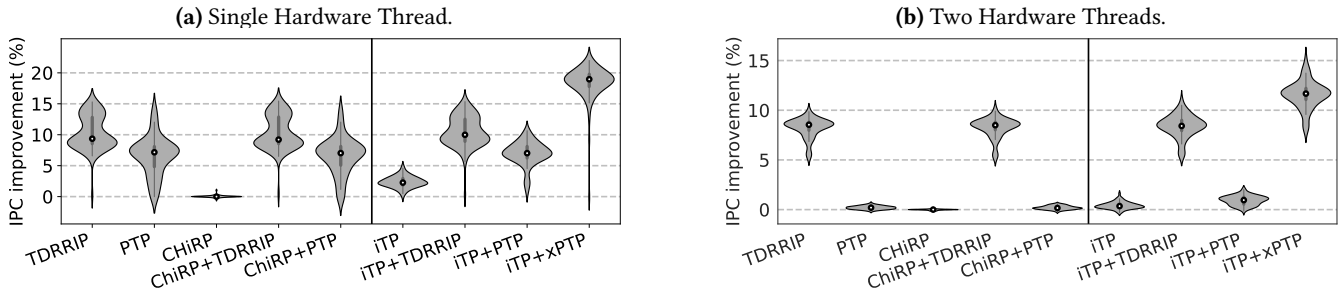


Figure 8. IPC comparison between state-of-the-art replacement policies and our proposals *iTP* and *iTP+xPTP*.

5.3 Considered Approaches

Our evaluation considers the most relevant state-of-the-art cache and STLB replacement policies. Table 2 lists all the considered policies and indicates the cache and TLB levels where we apply them. If we do not specify a particular replacement policy for a structure, the default policy is *LRU*. We consider two translation-aware cache replacement policies, *PTP* [63] and *TDRRIP* [79]. While the original versions of these translation-aware policies guide both L2C and LLC, we only use them on the L2C as our experiments indicate that only using these policies in the L2C delivers better performance for the *Qualcomm Server* workloads. We also consider the state-of-the-art STLB replacement policy, *CHiRP* [55]. We combine *CHiRP* at the STLB with either *TDRRIP* and *PTP* at the L2C, *i.e.*, *CHiRP+TDRRIP* and *CHiRP+PTP* in Table 2.

In addition, our evaluation considers our proposals: *iTP* (Section 4.1), *iTP+xPTP* (Section 4.3). We consider two additional policies, *iTP+TDRRIP* and *iTP+PTP*, where we combine the use of *iTP* in the STLB with *TDRRIP* and *PTP*, respectively, in the L2C. We evaluate *iTP* and *iTP+xPTP* considering a system using the LRU replacement policy for LLC (Sections 6.1 and 6.2), and systems using the *SHiP* [84] and *Mockingjay* [71] policies to drive LLC replacement (Section 6.3).

6 Evaluation

This section presents our experimental campaign. Section 6.1 compares our proposals, *iTP* and *iTP+xPTP*, with state-of-the-art replacement policies for L2C and STLB. Section 6.2 explains the performance gains of our proposals by quantifying their impact on the different cache and TLB levels. Section 6.3 evaluates our proposals when different state-of-the-art LLC replacement policies are used. Section 6.5 presents evaluation when multiple page sizes are used.

6.1 Performance Evaluation Against State-of-the-Art

Figure 8 compares the performance of the *iTP* and *iTP+xPTP* policies, presented in Section 4, with the state-of-the-art STLB and L2C replacement policies, listed in Table 2, over a baseline that uses the *LRU* replacement policy in all cache and TLB levels. Figures 8a and 8b show results considering one and two hardware threads, respectively. The x-axis of both figures displays all considered policies while the y-axis shows the Instructions Per Cycle (IPC) improvement over the baseline. For each policy, we present a violin plot showing the distribution of our results across all considered workloads. The geomean IPC speedup is indicated with a black point.

Figure 8a presents the performance comparison across the single-thread *Qualcomm Server* workloads. *iTP* and *iTP+xPTP* achieve geometric mean speedups of 2.2% and 18.9%, respectively, with respect to *LRU*. *PTP* and *TDRRIP* yield speedups of 7.1% and 9.3%, respectively. *CHiRP* achieves almost the same performance as *LRU*. *CHiRP* does not differentiate between instruction and data PTEs in the STLB, thus its prediction mechanism does not accurately identify highly reusable entries, which makes *CHiRP* to apply a replacement logic similar to *LRU*. Overall, *iTP+xPTP* provides the highest single-core performance gains across all considered techniques.

Figure 8b shows results considering the two-thread scenario, described in Section 5.2. *iTP+xPTP* achieves the highest speedups. Specifically, *iTP* and *iTP+xPTP* achieve geomean speedups of 0.3% and 11.4% over *LRU*, respectively. *TDRRIP* achieves a geomean 8.5% improvement with respect to LRU, while *PTP* provides very similar performance as *LRU*.

In both single- and two-thread scenarios we observe notable IPC uplifts when using *iTP* in the STLB. These gains are greatly amplified when adopting the combined approach of *iTP+xPTP*. Combining *iTP* with other state-of-the-art replacement policies that differentiate between translation and application blocks in the L2C (*iTP+TDRRIP*, *iTP+PTP*) provides benefits over their respective counterpart without *iTP* (*TDRRIP*, *PTP*). However, these policies deliver significantly lower speedups than the proposed *iTP+xPTP* scheme.

