Morrigan: A Composite Instruction TLB Prefetcher

Georgios Vavouliotis
georgios.vavouliotis@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya

Lluc Alvarez
lluc.alvarez@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya

Boris Grot
boris.grot@ed.ac.uk
University of Edinburgh

Daniel A. Jiménez
djimenez@acm.org
Texas A&M University

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

ABSTRACT

The effort to reduce address translation overheads has typically targeted data accesses since they constitute the overwhelming portion of the second-level TLB (STLB) misses in desktop and HPC applications. The address translation cost of instruction accesses has been relatively neglected due to historically small instruction footprints. However, state-of-the-art datacenter and server applications feature massive instruction footprints owing to deep software stacks, resulting in high STLB miss rates for instruction accesses.

This paper demonstrates that instruction address translation is a performance bottleneck in server workloads. In response, we propose *Morrigan*, a microarchitectural instruction STLB prefetcher whose design is based on new insights regarding instruction STLB misses. At the core of Morrigan there is an ensemble of table-based Markov prefetchers that build and store variable length Markov chains out of the instruction STLB miss stream. Morrigan further employs a sequential prefetcher and a scheme that exploits page table locality to maximize miss coverage. An important contribution of the work is showing that access frequency is more important than access recency when choosing replacement candidates. Based on this insight, Morrigan introduces a new replacement policy that identifies victims in the Markov prefetchers using a frequency stack while adapting to phase-change behavior. On a set of 45 industrial server workloads, Morrigan eliminates 69% of the memory references in demand page walks triggered by instruction STLB misses and improves geometric mean performance by 7.6%.

CCS CONCEPTS

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

KEYWORDS

virtual memory, address translation, translation lookaside buffer, TLB prefetching, TLB management, markov prefetching

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480049

ACM Reference Format:
Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel A. Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3466752.3480049

1 INTRODUCTION

Paging-based virtual memory is a fundamental feature of today's computers. To mitigate the high latency cost of page walks, Translation Lookaside Buffers (TLBs) cache the most recently used virtual-to-physical translations. Despite the use of multi-level TLB hierarchies and other hardware and software schemes for accelerating address translation, frequent data TLB misses still cause significant performance degradation due to long miss penalties [30, 32, 40, 47, 54, 58, 63]. In response, the research community has proposed many techniques for reducing the overhead of address translation associated with data accesses [36, 38, 53, 56, 60, 66, 68, 69, 73, 74, 79, 82].

Recent work [54, 62, 65, 83] has shown that modern server and datacenter applications not only have big datasets, but also large code footprints. Huge binaries and deep software stacks cause frequent instruction cache and instruction TLB misses, compromising performance due to unavoidable pipeline stalls. The instruction footprint of these applications increases at around 20-30% per year [54], indicating that the front-end bottleneck is likely to get worse.

When it comes to instruction address translation, TLB pressure caused by massive code working set sizes is amplified by contention in the second-level TLB (STLB), which is shared between instruction and data translations. Instruction references evict useful data translations and vice versa, imposing additional performance penalties. However, instruction STLB (iSTLB) misses are more critical than data STLB (dSTLB)[1] misses since instruction references are on the critical path of pipeline execution, while data misses can overlap independent instructions thanks to out-of-order execution, partially hiding their latency costs. Indeed, a recent work [65] shows that iSTLB misses are a critical bottleneck in Facebook workloads. Therefore, iSTLB misses are a growing problem in servers.

The impact of instruction address translation in terms of performance and page walk memory references has received minimal attention over the years. Existing software approaches comprise either compile-time techniques for code layout optimization [64] or operating system schemes leveraging large pages [43, 59, 83]. On the hardware side, there are incremental and disruptive schemes

[1] iSTLB and dSTLB refer to instruction and data references to the STLB, respectively.

for reducing TLB misses. While developed for data TLB misses, these approaches could also be effective for instruction TLB misses. Incremental approaches try to increase TLB reach [41, 68, 69], but they are limited by coalescing opportunities exposed by the application and the OS. Disruptive approaches call for an overhaul of the virtual memory subsystem [28, 55], which hinders their adoption and may introduce new security vulnerabilities.

This paper highlights that iSTLB misses are a bottleneck in server workloads because their large code footprints pressure the STLB, resulting in long-latency page walks for fetching the corresponding address translations. Specifically, on a suite of contemporary industrial server workloads, we find that over 40% of all STLB misses are caused by instruction references. Our findings corroborate the conclusions of previous industry works showing iSTLB pressure to be a performance bottleneck in their workloads [54, 62, 65].

Furthermore, we show that prior dSTLB prefetchers [53] are ineffective at capturing the iSTLB misses because (i) they correlate patterns with features that are unable to provide accurate iSTLB prefetches, and (ii) they use access recency for choosing prefetch candidates which does not correlate well with iSTLB misses. When applied to iSTLB prefetching, existing dSTLB prefetchers improve the performance on industrial server workloads by up to 1.6%, whereas the opportunity from perfect iSTLB prefetching is 11.1%.

We also examine the state-of-the-art instruction cache prefetchers [22] and conclude that they, too, are ineffective at prefetching for the iSTLB miss stream. Instruction prefetchers target the L1 I-cache and typically find the needed cache blocks in the L2 or the LLC [47, 72], which means that they are tuned for relatively short prefetch distances. Meanwhile, iSTLB misses result in page walks that cause serialized accesses to the memory hierarchy. Depending on the memory hierarchy level where these accesses are served, the page walk can take from tens to hundreds of cycles, which cannot be always covered by instruction cache prefetchers.

Based on these observations, this paper introduces *Morrigan*, a microarchitectural iSTLB prefetcher. To the best of our knowledge, this is the first work to characterize iSTLB misses and the first iSTLB prefetcher. Morrigan is composed of two complimentary prefetching modules. The first module is the *Irregular Instruction TLB Prefetcher (IRIP)*, an ensemble of four prediction tables that efficiently build and store variable length Markov chains from the iSTLB miss stream. IRIP is enhanced with a new replacement policy, named *Random-Least-Frequently-Used (RLFU)*, that drives replacements based on a frequency stack of iSTLB misses. RLFU uses randomness to avoid evicting recently installed but not yet frequently accessed entries, thus efficiently accommodating changes in the instruction access patterns, *e.g.*, due to phase-based behavior. The second module of Morrigan is the *Small Delta Prefetcher (SDP)*, a sequential prefetcher activated when the IRIP module is unable to produce new prefetches. Finally, both IRIP and SDP exploit page table locality [69, 79] to perform cost-effective spatial prefetching.

In summary, this paper makes the following contributions:

- We provide a first study on iSTLB prefetching using a set of 45 industrial server workloads [14, 22]. Key conclusions of the study are that (i) state-of-the-art designs of dSTLB prefetchers are unable to cover iSTLB misses, and (ii) instruction cache prefetchers are ineffective at eliminating iSTLB misses.
- We demonstrate that iSTLB misses (i) follow a skewed distribution, with a modest number of instruction pages responsible for the majority of the iSTLB misses, and (ii) have spatial locality limited to a small region around the triggering miss.
- We propose *Morrigan*, a novel iSTLB prefetcher composed of two specialized prefetch engines: a novel Markov-based prefetching module that uses a new frequency-based replacement policy to manage its internal state, and an enhanced small delta prefetcher.
- Across a set of 45 industrial server workloads[14, 22], Morrigan provides a geometric mean speedup of 7.6% and reduces the references to the memory hierarchy due to demand page walks for instructions by 69% over a baseline without iSTLB prefetching.

2 VIRTUAL MEMORY SUBSYSTEM

Each memory access on a paging-based virtual memory system requires a virtual-to-physical address translation. To accelerate address translation and improve virtual memory management, modern systems use a combination of software and hardware support.

On the software side, the *page table* is an OS-managed and architecturally-visible structure that contains the virtual-to-physical translations for all pages loaded to memory. In x86-64 architectures, the page table is implemented as a multi-level radix tree.

On the hardware side, the *Translation Lookaside Buffer (TLB)* and the *MMU-Caches* are hardware structures dedicated to alleviate the address translation overheads. TLBs cache the most recently used virtual-to-physical translations. Modern architectures implement multi-level TLB hierarchies, with small instruction and data first-level TLBs (I-TLB and D-TLB) and a large second-level TLB (STLB).

On each memory access (either instruction or data), the corresponding first-level TLB is accessed and, in case of a miss, the STLB is looked up. On STLB misses, the page table walker is invoked, traversing the page table to find the requested translation. Frequent page walks have a pernicious performance impact since they require multiple accesses to the memory hierarchy. To reduce page walk latency, MMU caches (called Page Structure Caches (PSCs) on x86 [67]) cache partial translations, hence reducing the number of page walk accesses to the memory hierarchy. Finally, page table entries (PTEs) from both intermediate and leaf nodes of the page table are also cached in the existing cache hierarchy.

Page Table Locality. In x86-64 architectures, the cache line size is 64 bytes and each PTE occupies precisely 8 bytes. As a result, a single 64-byte cache line can accommodate up to 8 contiguously-stored PTEs [37, 69, 76, 79]. When a requested PTE is read from memory, it is grouped with 7 neighboring PTEs and they are stored into a 64-byte cache line. Therefore, a single cache line stores the requested PTE plus 7 more PTEs that do not require additional accesses to the memory hierarchy to be prefetched.

2.1 Translation Prefetching

STLB misses (either instruction or data) trigger long-latency page walks. Accurately prefetching PTEs ahead of demand STLB accesses can improve performance by reducing STLB misses.

Figure 1 depicts the operation of a system with STLB prefetching, considering the most common scenario whereby a Prefetch Buffer (PB) is used to store the prefetched PTEs and the prefetch logic is engaged on STLB misses [26, 53, 79]. When an instruction or data

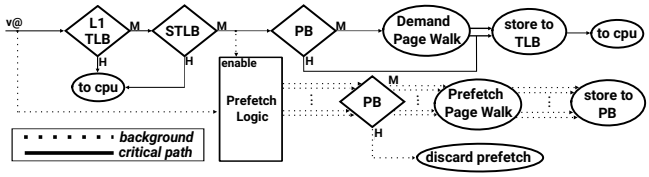

Figure 1: System with STLB prefetching. Diamonds indicate decision points, circles are actions.

