NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Improving the Locality of Page Table Walks in the Cache Hierarchy of Modern Microprocessors

Angelos E. Chatzopoulos

Supervisor: **Vasileios Karakostas,** Assistant Professor

ATHENS

August 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιώνοντας την Τοπικότητα του Πίνακα Σελίδων στην Ιεραρχία της Κρυφής Μνήμης Σύγχρονων Μικροεπεξεργαστών

Άγγελος Ε. Χατζόπουλος

Επιβλέπων: **Βασίλειος Καρακώστας,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

Αύγουστος 2024

BSc THESIS

Improving the Locality of Page Table Walks in the Cache Hierarchy of Modern Microprocessors

Angelos E. Chatzopoulos

S.N.: 1115201900217

SUPERVISOR: **Vasileios Karakostas,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιώνοντας την Τοπικότητα του Πίνακα Σελίδων στην Ιεραρχία της Κρυφής Μνήμης Σύγχρονων Μικροεπεξεργαστών

Άγγελος Ε. Χατζόπουλος

Α.Μ.: 1115201900217

ΕΠΙΒΛΕΠΩΝ: **Βασίλειος Καρακώστας,** Επίκουρος Καθηγητής

ABSTRACT

As the memory footprints of modern, memory-intensive workloads are increasing, conventional TLBs are often inadequate to fully cover their growing working sets, leading to frequent TLB misses that cause long latency page table walks due to accesses in the main memory.

In this thesis we propose PT-Baker, a software, system-level approach for reducing the latency of page table walks for applications that suffer from a high number of TLB misses and insufficient page table walks locality. The key idea of PT-Baker is the introduction of a helper thread that periodically iterates and accesses the workload's page table entries to preserve them within the cache hierarchy and accelerate the address translation process. We design and implement this approach in both (i) user-level, where the helper thread touches the workload's allocated memory using a page-size stride and triggers on purpose page walks, and (ii) kernel-level, by introducing a new system call that directly accesses the page table of the application. The user-level approach avoids any kernel modifications. However, it results in increased memory pressure because, as the helper thread iterates through the application's allocated memory to touch the page table entries and fetch them in the cache hierarchy, it also fetches the corresponding application data. On the other hand, the kernel-level approach directly fetches the page table entries without filling the memory hierarchy with application data, but this approach requires kernel modifications, in addition to application modifications.

We evaluate our approach on a native and a virtualized system, as well as on a memory-pressured system. We conduct experiments with various parameter settings to adjust the aggressiveness of the helper thread, while we also consider thread placement on the microprocessor to utilize different parts of the cache hierarchy. Our evaluation shows that PT-Baker reduces the main memory accesses due to page walks by up to 83% and improves the performance by up to 5.9% with the user-level approach on a native system. With the kernel-level approach, PT-Baker reduces the main memory accesses due to page walks by up to 99% and improves the performance by up to 18%.

SUBJECT AREA: Computer Architecture, Operating Systems, Hardware/Software Interaction

KEYWORDS: Virtual Memory, Address Translation, Translation Lookaside Buffer, Page Table, Cache Hierarchy, Cache Locality, Memory System

ΠΕΡΙΛΗΨΗ

Καθώς οι απαιτήσεις μνήμης των σύγχρονων εφαρμογών αυξάνονται, οι συμβατικοί πίνακες μετάφρασης σελίδων (TLB) των μικροεπεξεργαστών είναι συχνά ανεπαρκείς για να καλύψουν τα συνεχώς αυξανόμενα δεδομένα που επεξεργάζονται οι εφαρμογές αυτές, οδηγώντας σε συχνές αστοχίες TLB που προκαλούν χρονοβόρες διασχίσεις του πίνακα σελίδων λόγω προσβάσεων στην κύρια μνήμη.