6.2 Impact on the STLB and the Cache Hierarchy

This section explains the reasons why our proposals deliver better performance than the state-of-the-art, as Section 6.1 indicates, by looking at the MPKI rates and miss latencies experienced by the cache hierarchy and the STLB.

Figure 9a shows the average MPKI observed at STLB, L2C, and LLC when applying the *iTP* and *iTP+xPTP* policies, presented in Section 4, as well as the state-of-the-art STLB and L2C replacement policies listed in Table 2. Figure 9b shows

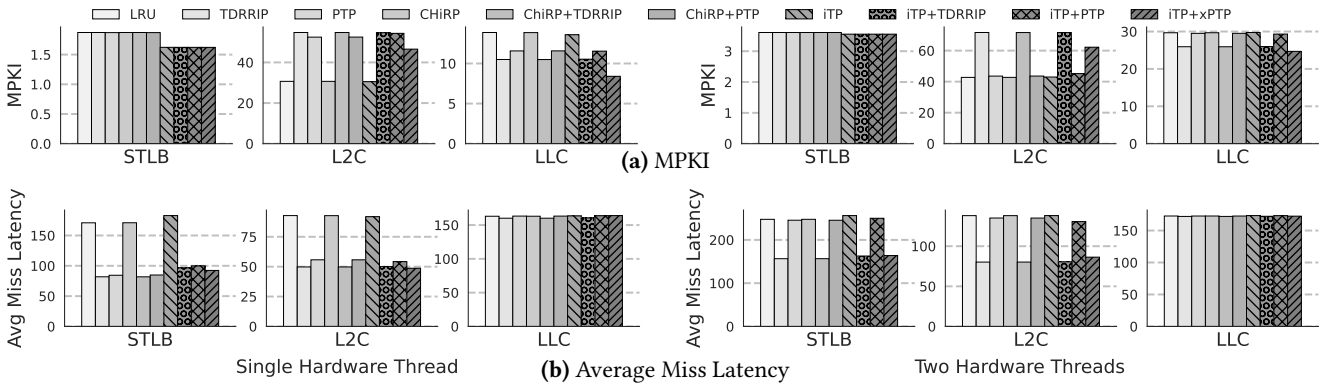


Figure 9. MPKI (9a) and average miss latency (9b) at the STLB and the cache hierarchy.

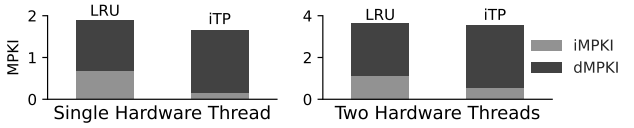


Figure 10. STLB MPKI breakdown between instruction translations (iMPKI) and data translations (dMPKI), across *LRU* and *iTP*, for one (left) and two (right) hardware threads.

the impact on the average miss latency observed at STLB, L2C, and LLC across the same policies as Figure 9a. Figure 9 presents results for both single- and two-thread scenarios in the left- and right-hand side plots, respectively.

Regarding the single-thread scenario, *iTP+xPTP* provides the highest IPC gains. Many aspects contribute to these gains. The average STLB MPKI when using *iTP+xPTP* is reduced from 1.8 to 1.6 (11.1% reduction) with respect to *LRU* while the average STLB miss latency is reduced from 170.9 to 92.3 (45.9% reduction). We observe this behavior because *iTP* prioritizes instructions over data in the STLB, thus trading instruction page walks that can cause pipeline stalls for data page walks that are quickly served from the cache hierarchy since *xPTP* avoids evicting data PTEs from the L2C. Focusing on the cache hierarchy, *iTP+xPTP* lowers the LLC MPKI rate over LRU from 13.8 to 8.4 while increasing the L2C MPKI rate over *LRU* from 30.6 to 46.5. Two factors compensate for this L2C MPKI increase and explain the superior performance of *iTP+xPTP*. First, when applying *iTP+xPTP*, the L2C MPKI of cache blocks storing data PTEs decreases with respect to *LRU* (from 1.0 to 0.4), which reduces the cost of page walks triggered by data accesses. Second, *iTP+xPTP* delivers the highest LLC MPKI reduction, which heavily reduces the average L2C miss latency, as Figure 9b shows, since most L2C misses are served by the LLC. Specifically, *iTP+xPTP* reduces average L2C miss latency over LRU by 47.5%.

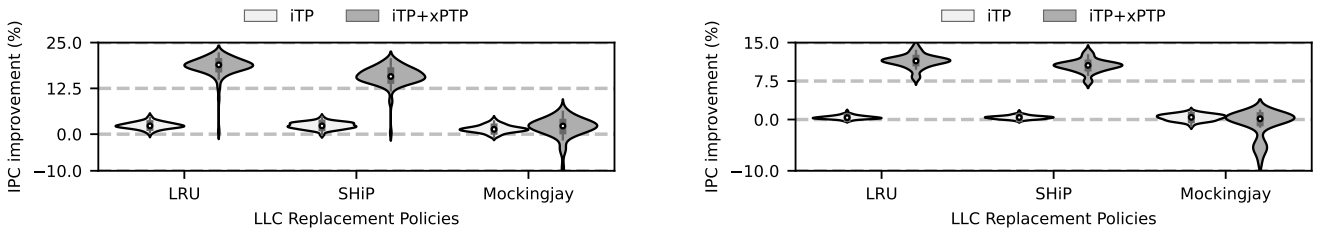
We observe similar behavior under thread co-location. *iTP+xPTP* significantly lowers the STLB miss latency with respect to *LRU*, similar to the single-thread scenario. Focusing on the caches, *iTP+xPTP* lowers the LLC MPKI over LRU from 29.6 to 24.6 while increasing the L2C MPKI over LRU from 42.7 to 62.2. The L2C MPKI increase of *iTP+xPTP* is compensated by an LLC MPKI much lower than the one experienced by *LRU*, which reduces the average L2C miss latency by 37.3%, as Figure 9b shows. *TDRRIP* and *iTP+TDRIPP* experience larger MPKI rates than *iTP+xPTP* in both L2C and LLC and, consequently, they deliver lower performance than *iTP+xPTP*.