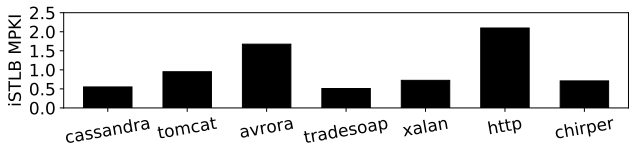

Figure 2: iSTLB MPKI of Java server workloads from the Java DaCapo [39] and Java Renaissance [71] benchmark suites.

memory access occurs, the corresponding first-level TLB is looked up and, on a miss, the STLB is probed. In case the STLB misses, the requested PTE is searched for in the PB. If the translation is present in the PB, it is moved to the STLB, the page walk is avoided, and the processor replays the request. On a PB miss, a *demand page walk* is initiated to fetch the corresponding translation. In case of either PB hit or miss, the STLB prefetcher is activated and produces new prefetches. Each prefetch requires a *prefetch page walk* to fetch the corresponding translation into the PB. Note that (i) the prefetch page walks are triggered in the background, (ii) prefetches are speculative events, thus only non-faulting prefetches are permitted, and (iii) before issuing new prefetches, the prefetch logic checks if the translation already resides in the PB, but not in the STLB, since searching the STLB for duplicates would contend with demand STLB accesses, potentially delaying the latter.

To the best of our knowledge, there is no previously proposed instruction STLB (iSTLB) prefetcher. However, state-the-art data STLB (dSTLB) prefetchers, discussed next, can also be used to attempt to capture the iSTLB miss stream.

Sequential Prefetcher (SP). SP [53, 78] prefetches the PTE of the page located next to the one triggered the STLB miss.

Arbitrary Stride Prefetcher (ASP). ASP [31, 53] targets varying stride patterns. To do so, it uses a prediction table indexed by the PC of the instruction that triggered the STLB miss.

Distance Prefetcher (DP). DP [53] correlates patterns with the distance between pages. To do so, DP uses a prediction table indexed by the distance between the current and the previous missing pages.

Markov Prefetcher (MP). MP [53] targets irregular STLB patterns by building Markov chains out of the STLB miss stream. MP employs a prediction table with three fields per entry; the virtual page for indexing, and two prediction slots that store the pages of the PTEs to be prefetched when a new STLB miss occurs on that page.

3 MOTIVATION

This section elaborates on the front-end bottleneck of servers and motivates the need for new approaches that alleviate the instruction address translation overheads, highlighting the potential performance gains of applying instruction STLB (iSTLB) prefetching.

3.1 Front-end Bottleneck

Modern server workloads have massive instruction working sets that span many levels of the software stack, making the front-end of the processor a major performance pain point [47]. Indeed, recent work from Google [54, 62] demonstrates that their server workloads face severe problems due to pressure on front-end structures. Moreover, they highlight that the front-end bottleneck is increasing, since most of these server applications exhibit high instruction growth rates (~20-30% per year), outpacing the growth in instruction cache and TLB sizes. Specifically, Kanev *et al.* [54] reveal that the front-end stalls of the Google server workloads account for 15-30% of pipeline slots, with many workloads being starved for instructions for 5-10% of cycles. Similarly, another recent work [65] reveals that Facebook workloads experience serious bottlenecks due to front-end stalls mostly caused by iSTLB misses.

To justify that instruction address translation is a significant bottleneck in server applications, we analyze the iSTLB behavior of server applications from (i) the Java DaCapo suite [39] (cassandra, tomcat, avrora, tradesoap, xalan), and (ii) the Java Renaissance suite [71] (http, chirper). We run these server applications on an Intel Skylake CPU with a 1536-entry STLB, and gathered performance counters associated with the iSTLB accesses using *perf* [17].

Figure 2 presents the iSTLB MPKI rates of these workloads. For this experiment, we enable the Transparent Huge Page support to use 2MB pages for data accesses while mapping the code pages into 2MB pages using *libhugetlbfs* since there is no transparent way to map code pages into huge pages today (Section 5 elaborates on the implications of using huge pages for code). We observe that, even with huge pages, these applications experience high iSTLB MPKI rates that range between 0.6 and 2.1, which results in over 5% of their execution cycles spent in iSTLB miss handling.

Intuitively, the increasing instruction footprint of server applications affects the performance of the I-TLB as well as the STLB, since more instruction page table entries (PTEs) must be allocated to map the instruction working set of the applications. Hence, the I-TLB experiences high MPKI rates and, as a result, more requests for instruction address translations are sent to the STLB. Since the STLB contains both data and instruction PTEs, there is increasing contention between them. Higher contention leads to more frequent STLB misses that must be resolved through a long-latency page walk. However, iSTLB misses are more critical than data STLB (dSTLB) misses because instruction references are on the critical path of execution, while data misses can overlap the execution of independent instructions in out-of-order processors. This is the reason why processor vendors (i) employ larger I-TLBs than D-TLBs (*e.g.,* Intel's Skylake 2018 chips have an 128-entry (8-way) I-TLB and an 64-entry (4-way) D-TLB [27]), and (ii) keep increasing the STLB size – from a 512-entry STLB for Sandy Bridge [5] to a 1024-entry STLB for Haswell [6], and a 1536-entry STLB for Coffe Lake [9].

3.2 Analyzing Industrial Server Workloads

To validate the observations of Section 3.1, we analyze the instruction cache (I-cache) and TLB behavior of 45 industrial server workloads provided by Qualcomm (QMM) for CVP-1 [14] and IPC-1 [22].

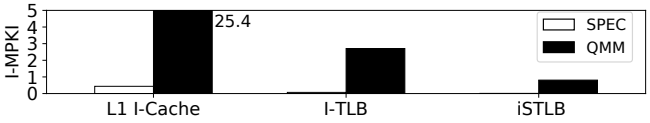

Figure 3: Instruction MPKI for front-end structures.

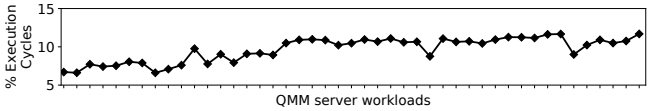

Figure 4: Cycles spent serving iSTLB accesses.

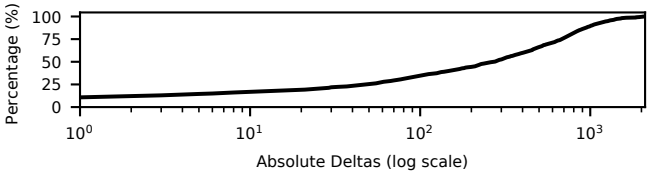

Figure 5: Accumulative distribution of deltas (absolute values) between pages that produce consecutive iSTLB misses.

The QMM workloads were also used in recent works on TLB management [61, 79]. We further study the SPEC CPU 2006 [2] and SPEC CPU 2017 [10] benchmark suites. This analysis is conducted using ChampSim [15] enhanced with a realistic x86 page table walker. Section 5 explains in detail our experimental setup.

Figure 3 presents the average MPKI rates of the L1 I-cache, the I-TLB, and the STLB (considering only the instruction misses) for the SPEC and the QMM workloads. We observe that (i) the QMM workloads experience an order of magnitude more instruction misses in the three hardware structures compared to the SPEC workloads, corroborating the conclusions of prior industrial works from Google [54, 62], presented in Section 3.1, and (ii) the iSTLB MPKI rates of the QMM workloads are similar to the ones of the Java DaCapo and Java Renaissance workloads (Section 3.1).

Focusing on the QMM workloads, we measured the fraction of the STLB misses that are caused by instruction and data references. We found that the iSTLB misses constitute 41.6%, on average, of the total STLB misses (the rest 58.4% are dSTLB misses). We further measured that the average page walk latency of iSTLB and dSTLB misses is 69 cycles and 112 cycles, respectively. Higher page walk latency is observed for the dSTLB misses because the data footprint is larger than the instruction footprint, thus, data PTEs experience worse cache locality than the instruction PTEs, resulting in higher page walk latencies. However, unlike dSTLB misses – whose latency can be partially hidden by exploiting ILP and MLP in out-of-order cores – iSTLB misses cause unavoidable pipeline stalls. Hence, iSTLB misses constitute an important performance bottleneck in server workloads.

Intel's VTune profiler [1, 3, 7, 42] considers instruction address translation as a bottleneck when the stall cycles due to iSTLB accesses represent more than 5% of the total execution cycles. Figure 4 shows the cycles spent serving iSTLB accesses as a percentage of the total execution cycles for the QMM workloads. We observe that the QMM workloads spend 6.6%-11.7% of their execution cycles serving iSTLB requests, exceeding the 5% threshold. Therefore, instruction address translation is a bottleneck for the QMM workloads.

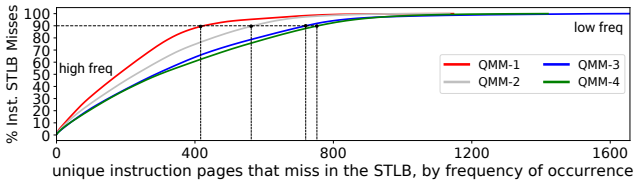

Figure 6: Instruction pages sorted by STLB miss frequency.

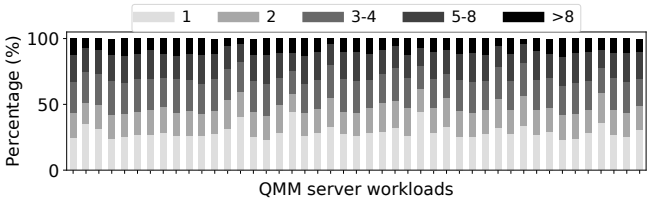

Figure 7: Number of successor pages per instruction page that misses in the STLB.

3.3 Understanding the iSTLB Misses

To understand the behavior of iSTLB misses, Figure 5 depicts the accumulative distribution of deltas (absolute values) between pages that produce consecutive iSTLB misses for the QMM workloads in order of increasing deltas. While we observe a wide distribution of deltas, we note that small deltas occur frequently (*e.g.*, deltas from 1 to 10 account for 19% of the total deltas).

> ***Finding 1.*** *iSTLB misses have only limited spatial locality mostly restricted to a small region around the triggering miss.*

Next, we analyze the distribution of iSTLB misses. Figure 6 plots the accumulative distribution of iSTLB misses per page in order of decreasing page occurrence frequency, considering a set of representative QMM workloads. The rest of the QMM workloads follow a distribution that is either close or in between the ones presented in Figure 6. We observe that a small number of pages is responsible for a significant fraction of all iSTLB misses. Specifically, 400-800 pages cause 90% of the iSTLB misses across all QMM workloads.

> ***Finding 2.*** *Most iSTLB misses can be attributed to a modest number of instruction pages.*

We define *successor page* as a page immediately following a given page in the iSTLB miss stream.[2] Figure 7 shows a breakdown of the average number of successors per each instruction page that missed in the STLB, across all QMM workloads. It can be observed that (i) a significant fraction of instruction pages has only 1 or 2 successor pages, (ii) the percentage of instruction pages that have up to 4 and up to 8 successor pages is also large, and (iii) only a small number of instruction pages have more than 8 successor pages.

Figure 7 reveals that a significant fraction of the instruction pages has more than 2 and up to 8 successors. However, to alleviate the instruction address translation bottleneck, it is natural to mainly focus on the instruction pages that miss the most in the STLB. Figure 8 shows the probability of accessing a specific successor for the top 50 instruction pages that miss the most in the STLB, across

[2]Page Y is a successor of page X if an iSTLB miss on page X is immediately followed by an iSTLB miss on page Y.

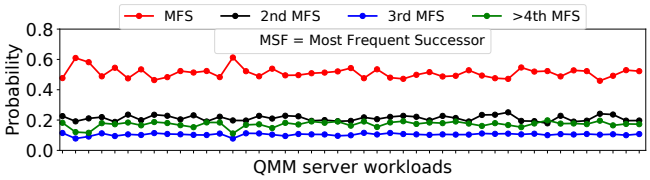

Figure 8: Probability of accessing the same successor page after an iSTLB miss for a given page.

all QMM workloads. On average, 51% of the time the most-frequent successor is accessed after an iSTLB miss, while 21% and 11% of the time the same second and third most-frequent successors are accessed after a miss, respectively. The remaining 17% of the times, the access after a miss is to a less-frequent successor page.

Finding 3. *Instruction pages that miss frequently in the STLB have only a few likely successor pages whose reference probability is high.*

3.4 Can Existing dSTLB Prefetchers Help?

Followingly, we measure the effectiveness of the prior dSTLB prefetchers (SP, ASP, DP, MP), presented in Section 2.1, on the iSTLB miss stream. We set the configuration parameters of each dSTLB prefetcher as proposed in the original papers, the prefetched PTEs are placed into a 64-entry Prefetch Buffer (PB), and new prefetch requests are issued on iSTLB misses (Section 2.1). Figure 9 illustrates the performance of the existing dSTLB prefetchers when prefetching for the iSTLB miss stream, including the performance of an idealized scenario; a Perfect STLB for instruction accesses where all iSTLB lookups are hits (Perfect iSTLB). This ideal scenario quantifies the upper bound for the performance improvement by optimizing STLB operation for instruction references.

The ideal scenario (Perfect iSTLB) delivers a geometric speedup of 11.1%. Meanwhile, dSTLB prefetchers provide negligible speedups because they are mainly unable to capture the iSTLB miss patterns. SP improves performance by 1.6% because some of the instruction accesses are sequential but it fails at capturing the complex delta patterns (Figure 5). ASP and DP provide almost no speedup because they use features (PC and distances, respectively) that do not correlate well with the iSTLB misses, thus, their prediction tables experience massive conflicting accesses (96.3% and 93.7%, respectively). Intuitively, we were expecting MP to improve performance since Figure 7 shows that the instruction pages that miss in the STLB have a small number of successors pages. Yet we observe that MP performs poorly, improving performance by a mere 0.2%.

To explain the poor performance of MP and examine its potential for iSTLB prefetching, we evaluate two idealized versions of MP; both versions have an unbounded prediction table that accommodates all instruction pages that miss in the STLB. The two only differ in the number of successor pages they can store per prediction table entry; one version maintains up to two successors, and the other can store any number of successor pages per entry. The unbounded MP with two and infinite successor pages per prediction table entry deliver 7.9% and 10.3% geomean performance, respectively.

There are two important conclusions from this study. First, increasing the number of entries in the prediction table significantly improves MP's performance (from 0.2% speedup with the baseline having a 128-entry prediction table to 7.9% speedup with infinite

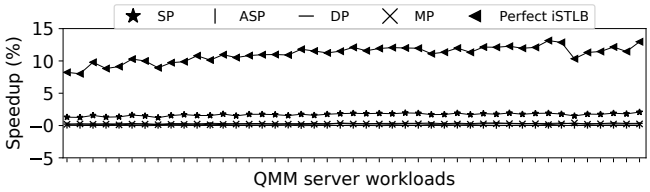

Figure 9: Performance comparison between state-of-the-art dSTLB prefetchers and an ideal scenario.

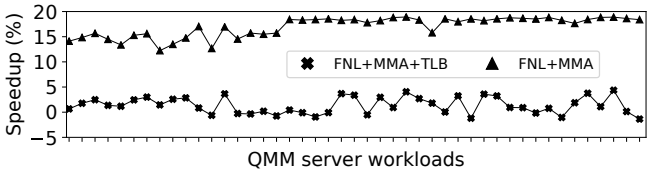

Figure 10: Performance of FNL+MMA with and without taking into account instruction address translation. The baseline system utilizes the next-line I-cache prefetcher.

number of prediction table entries). Our analysis indicates that the replacement policy of MP is one of the reasons why MP does not improve performance with practical prediction table sizes. Since MP uses the LRU policy we conclude that recency is not a useful feature for replacement decisions. Secondly, accommodating multiple successors per page, beyond just two, further increases the speedup from 7.9% to 10.3%, which approaches the ideal of 11.1%.

Finding 4. *A Markov prefetcher has potential for iSTLB prefetching but it requires dynamically building variable length Markov chains out of the iSTLB miss stream in a storage-efficient manner and an effective replacement policy for its prediction table.*

3.5 Instruction Cache Prefetching

Modern I-cache prefetchers may trigger instruction prefetches across page boundaries [22]. When that happens, if the corresponding translation is absent in the TLB, a page walk is triggered. Hence, I-cache prefetchers implicitly work as instruction TLB prefetchers; however, their effectiveness in this role has not been analyzed.

To quantify how effective state-of-the-art I-cache prefetchers are at prefetching for the iSTLB miss stream, we consider the three top performers of the IPC-1 contest: EPI, FNL+MMA, and D-Jolt. IPC-1 infrastructure does not model instruction address translation, *i.e.*, all I-cache prefetches that cross page boundaries are translated without cost. We extend the IPC-1 infrastructure to consider address translation costs (Section 5) and configure the IPC-1 prefetchers to store in the STLB PB the PTEs of the beyond-page-boundaries prefetches, thus providing iSTLB prefetches.

Our analysis indicates that FNL+MMA outperforms the other IPC-1 prefetchers when address translation is taken into account, thus we focus on this I-cache prefetcher. Figure 10 shows the performance of FNL+MMA. Line FNL+MMA+TLB (FNL+MMA) shows the measured performance of the prefetcher when instruction address translation is (is not) considered. When address translation is taken into account (FNL+MMA+TLB), we observe significantly lower speedups than the ones reported in IPC-1. This degradation comes from the instruction prefetches that cross page boundaries

and fail to find the corresponding translation in the TLB hierarchy, thus requiring long-latency page walks to fetch it. Such prefetches hurt the timeliness of the FNL+MMA and delay demand STLB accesses by occupying the page table walker ports, resulting in poor performance. Moreover, we observe only a small reduction (29.6% on average) in demand iSTLB misses because FNL+MMA is unable to cover iSTLB misses due to their poor timeliness in the face of long-latency page walks that require serialized memory accesses. Therefore, state-of-the-art I-cache prefetchers require a smart iSTLB prefetcher to effectively cross page boundaries.

Finding 5. *I-cache prefetchers are mainly ineffective at reducing iSTLB misses due to poor timeliness.*

4 MORRIGAN

To alleviate the instruction address translation performance bottleneck, this paper proposes *Morrigan* (*Irish goddess of destiny*), a composite iSTLB prefetcher. Morrigan is fully legacy-preserving and does not disrupt the existing virtual memory subsystem. Morrigan is also synergistic with I-cache prefetchers as it improves their timeliness when they cross page boundaries.

4.1 Design

Morrigan is inspired by our analysis findings regarding the iSTLB miss behavior (Section 3) and consists of two complementary modules: the *Irregular Instruction TLB Prefetcher (IRIP)* which builds and stores Markov chains out of the iSTLB miss stream, and the *Small Delta Prefetcher (SDP)*, an enhanced sequential prefetcher. Sections 4.1.1 and 4.1.2 present the IRIP and SDP modules while Section 4.2 explains the operation of Morrigan.

4.1.1 Irregular Instruction TLB Prefetcher (IRIP). The IRIP module is designed as a Markov prefetcher since our analysis indicates that a Markov prefetcher has potential for iSTLB prefetching (Finding 4, Section 3.4). Specifically, IRIP is an ensemble of four table-based Markov prefetchers that efficiently build and store variable length Markov chains from the iSTLB miss stream. IRIP also takes into account the variable number of successor pages (Figure 7) of the instruction pages that miss in the STLB. Designing IRIP as a Markov prefetcher with a single prediction table and a fixed number of successors per entry–as the state-of-the-art MP does (Section 2.1)–results in suboptimal performance gains (Section 6.3).

IRIP employs four prediction tables (PRT-S1, PRT-S2, PRT-S4, PRT-S8) that dynamically build a store variable length Markov chains from the iSTLB miss stream. Each prediction table entry stores up to a pre-defined number of successors; PRT-S1, PRT-S2, PRT-S4, and PRT-S8 accommodate instruction pages that have one, two, up to four, and up to eight successor pages, respectively. Each prediction table is realized as a set-associative buffer and stores the virtual page of the missed instruction for indexing, *s* prediction slots, and *s* confidence counters, one per prediction slot. For example, each PRT-S2 entry has *s=2* prediction slots, and *s=2* confidence counters. The only difference in the design of the prediction tables is the number of prediction slots and confidence counters. For simplicity, we illustrate in Figure 11 the design and the operation of PRT-S2.

A naive IRIP design would store the full virtual page number (VPN) in each prediction slot (as the state-of-the-art MP [53] does).

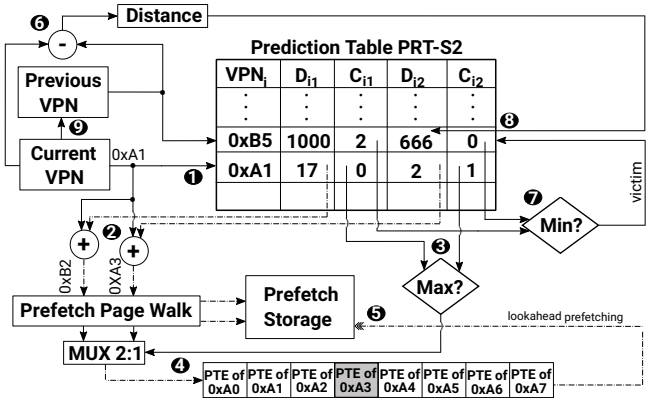

Figure 11: Operation of IRIP on PRT-S2 hits.

However, such a design choice is expensive, storage-wise, since each VPN requires 36 bits of state. To lower this storage cost, IRIP stores the distances between the current and the previous virtual pages that produced an iSTLB miss. This approach lowers the amount of storage for the prediction tables without any performance loss.