The *xPTP* replacement policy, which prioritizes blocks containing data PTE entries in the L2C, is not conceived to be used alone since just prioritizing cache blocks storing PTEs provides low benefits. Instead, we combine the use of *xPTP* in the L2C with *iTP* in the STLB. While *iTP* maximizes the number of instruction hits in the STLB at the expense of increasing data page walks, *xPTP* reduces the impact of these additional data page walks by reducing L2C misses due to data page walks. When combined with *iTP*, *xPTP* dramatically reduces the average L2C miss latency and the LLC miss count with respect to *LRU* for both single- and two-hardware thread scenarios, as Figure 9a shows.

Figure 10 highlights the effectiveness of *iTP* on reducing STLB instruction misses by showing a breakdown of the average STLB MPKI for instructions and data corresponding to the *Qualcomm Server* workloads. Figure 10 considers both *LRU* and *iTP* for the single- and two-thread scenarios. The main takeaway of this study is that instruction translation MPKI (*iMPKI*) experiences a significant reduction when *iTP* is used as STLB replacement policy, while the data translation MPKI (*dMPKI*) suffers an increase.

6.3 Sensitivity to LLC Replacement

This section evaluates *iTP* and *iTP+xPTP* considering baseline systems using the *SHiP* [84] and the *Mockingjay* [71] policies to drive LLC replacement; for completeness, we also report the *LRU* results, similar to previous sections. Figure 11 shows geometric mean speedups over a baseline where *LRU* is used on STLB and L2C, but the corresponding LLC policy is used for each respective LLC replacement scenario.



(a) Single Hardware Thread. **(b)** Two Hardware Threads.

Figure 11. IPC comparison between state-of-the-art replacement policies and our proposals *iTP* and *iTP+xPTP*.

Focusing on the single-thread scenario, Figure 11a demonstrates that *iTP* delivers consistent performance gains across all considered LLC replacement policies, *i.e.*, its benefits are independent from the underlying LLC replacement policy. Specifically, *iTP* improves geomean performance by 2.2%, 2.3%, and 1.4% when *LRU*, *SHiP*, and *Mockingjay* operate in the LLC, respectively. *iTP+xPTP* delivers great performance uplifts when *LRU* and *SHiP* are used as LLC replacement policies (18.9% and 15.8% respectively) and modest gains (1.6%) when *Mockingjay* operates in the LLC. These moderate gains when *Mockingjay* is used to drive LLC replacement come from the fact that *Mockingjay* is less effective for the *Qualcomm Server* workloads than replacement policies like *SHiP*. Since *Mockingjay* was developed considering workloads with large data footprints and not applications with both large code and data footprints, it delivers less performance for server workloads than for desktop or HPC codes, which reduces the benefits of our proposal. When considering two-thread workloads (Figure 11b), we observe trends similar to the single-thread ones (Figure 11a).

6.4 Sensitivity to ITLB Size

Experimental results presented in previous sections assume a 64-entry ITLB (Table 1). This section quantifies the performance *iTP* and *iTP+xPTP* across different ITLB sizes. Figures 12a and 12b present the experimental results, for single- and two-threads, respectively.

Focusing on realistic ITLB sizes (64 and 128 entries), we observe geomean speedups ranging between 2.2-2.6% and 0.3-0.8% when using *iTP* for the single- and two-thread scenarios, respectively. *iTP+xPTP* yields a geomean speedup between 18.0-18.9% and 11.4-11.9% for the single- and two-thread scenarios, respectively. Overall, for ITLBs with less than 512 entries, both *iTP* and *iTP+xPTP* consistently deliver significant performance gains for both single- and two-threads evaluation with small variation. However, for the single-thread scenario, when using an ITLB of 1024 entries we observe that both *iTP* and *iTP+xPTP* deliver lower performance gains than the scenarios with smaller ITLB sizes. Specifically, *iTP* and *iTP+xPTP* improve geomean performance by 1.3% and 4.6% for single- and two-threads, respectively, assuming an 1024-entry ITLB. This behavior is expected since increasing the ITLB size to 1024 entries reduces the instruction address translation bottleneck, as Figure 1 indicates, because most instruction translation requests are served now by the ITLB, thus STLB accommodates less instruction PTEs, limiting the potential of (i) *iTP* which targets to reduce the instruction translation misses in STLB and (ii) *xPTP* which mitigates the increase in data translation page walks caused by *iTP* to optimize cache management; if *iTP* does not frequently evict data PTEs to preserve instruction ones, then also *xPTP* has minimal effect on performance. Regarding the two-thread scenario, even with 1024-entry ITLB, there is still a significant number of instruction misses in both ITLB and STLB to highlight the benefits of our proposals.