The confidence counters associated with the prediction slots are exploited in a two-fold manner: (i) to drive the replacement policy of the prediction slots, *i.e.*, when all the prediction slots are occupied and a new distance has to be placed in one of these slots, the distance with the lowest confidence is replaced, and (ii) the distance with the highest confidence is selected to apply spatial prefetching, leveraging page table locality (Section 2). Specifically, on PRT-S2 hits, IRIP issues one prefetch request per predicted distance of the hit entry. Each prefetch requires a page walk to fetch the corresponding translation (Section 2.1). At the end of a prefetch page walk, page table locality can be exploited to prefetch for free the PTEs that share the cache line with the target PTE. However, prefetching all the free PTEs in all prefetch page walks might harm performance by fetching a lot of inaccurate prefetches. To mitigate this problem, IRIP prefetches cache-line adjacent PTEs only for the distance with the highest confidence.

Figure 11 shows an operational example of PRT-S2, starting with an iSTLB miss for virtual page 0xA1. Initially, a PRT-S2 lookup takes place to determine if there is an entry corresponding to virtual page 0xA1 ❶. In the example, the PRT-S2 lookup experiences a hit. Hence, the predicted distances 17 and 2 of the hit entry are separately summed with the currently missed page (0xA1) to generate new prefetch requests for pages 0xB2 and 0xA3, respectively ❷. In parallel, IRIP finds the predicted distance with the highest confidence counter ❸. Since distance 2 has the highest confidence value, IRIP applies spatial prefetching for the prefetch 0xA3 ❹. Specifically, at the end of the prefetch page walk for 0xA3, IRIP leverages page table locality to also prefetch the PTEs adjacent to the PTE of 0xA3 ❺. To update PRT-S2, IRIP calculates the distance between the currently missed (0xA1) and previously missed (0xB5) virtual pages and stores the outcome into a register ❻. Meanwhile, IRIP finds which of the predicted distances for the previously missed virtual page has the lowest confidence counter ❼. Since distance 666 has the lowest confidence, IRIP replaces it with the current distance for future reuse, while resetting the corresponding confidence counter ❽. Finally, IRIP stores the currently missed virtual

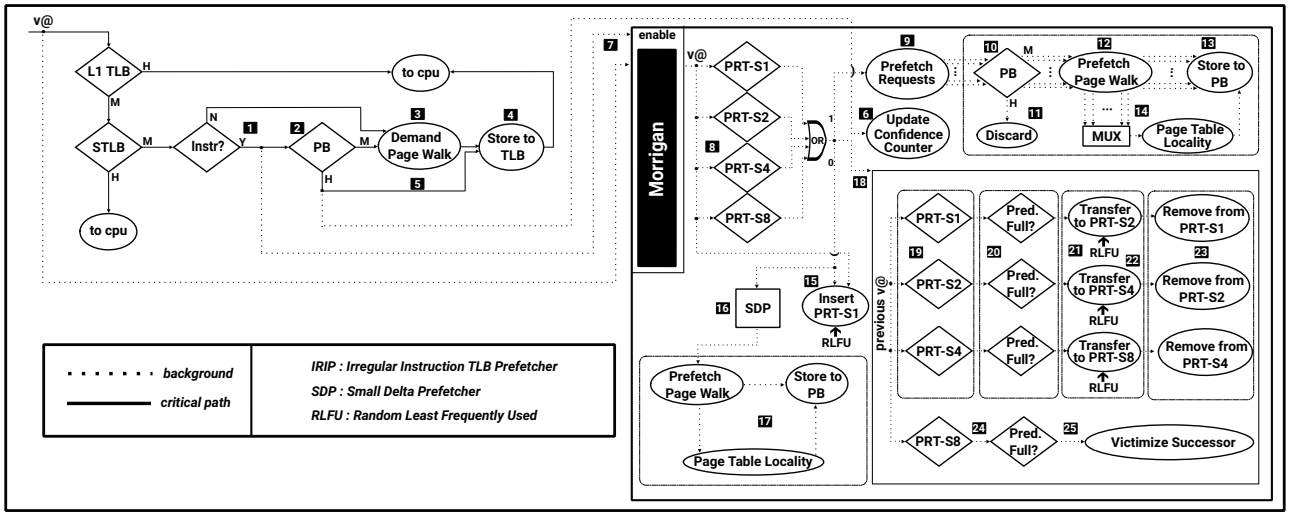

Figure 12: Design and operation of Morrigan. Diamonds indicate decision points, circles are actions.

page into the register holding the previously missed virtual page to be used on the next IRIP operation ❾. Note that when Morrigan operates, steps ❼ and ❽ take place only for PRT-S8; for PRT-S1, PRT-S2, and PRT-S4 Morrigan transfers the entry coupled with the new distance into a prediction table with more prediction slots per entry. Section 4.2 explains the operation of Morrigan in detail.

Updating the confidence counters. When a prefetch is proved to be accurate, *i.e.*, it produces a hit that eliminates a demand page walk, the confidence counter of the corresponding prediction slot is incremented by 1.

Replacement policy. A critical aspect of the IRIP design is the replacement policy of the prediction tables. While previous table-based dSTLB prefetchers, like MP [53], use the LRU policy, we find that LRU does not keep the most useful entries in the prediction tables because it is prone to lose track of important entries (Section 3.4). Our analysis findings indicate that the miss frequency of virtual pages is a good feature to correlate the iSTLB miss stream. Therefore, we employ a frequency-based replacement policy for all the prediction tables of IRIP which (i) maintains a frequency stack of the iSTLB misses to drive the replacement of entries on prediction table conflicts, similar to Least-Frequently-Used (LFU) policy, and (ii) uses a random component that gives recently installed entries, which have not yet accumulated a large number of hits, a chance to persist when a replacement candidate is selected. This policy gives IRIP the ability to adjust to phase-based behavior in workloads. We refer to this policy as Random-Least-Frequently-Used (RLFU). Finally, the complexity of RLFU is similar to LRU.

A problem with a frequency-based replacement policy is that it may be slow to adapt to phase changes in application behavior (*e.g.*, when a page causes frequent iSTLB misses in one phase but not in another). To avoid the associated performance pathologies, Morrigan periodically resets the frequency stack to better identify instruction pages causing the most iSTLB misses in a given interval.

4.1.2 Small Delta Prefetcher (SDP). SDP prefetches the PTE of the virtual page adjacent to the missed virtual page, similar to SP [53]. SDP further exploits page table locality to prefetch all the adjacent PTEs within the target cache line. In this way, SDP captures the majority of the small-strided iSTLB misses (Finding 1, Section 3.3).

For instance, assume an iSTLB miss for page 0xA7. First, SDP issues a prefetch request for page 0xA8 (0xA7+1). After the completion of the prefetch page walk for page 0xA8, SDP prefetches all the PTEs that share the cache line with the PTE of page 0xA8. Note that in this example, fetching the PTEs of pages 0xA7 and 0xA8 requires two separate page walks since the PTE of 0xA7 resides in the last position of a cache line (0xA7 & 0x07) while the PTE of 0xA8 is stored in the first position within another cache line.

4.2 Operation of Morrigan

This section explains the operation of Morrigan, considering the most common case where the iSTLB prefetcher is invoked on iSTLB misses, and the prefetched PTEs are stored into a Prefetch Buffer (PB), as explained in Section 2.1.

Figure 12 illustrates the operation of Morrigan. When an iSTLB miss occurs 1, the requested translation is looked up in the PB 2. On PB misses, a demand page walk is initiated 3 to fetch the corresponding translation into the TLB 4. On PB hits, the demand page walk is avoided and the corresponding translation is transferred from the PB to the TLB 5, and in the background we increment the confidence counter of the prediction table entry that produced the PB hit, if the prefetch was produced by the IRIP module 6.

Morrigan is engaged in case of either PB hit or miss 7. First, Morrigan looks up in parallel all prediction tables (PRT-S1, PRT-S2, PRT-S4, PRT-S8) of IRIP 8 (step ❶ in Figure 11). When there is a hit in one prediction table (there is no duplication of entries in the prediction tables, thus only one hit might occur), Morrigan generates one prefetch per valid prediction slot of the hit entry 9 (step ❷ in Figure 11) of the corresponding prediction table. Before issuing the prefetch requests, Morrigan checks whether the translations already reside in the PB 10. For the prefetches that are already stored in the PB the corresponding requests are discarded 11. For the rest, separate prefetch page walks are initiated to fetch the translations 12. At the end of the prefetch page walks the corresponding PTEs are stored into the PB 13. Then, Morrigan leverages page table

locality to apply lookahead prefetching by fetching in the PB the adjacent PTEs that are transferred together with the prefetched PTE solely for the prefetch with the highest confidence 14 (steps ❸-❺ in Figure 11). When all the prediction tables of IRIP experience a miss, Morrigan has to store the currently missed virtual page in one of the prediction tables. Since this page does not have any valid prediction, it is always placed in the PRT-S1 15 but it might be moved into another prediction table if future STLB misses reveal that it has multiple successor pages. If PRT-S1 is full, Morrigan uses the RLFU policy to find a victim entry. Therefore, on prediction table misses Morrigan is unable to produce prefetch requests based on the IRIP module. At this point, SDP is activated and issues prefetches 16 by exploiting page table locality, which are eventually stored into the PB 17. SDP is enabled only on IRIP misses, thus Morrigan does not loose any potential for performance improvement since it produces new prefetches on every iSTLB miss.

In case of either hit or miss in the prediction tables of IRIP, Morrigan inserts the new predicted distance in one of the prediction slots of the prediction table entry that accommodates the previously missed virtual page 18. If the previously missed page resides in one of PRT-S1, PRT-S2, and PRT-S4 19 and the prediction slots are fully occupied 20, then instead of victimizing one of the prediction slots we simply transfer this entry into the next prediction table that has more prediction slots 21; if it is full, Morrigan uses the RLFU replacement policy to open up space for the transferred entry 22. Next, this entry is removed from the previous prediction table 23. If the previously missed page resides in the PRT-S8 and the prediction slots are fully occupied 24, the new distance is placed into the prediction slot that has the lowest confidence counter 25. Note that in step 19 we do not search all prediction tables to find the previously missed page, but we use a register to store the identifier of the table that stores the previously missed page.

4.3 Additional Aspects

Operation on SMT Cores. Morrigan can operate under SMT colocation by sharing the IRIP module among the threads. To do so, it only requires a different register per thread holding the virtual page that produced the previous iSTLB miss (step 9, Figure 11) to ensure that each thread builds its own Markov chains without intermixing.

Context Switches. The prediction tables of the IRIP module must be flushed on a context switch. Their small sizes ensures that, following a context switch, they are quickly refilled. SDP is stateless; as such, it requires no action on a context switch.

Multiple Page Sizes. Sections 4.1 and 4.2 focus on a single page size to describe the design and operation of Morrigan. This is not a limitation of the design as multiple page sizes are supported without any modification. The page size is known only after address translation, thus, Morrigan can issue two prefetches per request to target 4KB and 2MB pages. Once the page size is known, Morrigan discards the outcome of the prefetch page walk for the mismatched page size. This approach does not add complexity in the design since modern architectures support speculative page walks [67].

Page Replacement Policy and TLB Shootdowns. Morrigan sets the access bit of all prefetched pages since the x86 memory consistency model dictates that all TLB prefetches are obliged to do so [27].

Component	Description
L1 I-TLB	128-entry, 8-way, 1-cycle, 4-entry MSHR
L1 D-TLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L2 TLB	1536-entry, 6-way, 8-cycle, 4-entry MSHR, 1 page walk / cycle
Page Structure Caches	3-level Split PSC, 2-cycle. PML4: 2-entry, fully; PDP: 4-entry, fully; PD: 32-entry, 4-way.
Prefetch Buffer (PB)	64-entry, fully assoc, 2-cycle
L1 I-Cache	32KB, 8-way, 4-cycle, 8-entry MSHR, next line prefetcher
L1 D-Cache	32KB, 8-way, 4-cycle, 8-entry MSHR, next line prefetcher
L2 Cache	512KB, 8-way, 8-cycle, 32-entry MSHR, SPP [57]
LLC (per core)	2MB, 16-way, 10-cycle, 64-entry MSHR
DRAM	tRP=tRCD=tCAS=12, 12.8 GB/s
Branch Predictor	hashed perceptron

Table 1: System configuration.

Therefore, Morrigan does not complicate TLB shootdowns because the information about the prefetched instruction PTEs is conveyed to the OS as usual. Regarding the impact on the page replacement policy, a prefetch is harmful for the page replacement policy if it is evicted from the STLB PB without providing any hit and does not belong to the active footprint of the application. Morrigan issues prefetches based on the control-flow behavior and does not permit faulting prefetches, thus the probability of negatively affecting the page replacement policy is negligible. To annihilate this probability, Morrigan could issue a correcting page walk to reset the access bit of the PTEs that are evicted from the PB without providing any hit. These correcting page walks could be issued when the TLB MSHR is not full to avoid delaying any other page walk.

Synergy with I-Cache Prefetching. Morrigan is complementary to I-cache prefetchers because it prefetches instruction PTEs, thus improving the timeliness of I-cache prefetches that go beyond page boundaries by avoiding long-latency page walks (Section 3.5). Section 6.5 quantifies the performance gains of using Morrigan to improve the timeliness of a state-of-the-art I-cache prefetcher.

Different Architectures. We focus on x86 architectures; however, architectural support for virtual memory used in x86 architectures [23, 25] is similar to other architectures (*e.g.*, ARM [8] and RISC-V [11]). Thus, Morrigan would be applicable to these architectures.

Page Tables. Morrigan is compatible with a 5-level radix tree page table [12], and may deliver higher performance gains because the extra page table level might increase the page walk latency. If a hashed page table [77, 82] is used, Morrigan would operate the same since hashed page tables preserve page table locality.

TLB Prefetching Strategy. TLB prefetching schemes are typically engaged on STLB misses and store the prefetched PTEs into a PB (Section 2.1). Our analysis indicates that these two strategies have a positive effect on performance. Nonetheless, Morrigan could be also activated on STLB hits and prefetch directly into the STLB.

5 METHODOLOGY

Simulation Infrastructure. To evaluate Morrigan, we use the latest version of ChampSim [15], a detailed simulator that models a 4-wide out-of-order processor. We extend ChampSim to simulate a realistic x86 page table walker, modeling the variable latency cost of page walks and also the variable number of memory references

they require to complete. Specifically, we added a 4-level page table, a page table walker, and a 3-level split PSC. The page table walker supports up to 4 concurrent TLB misses, similar to Skylake microarchitecture [27], while one page walk can be initiated per cycle. Finally, our baseline uses the next-line I-cache prefetcher but we also consider the I-cache prefetchers from IPC1 [22] in Sections 3.5 and 6.5. Table 1 summarizes our experimental setup.

We also extended ChampSim to simulate a dual-threaded SMT core to evaluate our proposal under workload colocation. Every cycle, a different thread fetches one basic block of instructions. Our SMT model fully accounts for the contention due to colocation in all shared microarchitectural structures (TLBs, PSCs, cache hierarchy).

Our work focuses on 4KB pages, similar to prior work using the QMM workloads [61]. So why not use huge pages to mitigate the address translation overhead? Although profitable when the application exhibits high locality and the system is not fragmented, huge pages are not a stop-gap solution to the address translation bottleneck for both data and code accesses. In practise, using huge pages for data and code potentially hurts performance in datacenters and exposes security vulnerabilities, as we explain below. Furthermore, the performance of legacy systems and cloud applications that continue to use 4KB pages still matters for their users.

Huge pages have been shown to introduce performance pathologies [29, 59, 83], particularly for servers. Another problem is the lack of flexibility in memory management with huge pages compared to standard 4KB pages [24, 50, 70]. Specifically, huge pages require memory contiguity and defragmentation that is not guaranteed in datacenters due to high uptimes and the fact that datacenters handle thousands of diverse applications [24, 54, 81]. Indeed, [59] demonstrates that memory defragmentation can result in tail latency spikes and performance variability, both of which might negatively impact the performance of datacenter applications. Moreover, a recent work [24] shows that transparent 2MB support for data pages is not adequate anymore and there is need for creating transparent support for 1GB pages. Finally, [48] reveals that huge pages can harm the performance of NUMA machines; this problem might be amplified with the advent of heterogeneous memories where the OSes have to migrate data between fast and slow tiers of memory.

In addition to the above, concurrently supporting multiple page sizes is a complex problem; this is the reason why Linux has support for transparent 2MB pages only for data, which, in fact, took a long time to be properly implemented [27]. Today, Linux does not have support for 2MB transparent huge pages for code blocks. The only way to map executable files onto huge pages in Linux is to use *libhugetlbfs* [4]. However, *libhugetlbfs* does not provide automatic and transparent support for huge page code mappings since it requires shaping the text layout in the application's address space [44]. Indeed, a recent work [64] reveals that (i) mapping the *.text* section of server applications onto huge pages provides performance degradation since it puts pressure on the limited number of L1 I-TLB entries that can accommodate huge pages, and (ii) mapping too many huge pages using *libhugetlbfs* in production machines makes the Linux kernel misbehave as it becomes overwhelmed by the need to relocate physical pages to satisfy requests for huge pages.

Another concern with mapping code in huge pages is that doing so represents a security risk. Modern systems use Address Space Layout Randomization (ASLR) to obstruct certain security attacks

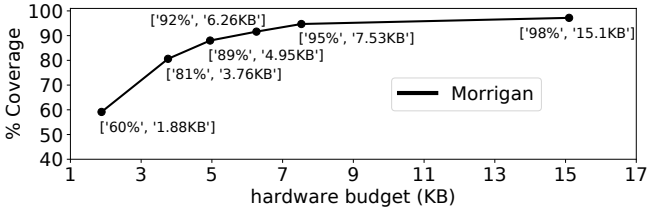

Figure 13: Miss coverage of Morrigan for various budgets.

by making it difficult for an adversary to predict target addresses. Prior work has shown that using huge pages for code significantly diminishes the effectiveness of ASLR [34, 35, 46, 75]. Another security risk is the *iTLB multihit* [16] vulnerability that arises when huge pages are used for code. Specifically, when an instruction fetch hits multiple entries in the I-TLB it may incur a machine check error. To mitigate this issue, cloud providers such as Microsoft Azure and Amazon force all executable instruction pages to be mapped into 4KB pages [13, 18–21], removing the possibility of multiple hits.

For these reasons, we focus our evaluation on 4KB pages but Morrigan is entirely compatible with larger page sizes (Section 4.2).

Workloads. We use a set of server workloads provided by Qualcomm (QMM) for the CVP-1 [14] and IPC-1 [22] contests that were previously used in other TLB-related research works [61, 79]. Workloads with an iSTLB MPKI of at least 0.5 are considered instruction TLB intensive, thus our evaluation considers 45 instruction TLB intensive QMM server workloads. Our simulations use 50 million warmup instructions, then 100 million instructions are executed to measure the experimental results, similar to prior work [61].

We also analyze the SPEC CPU 2006 [2] and SPEC CPU 2017 [10] benchmark suites, but we find that these workloads have an iSTLB MPKI of 0.5 or less, so they are not considered in our evaluation. However, we use the SPEC CPU workloads in Section 3 to show that we are consistent with the conclusions of previous works [54, 62].

Finally, datacenters colocate applications on SMT cores for better CPU and memory utilization [54, 80]. To consider colocation, we simulate a dual-threaded SMT core executing two different QMM workloads. Our evaluation (Section 6.6) considers 50 randomly chosen pairs of QMM workloads.

6 EVALUATION

6.1 IRIP Module

The IRIP module of Morrigan is an ensemble of table-based hardware Markov prefetchers. Therefore, the effectiveness of Morrigan directly depends on the number entries in the prediction tables (PRT-S1, PRT-S2, PRT-S4, PRT-S8) of the IRIP module. Each prediction table entry requires 16 bits for storing a partial tag of the virtual page for indexing, 15 bits per predicted distance of the prediction slots, and a 2-bit saturating counter per predicted distance (Section 4). Note that Sections 6.1.1 and 6.1.2 consider fully associative prediction tables and a 64-entry Prefetch Buffer (PB); Section 6.1.3 examines different prediction table associativities and PB sizes.

6.1.1 Miss Coverage. Figure 13 presents the miss coverage of Morrigan across all QMM workloads as a function of different storage budgets. Starting with small storage budgets, we observe a large increase in the miss coverage of Morrigan as the storage budget

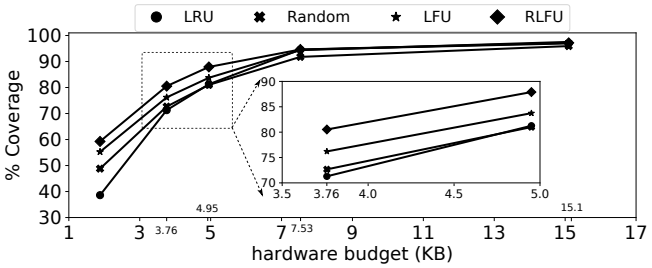

Figure 14: Miss coverage of Morrigan when the prediction tables of the IRIP module use different replacement policies for various storage budgets.