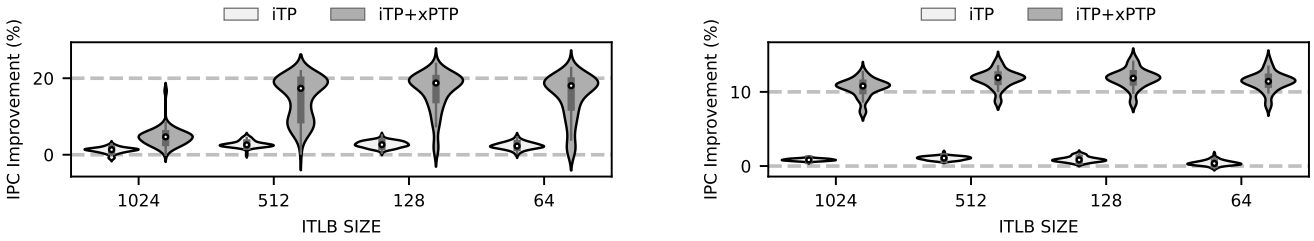
The main takeaway of this study is that our proposals, *iTP* and *iTP+xPTP*, deliver significant and consistent benefits across different ITLB sizes while highlighting that can improve performance even of future microarchitectural designs with significantly bigger ITLBs than the ones used in today's designs (*e.g.*, 64-128 entries).

6.5 Allocating Instructions and Data on Large Pages

This section evaluates our proposal with the highest performance gains, *iTP+xPTP*, against *TDRRIP*, *PTP*, and *CHiRP* considering a system that uses both 2MB and 4KB memory pages to allocate data and instructions (Section 5.1). Figure 13 shows the geomean IPC uplifts with respect to *LRU* for all considered approaches in both single- and two-hardware thread scenarios. The x-axis shows the percentage of code and data footprint allocated in 2MB pages.

For single-thread workloads, *iTP+xPTP* achieves 18.9% geomean speedup when only 4KB pages are used, while *SHiP*, *PTP*, and *CHiRP* deliver significantly lower performance than *iTP+xPTP* in this scenario. When mapping 10% of the application code and data to 2MB pages, *iTP+xPTP* delivers a speedup of 10.1%, while the competing policies perform much worse. When the portion of data and code footprint

We use the source code of *Mockingjay* provided by its authors and apply it to our experimental campaign. We verify that this *Mockingjay* implementation delivers speedups similar to the ones reported in the original paper [71] and that it outperforms *SHiP* for the SPEC workloads. The largest speedups we observe are 31.0%, 9.8%, 2.4%, 1.9% and 1.7% for the 482.sphinx3, 462.libquantum, 627.cam4, 403.gcc and 433.milc SPEC workloads, respectively.



(a) Single Hardware Thread. **(b)** Two Hardware Threads.

Figure 12. Performance of *iTP* and *iTP+xPTP* across different ITLB sizes.

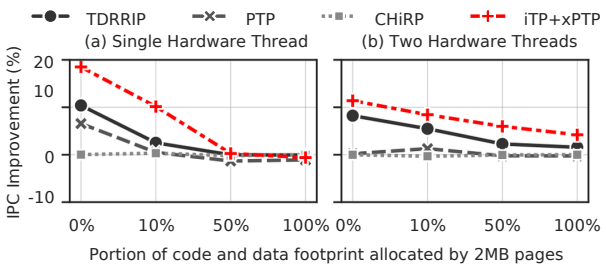


Figure 13. IPC comparison considering state-of-the-art replacement policies and *iTP+xPTP* using 4KB and 2MB pages.

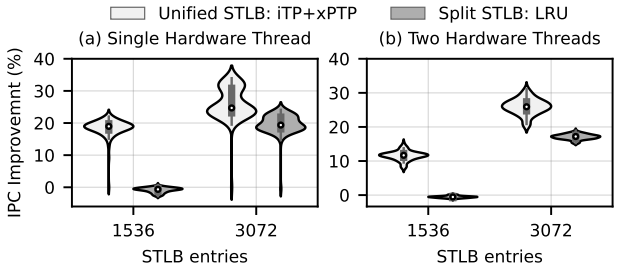


Figure 14. Comparison of Unified STLB with *iTP+xPTP* against Split STLB for instructions and data.

allocated to 2MB pages is 50% or larger, all schemes provide negligible speedups over *LRU* since the vast majority of STLB accesses (both data and instructions) are hits.

For the two-thread scenario, *iTP+xPTP* delivers performance speedups of 11.4%, 8.4%, 5.9%, and 4.2%, when 0%, 10%, 20%, 100% of code and data are mapped to 2MB pages, respectively, while *TDRRIP* delivers speedups of 8.2%, 5.4%, 2.2% and 1.5% in these same scenarios. The speedups achieved by *iTP+xPTP* are larger than the ones delivered by *TDRRIP* for all multi-size page scenarios. *PTP* and *CHiRP* deliver worse performance than *iTP+xPTP* and *TDRRIP* in all scenarios.

Figure 13 indicates the superior performance of *iTP+xPTP* with respect to state-of-the-art schemes for all considered multi-size page scenarios. In addition, Figure 13 indicates that the benefits of all considered approaches, including *iTP+xPTP*, decrease as the percentage of data and code footprint mapped to 2MB increases. This effect comes from the fact that increasing the code and data footprint mapped to 2MB pages reduces STLB misses. Even when mapping 100% of the code and data footprint to 2MB pages, *iTP+xPTP* brings a remarkable speedup of 4.2% on the two-threads scenario.