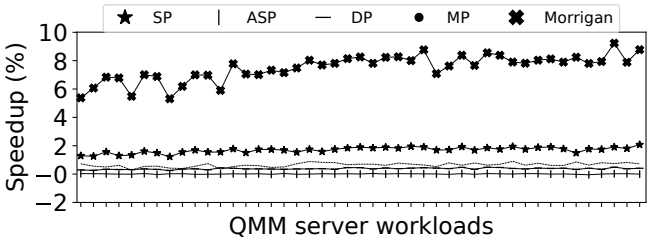

Figure 15: Performance comparison between Morrigan and the state-of-the-art data STLB (dSTLB) prefetchers.

increases. However, after 5KBs, the miss coverage begins to plateau. Going beyond 7.5KB of storage budget provides negligible benefits.

6.1.2 Replacement Policy. The prediction tables of the IRIP module use the RLFU policy (Section 4). To highlight the benefits of RLFU we compare against the following alternatives: (i) LRU policy, (ii) Random policy, and (iii) LFU policy that replaces the least frequently accessed entry. Figure 14 shows the miss coverage of Morrigan when the IRIP module leverages the above explained replacement policies as a function of different budgets, similar to Section 6.1.1.

Looking at Figure 14, we observe that the RLFU replacement policy provides significantly higher miss coverage than the other replacement policies when the prediction tables of the IRIP module accommodate a small number of entries. As the size of the prediction tables increases, the miss coverage gap between RLFU and the other policies shrinks because the prediction tables can store the majority of the virtual pages that produce iSTLB misses (Sections 3.3 and 3.4), thus making the replacement policy irrelevant.

Considering Morrigan with 3.76KB of storage budget, Figure 14 reveals that the LRU and Random replacement policies provide the lowest miss coverage since the former evicts useful entries based on their recency position and the latter randomly selects victims without any insight. The LFU replacement policy provides higher coverage than LRU and Random replacement policies, highlighting that the iSTLB miss stream correlates well with the miss frequency of the virtual pages. Finally, the RLFU policy improves miss coverage over the LFU policy by 4.9%. This happens because RLFU randomly replaces one of the least recently used entries, acting like a second-chance policy for not yet frequently accessed entries.

6.1.3 Configuring IRIP. Taking into account the results of Sections 6.1.1 and 6.1.2, we conclude that there is a cost-performance trade-off in the design space of Morrigan. For the rest of the paper, we focus on the configuration of Morrigan with 3.76KB of storage budget, which achieves 81% miss coverage (Figure 13). We select this configuration because it represents an attractive point in terms of miss coverage and required storage budget.

Using the above selected version of Morrigan, we evaluated different capacities and associativities for the prediction tables of IRIP. Empirically, we found the following preferred configuration: 128-entry (32 ways) PRT-S1, 128-entry (32 ways) PRT-S2, 128-entry (32 ways) PRT-S4, and a 64-entry (16 ways) PRT-S8. Among the prediction tables, PRT-S8 is the smallest one because the number of instruction pages that have more than 4 and up to 8 successors is lower than the number of instruction pages that have 1, 2, and up to 4 successor pages (Section 3.3) and the probability of accessing a non frequent successor page is relatively low (Figure 8). Finally, the empirically selected configuration provides a miss coverage of 76% (5% lower than the version with fully associative prediction tables).

Regarding the PB size, we consider a 64-entry PB because a PB with 16 or 32 entries provides rather poor miss coverage compared to the 64-entry PB (4%-12% reduction), whereas a 128-entry PB increases coverage by 2% compared to the 64-entry PB.

6.2 ISO-Comparison with dSTLB Prefetchers

This section compares Morrigan with the state-of-the-art dSTLB prefetchers (Section 2.1) that are configured to prefetch for the iSTLB miss stream, similar to Section 3.4. To make a fair comparison, we set the configuration parameters of these prefetchers in such a way that they match the storage budget of Morrigan (3.76KB).

Performance Comparison. Figure 15 shows the performance comparison between Morrigan and the dSTLB prefetchers. The baseline considers the system without STLB prefetching. SP, DP, ASP, MP, and Morrigan provide a geometric speedup of 1.6%, 0.1%, 0.4%, 0.7%, and 7.6%, respectively. Morrigan significantly outperforms all previously proposed dSTLB prefetchers because the QMM workloads exhibit highly complex patterns that the dSTLB prefetchers are unable to capture. Specifically, SP captures only the sequential patterns, DP and ASP experience massive conflicts in their prediction tables, and MP uses the LRU policy that fails at keeping in the prediction table the most useful instruction pages (Section 3.4).

In terms of PB hits provided by the two modules of Morrigan (IRIP and SDP), we measured that 93% of the prefetches that hit in the PB were triggered by the IRIP, while the remaining 7% by SDP.

Cost of Prefetching & Analysis. Figure 16 presents the distribution of the normalized number of memory references triggered by demand and prefetch page walks for Morrigan and the prior dSTLB prefetchers. For the purposes of this study, the term *memory reference* refers to a page walk reference that is served by the memory hierarchy (L1, L2, LLC, DRAM). Note that (i) we take into account cache locality in page walks (Section 5), and (ii) a page walk memory reference is triggered only for references that miss in the PSC, which we also model. The normalization factor, 100% in Figure 16, is the number of memory references due to demand page walks without STLB prefetching.

SP, ASP, DP, MP, and Morrigan reduce the memory references due to demand page walks by 11%, 1%, 2%, 8%, and 69%, respectively. Regarding the prefetch page walks, SP, ASP, DP, MP, and Morrigan

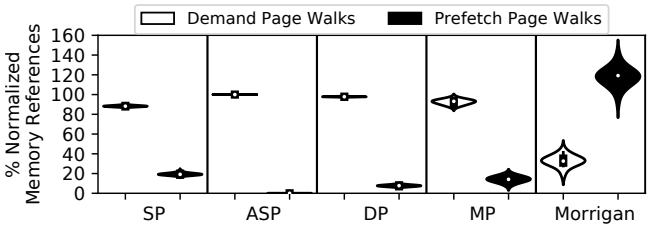

Figure 16: Normalized page walk memory references.

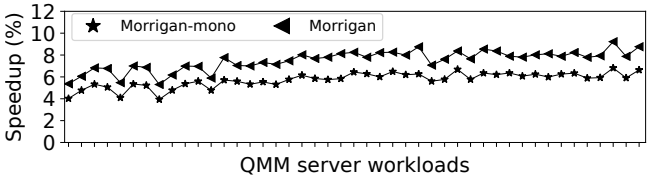

Figure 17: Performance of Morrigan when the IRIP module uses an ensemble of four tables (Morrigan) versus a single-table design (Morrigan-mono).

trigger 20%, 1%, 6%, 7%, and 117% additional memory references due to prefetch page walks with respect to the baseline, respectively.

The prior dSTLB prefetchers do not reduce demand page walk memory references for instructions, so they provide negligible performance improvements, as Figure 15 shows. They also introduce only a small number of memory references for prefetch page walks because (i) SP issues only one prefetch per iSTLB miss, (ii) ASP and DP experience a lot of conflicting accesses in their prediction tables which does not allow them to produce prefetch requests, and (iii) MP leverages the LRU replacement policy that fails at keeping the most useful entries in the prediction table; on prediction table lookup misses, no prefetches are issued.

While Morrigan does generate more memory references for prefetch page walks than the existing dSTLB prefetchers, it achieves much higher coverage than the prior designs. Indeed, Morrigan reduces the memory references for demand page walks by 69% due to its high coverage. The vast majority of memory references due to prefetch page walks are caused by the IRIP module since the SDP module (i) issues only one prefetch at a time that requires a prefetch page walk, and (ii) is enabled only when the IRIP module is unable to issue prefetch requests (Section 4.1.2). However, the demand page walks are responsible for the iSTLB performance bottleneck since they take place on the critical path of execution causing unavoidable pipeline stalls, while the prefetch page walks are performed in the background without stalling the pipeline execution.

Finally, we examine the fraction of prefetch page walk memory references served by each level of the memory hierarchy. We find that 20%, 25%, 45%, and 10% of Morrigan's prefetch page walk memory references are served by L1, L2, LLC, and DRAM, respectively. Hence, the large reduction of demand page walk memory references that Morrigan achieves, lowers the instruction address translation overhead, thus providing significant performance gains.

6.3 Comparing Different IRIP Designs

This section highlights the benefits of using multiple prediction tables with different number of prediction slots per entry for the IRIP module over the state-of-the-art approach that uses a single prediction table with fixed number of successors per entry. To do so, we implement Morrigan-*mono* whose operation is identical to Morrigan but its IRIP module leverages a single prediction table with a fixed number of successors per entry, as the state-of-the-art MP [53] does. We opt to provide an ISO-storage comparison between Morrigan and Morrigan-mono, so we configure the IRIP module of Morrigan-mono with a 203-entry prediction table with 8 prediction slots per entry,[3] and a 2-bit confidence counter per prediction slot to match the storage and the operation of Morrigan's IRIP module.

Figure 17 reveals that Morrigan outperforms Morrigan-mono (1.9% on average) across all the QMM server workloads. We observe this behavior because Morrigan makes better use of the available storage budget, hence tracking a much larger effective working set. Whereas Morrigan dynamically tracks the required number of prediction slots per instruction page and enables efficient transferring of entries between the prediction tables, Morrigan-mono accommodates eight prediction slots per prediction table entry. Specifically, Morrigan-mono tracks 203 entries and Morrigan effectively tracks 448 entries (128*3+64). Indeed, we find that Morrigan-mono requires 6.9KB of storage to match the performance of Morrigan having a 3.76KB storage budget.

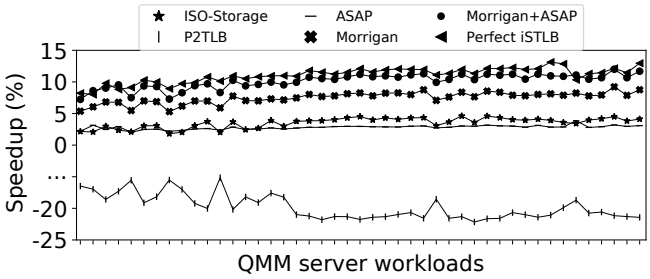

Figure 18: Performance comparison with other approaches that improve TLB performance.

6.4 Comparison with Other Approaches

Figure 18 compares Morrigan with other approaches that improve TLB performance and the ideal case of the Perfect STLB for instruction references (Perfect iSTLB), as explained in Section 3.4.

ISO-Storage Comparison. We compare Morrigan against a system that does not apply STLB prefetching but for fairness it is enhanced with an enlarged STLB. Specifically, STLB is augmented with 388 additional entries to match the storage budget of Morrigan (including the PB) without affecting its access time. Figure 18 shows that Morrigan outperforms the this scenario by 4.1%.

Prefetching into TLB. Prior STLB prefetchers [38, 53] and patents [26, 52] use a PB to store the prefetched PTEs. Figure 18 shows that placing the prefetches of Morrigan directly into the STLB (P2TLB) provides a 18.9% performance degradation because it causes STLB pollution when the prefetches are inaccurate. Our results are consistent with prior work [27, 38, 53] stating that prefetching directly into the STLB causes pollution and performance degradation.

[3]The IRIP module of Morrigan-mono is enhanced with 8 prediction slots per prediction table entry to make a fair comparison with the IRIP module of Morrigan since PRT-S8 can store up to 8 predictions per entry.

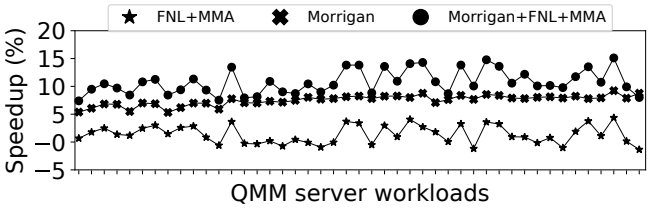

Figure 19: Impact of Morrigan on I-cache prefetching.

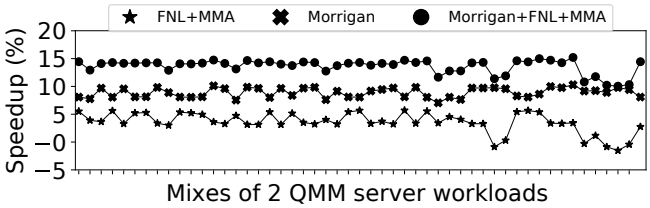

Figure 20: Performance of Morrigan under SMT colocation.

Prefetched Address Translation (ASAP) [60]. ASAP is a microarchitectural scheme which lowers the latency cost of page walks by prefetching deeper levels of the radix tree page table, avoiding serialized memory references on PSC misses. Figure 18 depicts that Morrigan outperforms ASAP by 4.8% (on average) because the PSCs experience high hit rates for the QMM workloads, thus limiting the performance gains of ASAP. We find that, on average, 1.4 memory references are required per page walk due to PSC misses. The leaf page table level always triggers a memory reference, hence only 0.4 memory references (on average) are triggered due to the other 3 page table levels. Therefore, the high PSC hit rate of the QMM workloads hurts the effectiveness of ASAP.

Combining Morrigan with ASAP. STLB prefetching is orthogonal to techniques that aim at lowering page walk latency. Consequently, it is natural to combine Morrigan with ASAP. The core idea is that ASAP lowers the page walk latency, thus it can be further used to accelerate the prefetch page walks of Morrigan. Figure 18 illustrates that combining Morrigan with ASAP improves geometric mean performance by 10.1%, approaching the ideal performance results (Perfect iSTLB) for most QMM workloads. We observe such behavior because ASAP improves the timeliness of Morrigan's prefetches by accelerating the corresponding prefetch page walks.

6.5 Synergy with I-Cache Prefetching

This section demonstrates that Morrigan is synergistic with I-cache prefetching. Recall that our baseline includes next-line I-cache prefetching that does not cross page boundaries (Section 5). However, modern I-cache prefetchers cross page boundaries, as explained in Section 3.5. This section studies a state-of-the-art I-cache prefetcher, FNL+MMA, which crosses page boundaries, because it provides the highest performance among the IPC1 prefetchers [22] when instruction address translation is considered (Section 3.5).

Figure 19 shows the performance results of (i) FNL+MMA, (ii) Morrigan with next-line I-cache prefetcher (Morrigan), as evaluated in all previous sections, and (iii) Morrigan when combined with FNL+MMA (Morrigan+FNL+MMA). The baseline corresponds to a system with a next-line I-cache prefetcher and no STLB prefetching.

Overall, FNL+MMA, Morrigan, and Morrigan+FNL+MMA provide a geometric speedup of 1.2%, 7.6%, and 10.9%, respectively. We observe that the performance of Morrigan+FNL+MMA exceeds the sum of the benefits of the individual prefetchers. The reason why the total is greater than the sum of its parts is that Morrigan improves the timeliness of FNL+MMA. Specifically, 51.7% of the beyond-page-boundary prefetches of FNL+MMA that require a page walk hit in the PB of Morrigan+FNL+MMA, thus improving the timelines of the respective instruction prefetches. The main takeaway is that Morrigan is synergistic with I-cache prefetching.

6.6 Workload Colocation in SMT Cores

This section quantifies the performance of Morrigan under SMT colocation (Section 5). For this experiment, we double the size of the prediction tables of the IRIP module since Morrigan has to separately build Markov chains for two threads in the same prediction tables. This increases the storage budget of Morrigan to 7.5KBs. We compare the same set of prefetchers as in Section 6.5: Morrigan, FNL+MMA, and Morrigan+FNL+MMA. The baseline corresponds to a system with the next-line I-cache prefetcher and no STLB prefetching. Figure 20 presents the performance results.

The overall trends are consistent with Section 6.5; however, the absolute performance gains are higher under SMT, since colocating two QMM workloads increases the pressure on the cache and the TLB hierarchy, providing higher opportunity for prefetching. Morrigan and FNL+MMA provide speedups of 8.9% and 3.4%, respectively. Morrigan+FNL+MMA improves performance by 13.7% because it (i) eliminates the majority of the observed iSTLB misses, and (ii) improves the timeliness of the FNL+MMA (Section 6.5).

If the size of the prediction tables of the IRIP module is not doubled in the SMT setup, Morrigan and Morrigan+FNL+MMA improve performance by 6.4% and 11.1%, on average, respectively.

7 RELATED WORK

Increasing TLB reach. Prior work increases the effective capacity of TLBs by coalescing virtually and physically contiguous PTEs into a single TLB entry [68, 69]. These approaches are (i) limited by coalescing opportunities exposed by the OS since physical contiguity is not guaranteed, and (ii) susceptible to security issues when applied for code pages because an adversary could exploit this contiguity to attack the system. In addition, Bhattacharjee *et al.* [37] propose a shared among cores last-level TLB that exploits page table locality only on demand page walks. Instead, Morrigan improves the performance of private-per-core TLBs via microarchitectural prefetching, exploits page table locality for both demand and prefetch walks, and does not disrupt the existing virtual memory subsystem.

Speculative address translation. Speculation-based approaches [33, 49, 70] predict the address translation of a non TLB-resident page, the processor continues executing instructions speculatively, and a page walk is initiated in the background to validate whether the predicted translation is correct. In case of valid speculation, the verification page walk overlaps useful work, hiding its latency cost. Speculation-based approaches are affected by the system state since they rely on explicit virtual and physical contiguity to predict the missing address translations which is not guaranteed in systems today. Morrigan exploits only virtual contiguity which comes at zero cost and is independent of the system state.

Mitigating TLB miss latency. Improving the performance of the MMU-Caches [32, 36] is an effective way to reduce the latency penalty of frequent TLB misses. POM-TLB [73] is a large die-stacked L3 TLB that reduces the page walk memory references to just one reference. DVMT [28] allows the application to define the appropriate page table format for an address space portion, reducing the required page walk memory references. Alternatively, hashed page tables [51, 77, 82] have been proposed to resolve TLB misses faster than the conventional radix tree page tables. Morrigan is complimentary to these approaches as it eliminates iSTLB misses via prefetching PTEs ahead of demand STLB accesses.

TLB management. Typically TLBs employ a variation of the LRU replacement policy. Mirbagher-Ajorpaz *et al.* [61] propose a new predictive replacement policy for the STLB. However, STLB replacement policies aim at keeping in the STLB the most useful PTEs while STLB prefetchers proactively fetch the PTE(s) that would be requested by the next memory access(es). Elnawawy *et al.* [45] identify heterogeneity in TLB behavior of data-intensive applications, *i.e.*, a few data pages have high reuse but poor temporal locality In response, they propose Diligent TLBs, a scheme that pins in the STLB such delinquent data pages. Although effective for data pages, our analysis (Section 3.3) indicates that [45] needs to pin hundreds of instruction pages in the STLB to achieve significant MPKI reductions for instruction accesses; such extensive pinning raises the STLB MPKI of data pages.

Software schemes. Compile-time optimization approaches [44, 64] modify *hugetlbfs* to place only hot functions in huge pages. Moreover, OS schemes using superpages [43, 59, 83] map small code regions into superpages via superpage promotion and page table sharing. Recency-based TLB Preloading [74] builds a recency stack of PTEs in the page table to derive prefetches based on past access patterns. There are two major differences between Morrigan and [74]. First, Morrigan is a microarchitectural prefetcher that does not imply any page table or software modification while [74] is a software prefetching scheme that modifies the page table. Secondly, Morrigan considers access frequency for prefetching while [74] relies on recency to drive prefetching – a feature that does not correlate well with iSTLB prefetching (Section 3.4).

Instruction cache prefetching. Numerous I-cache prefetchers have been proposed in recent literature [22, 72]. Although effective for capturing the L1 I-cache miss stream, these prefetchers fall short at prefetching for the iSTLB miss stream because they are tuned for short prefetch distances and low latencies, as the prefetched blocks are often found in the L2 or the LLC [47]. In contrast, iSTLB misses require larger prefetch distances and incur higher latency, caused by the serialized accesses to the memory hierarchy due to page walks. Reinman et al. [72] propose FDIP, a prefetching scheme that speculatively identifies instruction blocks that would potentially cause an L1 I-cache miss in the future and prefetch them from the lower level caches. Intuitively, the impact of FDIP on tolerating iSTLB misses is relatively small since it would just bring instruction PTEs into the STLB when the prefetched instruction blocks reside in memory pages different from the page where the initially missed instruction block resides. Finally, FDIP would pollute the STLB when the prefetched PTEs are inaccurate.

8 CONCLUSIONS

This paper provides evidence that instruction address translation is a significant performance bottleneck in server applications. To mitigate this bottleneck, this paper proposes *Morrigan*, a composite instruction TLB prefetcher whose design is based on new reuse and locality insights of instruction TLB misses. Morrigan consists of two complimentary prefetch engines; (i) the Irregular Instruction TLB Prefetcher (IRIP), an ensemble of table-based hardware Markov prefetchers that build and store variable length Markov chains out of the instruction TLB miss stream while leveraging a new frequency-based replacement policy to manage their internal state, and (ii) the Small Delta Prefetcher (SDP), an enhanced sequential prefetcher that is engaged only when the IRIP module of Morrigan is unable to issue prefetch requests. Considering an extensive set of industrial server workloads, this paper demonstrates that Morrigan provides large performance enhancements by saving the majority of the instruction TLB misses while significantly reducing the references to the memory hierarchy due to page walks.