6.6 Unified STLBs vs Split STLBs

Unified STLB designs imply that both instruction and data translations are stored in the same STLB structure. Split STLB designs implement separate STLB structures for data and instruction translations. Unified STLBs face interference between instruction and data translations while Split STLBs typically incur storage waste for workloads with small code footprints. *iTP+xPTP* improves the performance of Unified STLBs, which is the TLB organization generally chosen by vendors [3, 5, 6] due to their storage efficiency.

This section compares Unified STLBs and Split STLBs with and without our proposal, *iTP+xPTP*. Specifically, we compare a Unified STLB (1536- and 3072-entries) that uses *iTP+xPTP* with (i) a 1536-entry Split STLB design (768-entry data-STLB and 768-entry instruction-STLB) and (ii) a 3072-entry Split STLB design (1536-entry data-STLB and 1536-entry instruction-STLB). The results of Figure 14 are computed over a baseline with a 1536-entry Unified STLB (Table 1), similar to all previous sections.

Figure 14 provides three main conclusions. First, we observe that a Split STLB with the same number of entries as the baseline Unified STLB is slightly behind in performance for both single-thread and two-thread scenarios (2nd violin Figures 14 (a), (b)). Second, doubling the entries of the Split STLB (1536-entry data-STLB and 1536-entry instruction-STLB, 4th violin in Figures 14 (a), (b)) provides similar single-thread IPC gains as the 1536-entry Unified STLB using *iTP+xPTP*. Note that the former scenario requires maintaining two separate STLBs for data and instructions (double storage overhead, higher power) while the latter uses *iTP+xPTP*, which incurs low storage overhead. Finally, although the 3072-entry Split STLB provides slightly higher performance than the 1536-entry Unified STLB with *iTP+xPTP*, the 3072-entry Unified STLB with *iTP+xPTP* outperforms the 3072-entry Split STLB for both single-thread and two-thread scenarios, further highlighting the benefits of *iTP+xPTP*.

7 Related Work

Translation-Oblivious Cache Replacement Policies. Prior work in translation-oblivious cache replacement policies, *i.e.*, policies that do not differentiate between blocks with instructions/data payload and blocks accommodating PTEs, can be classified in two categories: i) memoryless replacement policies that use cache block recency to drive replacement decision and do not use histories of previous misses [26, 34, 50, 59, 67, 75, 83] and ii) predictive replacement policies that predict the reuse distance of cache lines by correlating program and system features with the behavior of past accesses [21, 23, 28, 32, 35, 43, 44, 46, 51, 71, 72, 77, 85]. Our work takes a step forward since it improves the decision-making of STLB and cache replacement policies in a threefold manner: i) it makes the STLB replacement policy aware of whether an entry contains an instruction or data translation, ii) it makes the L2C replacement policy aware of the existence of blocks accommodating data PTEs, and iii) creates synergy between STLB and cache replacement policies.

Instruction-Aware Cache Replacement Policies. Recent works [33, 57] show that server and datacenter applications spend a significant portion of their total execution cycles serving code misses at the lower-level caches. CLIP [33] builds on top of the RRIP policy [34] and uses Set-Dueling [68] to increase the priority of code blocks in the L2C at the expense of having more data misses when needed. In similar spirit, Emissary [57] is another translation-oblivious L2C replacement policy that avoids evicting critical for performance code blocks. Our work is orthogonal to instruction-aware L2C replacement policies since *xPTP* does not differentiate between data and code blocks; *xPTP* gives higher priority to L2C blocks containing data translations to compensate for the extra data page walks introduced by *iTP*. A scheme that leverages *iTP* as STLB replacement policy and combine *xPTP* with Emissary at L2C has the potential to provide larger performance gains than *iTP*+*xPTP* since it would preserve critical code blocks in the L2C thanks to Emissary.

Dead Block/Page Prediction. Dead block (page) predictors anticipate whether a block (page) will be referenced again before it is evicted by the corresponding cache (TLB) structure. Although there are numerous dead block predictors in recent literature [23, 44–46, 48, 49, 51], there has been little attention paid to dead page prediction for TLBs. Mazumdar et al. [54] propose a cooperative scheme that is built on the premise that dead blocks live in dead pages. Their proposal consists of i) a dead page predictor for the STLB that bypasses pages predicted to be dead and ii) a simple dead block predictor for the LLC that exploits the propagated dead page information to drive its prediction. Instead, we make the STLB and cache replacement policies capable of differentiating between data and instruction payload for applications with large data and code footprints. Dead page and dead block prediction are complementary to our proposals.

TLB Management and Optimizations. Elnawawy et al. [22] show that a few data pages of data-intensive applications have high reuse but poor temporal locality. They propose a scheme that pins in the STLB highly used PTEs. However, this approach needs to pin hundreds of instruction translations in the STLB to achieve significant performance gains [81], which raises the STLB MPKI for data pages. *iTP* optimizes the STLB replacement policy by prioritizing the eviction of data translations over instruction translations when the STLB pressure for instructions is high. POM-TLB [69] and DVMT [11] reduce page walks by maintaining a large die-stacked L3 TLB, and allowing the application to define the page table format, respectively. Victima [37] increases TLB reach by using L2C as an additional TLB level storing evicted STLB data entries. Hashed page tables [29, 38, 73, 87] resolve TLB misses faster than radix tree page tables. Our work is complimentary to these approaches.

Translation Prefetching. Prior work proposes STLB prefetching schemes [17, 36, 80, 81]. *iTP* is orthogonal to STLB prefetching and could be extended to consider STLB prefetching in its decision-making.

8 Conclusions

This paper provides evidence that contemporary applications with large instruction footprints face significant performance degradation due to pipeline stalls caused by frequent TLB misses. In response, we propose *iTP*, an STLB replacement policy that smartly favors instruction translations over data translations in the STLB, at the cost of increasing page walks for data references. To address this issue we propose *xPTP*, an L2C replacement policy that cooperates with *iTP* by favoring data PTEs in the L2C to compensate for the increase in data page walks incurred by *iTP*. Our experimental campaign demonstrates that our proposals improve performance over the state-of-the-art TLB and cache replacement policies in both single- and two-hardware thread scenarios.

9 Acknowledgments

This work has received funding from 'Future of Computing, a Barcelona Supercomputing Center and IBM initiative' (2023) and has been partially supported by the Spanish Ministry of Science and Innovation MCINAEI/10.13039/501100011033 (contract PID2019-107255GB-C21) and ESF Investing in your future, the Generalitat of Catalunya (contract 2021-SGR-00763), the European HiPEAC Network of Excellence, and the European Processor Initiative (EPI), which is part of the European Union's Horizon 2020 research and innovation program under grant agreement No. 826647. The authors thank the Departament de Recerca i Universitats de la Generalitat de Catalunya for supporting the Research Group "Performance understanding, analysis, and simulation/emulation of novel architectures" (Code: 2021 SGR 00865).

A Artifact Appendix

A.1 Abstract

Our artifact provides (i) the implementation of *iTP* and *xPTP*, (ii) the simulation infrastructure, (iii) the set of workloads, (iv) scripts for launching simulations and scripts to reproduce the most important figures.

A.2 Artifact check-list (meta-information)

- **Program: Application traces of server applications provided by Qualcomm for CVP-1 [1] and IPC-1 [9] and (optionally) traces of SPEC CPU 2006/2017 [7, 8].**
- **Compilation: gcc**
- **Metrics: Performance Improvement.**
- **Output: We provide scripts to recreate the most important evaluation figures (Figures 8 and 9).**
- **Experiments: We provide scripts that submit the required jobs. The SLURM job manager is required.**
- **How much disk space required (approximately)?: 150MB for the Simulation infrastructure and scripts. 40GB for the Quacomm Sever traces and (optionally) 80GB for SPEC CPU 2006/2017 traces.**
- **How much time is needed to prepare workflow (approximately)?: 20 minutes**
- **How much time is needed to complete experiments (approximately)?: Single job should take about 30-40 minutes (or 8-12 hours if the batched versions is used). Running all simulations for the most important data should be complete within 24 hours. This can vary depending on the hardware and SLURM configuration.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: No License (ChampSim simulator is under Apache-2.0 license).**
- **Data licenses (if publicly available)?: No License**
- **Workflow framework used?: SLURM Job Manager.**

A.3 Description

A.3.1 How to access. The source code, scripts and simulation infrastructure can be found in https://github.com/dchasap/itp_asplos25_AE or https://doi.org/10.5281/zenodo.14497052. The Qualcomm Server workload traces can be found at The SPEC CPU 2006/2017 workload traces can be found at https://doi.org/10.5281/zenodo.10959704 and https://doi.org/10.5281/zenodo.10960003, respectively.

A.3.2 Hardware dependencies. Any hardware capable of compiling ChampSim [2].

A.3.3 Software dependencies. SLURM manager for job management, python packages: zenodo_get, pandas, matplotlib, seaborn and scipy.

A.3.4 Data sets. Qualcomm Server workload traces, (optional) SPEC CPU 2006/2017 workload traces.

A.4 Installation

To deploy and setup the simulation infrastructure, follow these steps:

- Clone the artifact repository for the scripts and simulation infrastructure or download the tarball from from https://doi.org/10.5281/zenodo.14497052. To clone the repository run:
  git clone https://github.com/dchasap/itp_asplos25_AE

- Install the python dependncies:
  python3 -m pip install zenodo_get
  python3 -m pip install scipy
  python3 -m pip install pandas
  python3 -m pip install seaborn

- Download the Qualcomm Server workload traces:
  ./download_traces.sh AE
  The Qualcomm Server workload traces can also be manually downloaded from https://doi.org/10.5281/zenodo.14045185. There are two tar archives that need to extracted in itp_asplos25_AE/traces/qualcomm_srv
- (Optional) Download the SPEC CPU 2006/2017 workload traces:
  ./download_traces.sh spec
  The SPEC CPU 2006/2017 traces can also be manually from https://doi.org/10.5281/zenodo.10959704 and https://doi.org/10.5281/zenodo.10960003. Both workloads need to be placed in itp_asplos25_AE/traces/spec (The spec benchmarks are only needed to generate the figures in the motivation Section).

A.5 Experiment workflow

To reproduce the most important experimental results from Section 6 do the following:

- cd itp_asplos25_AE
- Check and edit SLURM's directives if your setup requires any special configuration to be set (e.g. queues, account) in itp_asplos25_AE/scripts/submit_jobs.sh and itp_asplos25_AE/scripts/submit_jobs_batch.sh
- source env.sh (sets the default paths and directories)
- ./submit_experiments.sh AE or ./submit_experiments_batch.sh AE (Launches experiments)

The batched version significantly reduces the number of jobs, but a single job takes substantially more time to execute. The *AE* argument runs experiments for the most important evaluation figures - if only a certain figure is needed use the argument *fig_XX*, where *XX* the digits corresponding to the figure number as numbered in the document. If *all* is provided as an argument, then all experiments from Sections 3 and 6 will be scheduled. This is not recommended as SLURM queue limit will most likely be exceeded).