9 ACKNOWLEDGEMENTS

The authors are profoundly grateful to (i) the anonymous reviewers for their constructive feedback, and (ii) the anonymous shepherd for his/her valuable comments that significantly improved the quality of the paper. This work is partially supported by the Spanish Ministry of Science and Technology through the PID2019-107255GB project, the Generalitat de Catalunya (contract 2017-SGR-1414), the NSF grant CCF-1912617, the Semiconductor Research Corporation grant 2936.001, and generous gifts from Intel Labs. Georgios Vavouliotis has been supported by the Spanish Ministry of Economy, Industry and Competitiveness and the European Social Fund under the FPI fellowship No. PRE2018-087046. Marc Casas has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Ramon y Cajal fellowship No. RYC-2017-23269.

A ARTIFACT APPENDIX

A.1 Abstract

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

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

- **Program:** Memory traces of server applications provided by Qualcomm for CVP-1 [14] and IPC-1 [22].
- **Compilation:** gcc.
- **Metrics:** Performance improvement.
- **Output:** We provide scripts that generate the most important evaluation figures (Figures 15 and 18).
- **Experiments:** We provide scripts that submit the required jobs. The only requirement is a SLURM manager.
- **How much disk space required (approximately)?:** 3.1GB.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes.

- **How much time is needed to complete experiments (approximately)?:** 1-3 hours, depending on the machine.
- **Workflow framework used?:** SLURM for job management.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5496052

A.3 Description

A.3.1 How to access. Our artifact is available at https://doi.org/10.5281/zenodo.5496052.

A.3.2 Hardware dependencies. Any hardware capable of compiling ChampSim [15].

A.3.3 Software dependencies. SLURM manager for job management and python pandas to generate the evaluation figures.

A.3.4 Data sets. We use memory traces provided by Qualcomm for CVP-1 [14] and IPC-1 [22].

A.4 Installation

Download the artifact from https://doi.org/10.5281/zenodo.5496052. Then, extract it by executing the following command.

```
tar xvzf paper-47-AE.zip
```

A.5 Experiment workflow

To reproduce the most important evaluation results, take the following steps.

- `cd paper-47-AE/ChampSim-SC`
- set the paths in `run_champsim.sh` (line 2)
- set the paths in `generate_binary.sh` (lines 2, 4, 8)
- compile the binaries by executing `bash micro_ae.sh`
- `cd /path-to/paper-47-AE`
- set the paths in `submit.sh` (lines 52, 55)
- execute `bash launch.sh` (it submits the jobs in batches; around 30 minutes to submit all jobs)

To check the status of the jobs, use the following command.

```
watch squeue -<user-name>
```

It takes 1-3 hours to finish all jobs, depending on the machine and the number of jobs that can be launched in parallel.

A.6 Evaluation and expected results

When all jobs are finished, generate the evaluation figures by executing the following commands.

```
cd /path-to/paper-47-AE/ChampSim-SC
bash generate_figures.sh
```

After executing these commands, there will be two figures under `/path-to/paper-47-AE/ChampSim-SC/Figures` with names `speedup_sota.pdf` and `plot_other_techniques.pdf`.

The collected figures for each separate study should match the expected figures in the directory /path-to/paper-47-AE/ChampSim-SC/Figures. Practically, figure `speedup_sota.pdf` should match figure speedup_sota_EXPECTED.pdf and figure plot_other_techniques.pdf should match figure plot_other_techniques_EXPECTED.pdf.

A.7 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] 1996. Before Memory Was Virtual. http://denninginstitute.com/pjd/PUBS/bvm.pdf.
[2] 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/.
[3] 2008. The Locality Principle. https://denninginstitute.com/pjd/PUBS/ENC/locality08.pdf.
[4] 2010. Huge Pages and libhugetlbfs. https://lwn.net/Articles/374424/.
[5] 2012. Sandy Bridge - Microarchitectures. https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client).
[6] 2013. Haswell - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client).
[7] 2014. Using Intel VTune Amplifier XE to tune software on the 6th generation Intel Core processor family. https://software.intel.com/content/dam/develop/external/us/en/documents/using-intel-vtune-amplifier-xe-on-6th-generation-intel-core-processors-1-0.pdf.
[8] 2015-2021. ARMv8 Architecture Reference Manual. ARM. https://developer.arm.com/documentation/ddi0553/bp.
[9] 2017. Coffee Lake - Microarchitectures. https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake.
[10] 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/.
[11] 2017. The RISC-V Instruction Set Manual. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.
[12] 2018. Intel. 5-Level Paging and 5-Level EPT. https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html.
[13] 2020. A Principled Technologies report: Hands-on testing. Real-world results. https://www.principledtechnologies.com/Intel/Xeon-8272CL-Microsoft-Azure-WordPress-science-0920.pdf.
[14] 2020. Championship Value Prediction (CVP). https://www.microarch.org/cvp1/.
[15] 2020. ChampSim. https://crc2.ece.tamu.edu/. [Online].
[16] 2020. iTLB multihit. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/multihit.html.
[17] 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.
[18] 2020. perl-Net-Amazon. https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon/hw_info.log.
[19] 2020. perl-Net-Amazon-EC2. https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon-EC2/hw_info.log.
[20] 2020. perl-Net-Amazon-S3. https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon-S3/hw_info.log.
[21] 2020. Public clouds and vulnerable CPUs: are we secure? https://archive.fosdem.org/2020/schedule/event/vai_pubic_clouds_and_vulnerable_cpus/attachments/slides/3650/export/events/attachments/vai_pubic_clouds_and_vulnerable_cpus/slides/3650/FOSDEM2020_vkuznets.pdf.
[22] 2020. The 1st Instruction Prefetching Championship. https://research.ece.ncsu.edu/ipc/.
[23] 2021. AMD64 Architecture Programmer Manual(Volume 2). https://www.amd.com/system/files/TechDocs/24593.pdf.
[24] 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. https://www.usenix.org/system/files/osdi21-hunter.pdf.
[25] 2021. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm.
[26] Abishek Bhattacharjee. 2010. Inter-core cooperative TLB prefetchers. https://patents.google.com/patent/US8880844B1/en.
[27] Abishek Bhattacharjee. 2018. 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.
[28] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/3079856.3080209
[29] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. 2011. Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure. In *Proceedings of the Middleware 2011 Industry Track Workshop (Middleware '11)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2090181.2090185

[30] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. 2018. Memory Hierarchy for Web Search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 643–656. https://doi.org/10.1109/HPCA.2018.00061

[31] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44, 5 (May 1995), 609–623. https://doi.org/10.1109/12.381947

[32] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[33] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[34] M. Bazm, M. Lacoste, M. Sudholt, and J. Menaud. 2017. Side-channels beyond the cloud edge: New isolation threats and solutions. In *2017 1st Cyber Security in Networking Conference (CSNet)*. 1–8. https://doi.org/10.1109/CSNET.2017.8241986

[35] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Sudholt, and Jean-Marc Menaud. 2017. Side Channels in the Cloud: Isolation Challenges, Attacks, and Countermeasures. (March 2017). https://hal.inria.fr/hal-01591808 working paper or preprint.

[36] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[37] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 62–63. http://dl.acm.org/citation.cfm?id=2014698.2014896

[38] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 359–370. https://doi.org/10.1145/1736020.1736060

[39] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[40] Douglas W. Clark and Joel S. Emer. 1985. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 31–62. https://doi.org/10.1145/214451.214455

[41] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[42] Peter J. Denning. 1970. Virtual Memory. *ACM Comput. Surv.* 2, 3 (Sept. 1970), 153–189. https://doi.org/10.1145/356571.356573

[43] Xiaowan Dong, Sandhya Dwarkadas, and Alan L. Cox. 2016. Shared Address Translation Revisited. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 18, 15 pages. https://doi.org/10.1145/2901318.2901327

[44] Kshitij Doshi and Jantz Tran. 2006. Using Hugetlbfs for Mapping Application Text Regions. (01 2006).

[45] Hussein Elnawawy, Rangeen Basu Roy Chowdhury, Amro Awad, and Gregory T. Byrd. 2019. Diligent TLBs: A Mechanism for Exploiting Heterogeneity in TLB Miss Behavior. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 195–205. https://doi.org/10.1145/3330345.3330363

[46] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Article 40, 13 pages.

[47] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[48] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, USA, 231–242.

[49] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[50] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. HUB: Hugepage Ballooning in Kernel-Based Virtual Machines. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3240302.3240420

[51] Jerry Huck and Jim Hays. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/165123.165128

[52] James Wang. 2009. TLB Prefetching. https://patents.google.com/patent/US20110010521.

[53] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. IEEE Computer Society, Washington, DC, USA, 195–206. http://dl.acm.org/citation.cfm?id=545215.545237

[54] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[55] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[56] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[57] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783763

[58] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting Through the Front-End Bottleneck with Shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 30–42. https://doi.org/10.1145/3173162.3173178

[59] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, USA, 705–721.

[60] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[61] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez. 2020. CHiRP: Control-Flow History Reuse Prediction. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 131–145. https://doi.org/10.1109/MICRO50266.2020.00023

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

[63] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. 1993. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/165123.165127

[64] G. Ottoni and B. Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[65] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. 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)*. IEEE Press, 2–14.

[66] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. ISCA 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. 13. https://doi.org/10.1145/3079856.3080217

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

[68] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[69] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[70] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[71] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[72] G. Reinman, B. Calder, and T. Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 16–27. https://doi.org/10.1109/MICRO.1999.809439

[73] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/3079856.3080210

[74] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/339647.339666

[75] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. Association for Computing Machinery, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[76] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, 352–363. https://doi.org/10.1109/MICRO.2018.00036

[77] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1093–1108. https://doi.org/10.1145/3373376.3378493

[78] Steven P. Vanderwiel and David J. Lilja. 2000. Data Prefetch Mechanisms. *ACM Comput. Surv.* 32, 2 (June 2000), 174–199. https://doi.org/10.1145/358923.358939

[79] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 85–98. https://doi.org/10.1109/ISCA52012.2021.00016

[80] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[81] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[82] Idan Yaniv and Dan Tsafrir. 2016. 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)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2896377.2901456

[83] Y. Zhou, X. Dong, A. L. Cox, and S. Dwarkadas. 2019. On the Impact of Instruction Address Translation Overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 106–116. https://doi.org/10.1109/ISPASS.2019.00018