A.6 Evaluation and expected results

To validate the most important experimental results from Section 6 follow these last steps:

- cd itp_asplos25_AE (if not there already)

- source env.sh (reset the default paths and directories)
- ./gen_plots.sh AE (Parses experiment data and generates plots for the most important evaluation figures)

A single figure can be generated by replacing *AE* with *fig_XX*. To generate all figures, use the *all* argument instead. The generated data is placed in the itp_asplos25_AE/figures directory. The generated figures can be either compared directly with the ones presented in the document or the pre-generated ones in itp_asplos25_AE/figures_PUBLISHED.)

A.7 Experiment customization

A.7.1 Adding benchmarks:

- Place your *.champsimtraces.xz in the traces directory, in their own folder (e.g. ./traces/mybenchmarks).
- Create a bash script file (e.g. mybenchmarks.sh) and place it in the scripts directory. This file should define two variables, one for the directory name the traces are in and one listing all the traces names. Review qualcomm_srv_workloads.sh in the scripts directory to see an example.
- Edit scripts/benchmarks.sh by adding at the top of the script source ./scripts/mybenchmarks.sh and a corresponding if-statement as following the example shown in the file with the Qualcomm Server workloads. The name you pick for the if-statement is going to be used to identify the new benchmarks in all other scripts (e.g. mybenchmarks).

A.7.2 Setting and running experiments:

- In the ChampSim direcotry edit the simulator configuration file champsim_fdip_baseline.json. You can edit prefetchers, replacement polcies and TLB and cache sizes. This file is the base for all experiments.
- To create you own set of experiments copy one of the bash scripts in exp_conf/fig_XX.sh. This file is used as a configuration file by the job launching script. Edit the variable *BENCHSUITES*, assigning the name of the benchmark suite you want to use (e.g. *BENCHSUITES=mybenchmarks*). The variable *CONFIGURATION_TAGS* needs to also be edited. This variable should hold a list of the experiments to run. The experiment names are parsed by ./scripts/gen_champsim_conf.py to modify the baseline configuration. The naming conversion follows the scheme:
  cache_type-r.rep_pol-s.num_sets-w.num_ways.
  For example if you want to just enable *iTP* and *xPTP*, and change the size of the *LLC*, you can use:
  stlb-r.itp_l2c-r.xptp_llc-s.1537-w.16.
- To launch the experiments run:
  ./scripts/submit_experiment.sh ./exp_conf/myexp.sh
  Where ./exp_conf/myexp.sh is the bash scripts you created in the previous step. The batched version can ./scripts/submit_experiment_batch.sh can also be used in the same manner.

A.7.3 Parsing experiment data:

- To generate a CSV file with the experimental data generated by the previous steps, run:
  ./scripts/parse_data.sh ./exp_conf/myexp.sh
  The generated data is placed in ./stats/exp_name.csv.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] Championship Value Prediction (CVP). https://www.microarch.org/cvp1/. Accessed: 05-9-2022.

[2] ChampSim. https://crc2.ece.tamu.edu/. Accessed: 05-9-2022.

[3] Intel golden cove vs raptor cove vs redwood cove vs lion cove: Intel's p-core architectures compared. https://hardwaretimes.com/skylake-vs-sunny-cove-vs-golden-cove-vs-redwood-cove-vs-lion-cove-intel-p-core-architectures/.

[4] 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.

[5] Memory management unit components – arm. https://developer.arm.com/documentation/107734/0001/Memory-management/Memory-Management-Unit-components.

[6] Qualcomm's oryon core: A long time in the making. https://chipsandcheese.com/2024/07/09/qualcomms-oryon-core-a-long-time-in-the-making/.

[7] SPEC CPU 2006. https://www.spec.org/cpu2006/. Accessed: 05-9-2022.

[8] SPEC CPU 2017. https://www.spec.org/cpu2017/. Accessed: 05-9-2022.

[9] The 1st Instruction Prefetching Championship. https://research.ece.ncsu.edu/ipc/. Accessed: 05-9-2022.

[10] 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.

[11] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th International Symposium on Computer Architecture*, ISCA '17, pages 457–468, New York, NY, USA, 2017. ACM.

[12] 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 '18, pages 643–656, Feb 2018.

[13] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656, 2018.

[14] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 462–473, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM.

[16] Abhishek Bhattacharjee. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, pages 383–394, New York, NY, USA, 2013. ACM.

[17] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 359–370, New York, NY, USA, 2010. ACM.

[18] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Trans. Comput. Syst.*, 3(1):31–62, feb 1985.

[19] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[20] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[21] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 389–400, USA, 2012. IEEE Computer Society.

[22] Hussein Elnawawy, Rangeen Basu Roy Chowdhury, Amro Awad, and Gregory T. Byrd. Diligent TLBs: A Mechanism for Exploiting Heterogeneity in TLB Miss Behavior. In *Proceedings of the International Conference on Supercomputing*, ICS '19, page 195–205, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193, 2017.

[24] Josué Feliu, Arthur Perais, Daniel Jimenez, and Alberto Ros. Rebasing microarchitectural research with industry traces. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–114, Ghent, Belgium, October 2023. IEEE Computer Society.

[25] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak 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*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[26] Hongliang Gao and Chris Wilkerson. A dueling segmented lru replacement algorithm with adaptive bypassing. 2010.

[27] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, sep 2006.

[28] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, page 209–220, USA, 2002. IEEE Computer Society.

[29] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, pages 39–50, New York, NY, USA, 1993. ACM.

[30] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[31] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Reestablishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 172–182, 2021.

[32] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89, June 2016.

[33] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C. Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353, 2015.

[34] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA, 2010. Association for Computing Machinery.

[35] Daniel A. Jiménez and Elvira Teran. Multiperspective Reuse Prediction. In *Proceedings of the 50th International Symposium on Microarchitecture*, MICRO '17, page 436–448, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th International Symposium on Computer Architecture*, ISCA '02, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[37] Konstantinos Kanelopoulos, Hong Chul Nam, F. Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide-Basilio Bartolini, and Onur Mutlu. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *to appear in proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[38] Konstantinos Kanelopoulos, Kosta Stojiljkovic, Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu. Utopia: Fast and efficient address translation via hybrid restrictive & flexible virtual-to-physical address mappings. In *to appear in proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[39] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[40] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[41] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015.

[42] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[43] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*, pages 245–250, 2007.

[44] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture, pages 175–186, Dec 2010.

[45] Mazen Kharbutli, Moath Jarrah, and Yaser Jararweh. Scip: Selective cache insertion and bypassing to improve the performance of last-level caches. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–6, 2013.

[46] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[47] Rakesh Kumar, Boris Grot, and Vijay 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*, ASPLOS '18, pages 30–42, New York, NY, USA, 2018. ACM.

[48] An-Chow Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 139–148, 2000.

[49] An-Chow Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[50] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, page 134–143, New York, NY, USA, 1999. Association for Computing Machinery.

[51] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Nov 2008.

[52] Suyash Mahar, Hao Wang, Wei Shu, and Abhishek Dhanotia. Workload behavior driven memory subsystem design for hyperscale. https://arxiv.org/abs/2303.08396, 2023.

[53] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd International Symposium on Microarchitecture*, MICRO '19, pages 1023–1036, New York, NY, USA, 2019. ACM.

[54] Chandrashis Mazumdar, Prachatos Mitra, and Arkaprava Basu. Dead page and dead block predictors: Cleaning tlbs and caches together. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 507–519, 2021.

[55] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez. CHiRP: Control-Flow History Reuse Prediction. In *proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 131–145, 2020.

[56] N. P. Nagendra, G. Ayers, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. *IEEE Micro*, 40(3):56–63, 2020.

[57] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A. Pokam, Simone Campanoni, and David I. August. Emissary: Enhanced miss awareness replacement policy for l2 instruction caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[58] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, pages 27–38, New York, NY, USA, 1993. Association for Computing Machinery.

[59] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[60] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 233–244. IEEE Press, 2017.

[61] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[62] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. *SIGARCH Comput. Archit. News*, June 2017.

[63] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every walk's a hit: Making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 128–141, New York, NY, USA, 2022. Association for Computing Machinery.

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

[65] 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 '14, pages 558–567, Feb 2014.

[66] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th International Symposium on Microarchitecture*, MICRO '12, pages 258–269, Washington, DC, USA, 2012. IEEE Computer Society.

[67] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 381–391, New York, NY, USA, 2007. Association for Computing Machinery.

[68] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. *SIGARCH Computer Architecture News*, 35(2):381–391, jun 2007.

[69] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th International Symposium on Computer Architecture*, ISCA '17, pages 469–480, New York, NY, USA, 2017. ACM.

[70] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB Preloading. In *Proceedings of the 27th International Symposium on Computer Architecture*, ISCA '00, pages 117–127, New York, NY, USA, 2000. ACM.

[71] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, 2022.

[72] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 413–425, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep 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*, ASPLOS '20, pages 1093–1108, New York, NY, USA, 2020.

Association for Computing Machinery.

[74] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 513–526, 2019.

[75] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 385–396, Dec 2006.

[76] David Tarjan and Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, sep 2005.

[77] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[78] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23(2):392–403, may 1995.

[79] Vasudha Vasudha and Biswabandan Panda. Address translation conscious caching and prefetching for high performance cache hierarchy. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 311–321, 2022.

[80] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. Morrigan: A Composite Instruction TLB Prefetcher. In *Proceedings of the 54th International Symposium on Microarchitecture*, MICRO '21, page 1138–1153, New York, NY, USA, 2021. Association for Computing Machinery.

[81] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. Exploiting Page Table Locality for Agile TLB Prefetching. In *Proceedings of the 48th International Symposium on Computer Architecture*, ISCA '21, pages 85–98, 2021.

[82] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas. Page Size Aware Cache Prtefetching. In *Proceedings of the 55th International Symposium on Microarchitecture*, MICRO '22, pages 956–974, 2022.

[83] W.A. Wong and J.-L. Baer. Modified lru policies for improving second-level cache behavior. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 49–60, 2000.

[84] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 430–441, New York, NY, USA, 2011. Association for Computing Machinery.

[85] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, Dec 2011.

[86] Zi Yan, David Nellans, Daniel Lustig, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 698–710, 2019.

[87] Idan Yaniv and Dan Tsafrir. Hash, Don'T Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, pages 337–350, New York, NY, USA, 2016. ACM.

[88] Y. Zhou, X. Dong, A. L. Cox, and S. Dwarkadas. On the Impact of Instruction Address Translation Overhead. In *Proceedings of the 2019 International Symposium on Performance Analysis of Systems and Software*, ISPASS '19, pages 106–116, March 2019.