Σε αυτή την πτυχιακή εργασία, προτείνουμε το PT-Baker, μια προσέγγιση λογισμικού που μειώνει την καθυστέρηση στη διάσχιση του πίνακα σελίδων για εφαρμογές με υψηλές αστοχίες TLB και ανεπαρκή τοπικότητα στον πίνακα σελίδων. Η βασική ιδέα του PT-Baker είναι η εισαγωγή ενός βοηθητικού νήματος στην εφαρμογή που περιοδικά πραγματοποιεί προσβάσεις μνήμης για να διατηρήσει τις καταχωρήσεις του πίνακα σελίδων της εφαρμογής εντός της ιεραρχίας της κρυφής μνήμης και να επιταχυνθεί με αυτόν τον τρόπο η μετάφραση εικονικών διευθύνσεων. Υλοποιούμε αυτή την προσέγγιση τόσο (i) σε επίπεδο χώρου χρήστη, όπου το βοηθητικό νήμα αγγίζει τη μνήμη της εφαρμογής ούτως ώστε να παράγει εκκούσια αστοχίες TLB και διασχίσεις στο πίνακα σελίδων για να φορτωθεί στην ιεραρχία της κρυφής μνήμης, όσο και (ii) σε επίπεδο πυρήνα, εισάγοντας μια νέα κλήση συστήματος που φορτώνει άμεσα τον πίνακα σελίδων της εφαρμογής στην ιεραρχία των κρυφών μνημών. Η προσέγγιση σε επίπεδο χρήστη αυξάνει την πίεση στην ιεραρχία της μνήμης γιατί φορτώνει και δεδομένα της εφαρμογής στην ιεραρχία των κρυφών μνημών, αλλά δεν απαιτείται εγκατάσταση τροποποιημένου πυρήνα λειτουργικού συστήματος. Η προσέγγιση σε επίπεδο πυρήνα φορτώνει άμεσα τις καταχωρήσεις του πίνακα σελίδων χωρίς να γεμίζει τη μνήμη με δεδομένα της εφαρμογής, αλλά απαιτείται επιπλέον τροποποίηση του πυρήνα.

Αξιολογούμε την προσέγγισή μας σε φυσικό και εικονικό σύστημα, καθώς και σε σύστημα με υψηλή πίεση μνήμης λόγω ανταγωνισμού στην ιεραρχία των κρυφών μνημών. Διεξάγουμε πολλαπλά πειράματα εξετάζοντας διάφορες παραμέτρους για να διαχειριστούμε την επιθετικότητα του βοηθητικού νήματος, και εξετάζουμε την τοποθέτησή του βοηθητικού νήματος στον μικροεπεξεργαστή για να αξιοποιήσουμε την ιεραρχία των κρυφών μνημών. Η αξιολόγηση μας δείχνει ότι το PT-Baker μπορεί να μειώσει τις προσβάσεις στην κύρια μνήμη έως και 83% και να βελτιώσει την απόδοση έως και 5.9% με την προσέγγιση επιπέδου χρήστη σε ένα φυσικό σύστημα. Με την προσέγγιση επιπέδου πυρήνα, το PT-Baker μπορεί να μειώσει τις προσβάσεις στην κύρια μνήμη έως και 99% και να βελτιώσει την απόδοση έως και 18%.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών, Λειτουργικά Συστήματα, Αλληλεπίδραση Υλικού/Λογισμικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Εικονική Μνήμη, Μετάφραση Διευθύνσεων, Κρυφή Μνήμη Μετάφρασης Διευθύνσεων, Πίνακας Σελίδων, Ιεραρχία Κρυφών Μνημών, Τοπικότητα Κρυφής Μνήμης, Σύστημα Μνήμης

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Vasileios Karakostas for his dedicated support and guidance. Without his persistent help and encouragement, this thesis would not have been possible.

In addition, I would like to extend my thanks to professor Dimitrios Gizopoulos and researcher George Papadimitriou for their valuable contributions and comments throughout the research.

Last but not least, I would like to thank my family and friends for the continuous support while undertaking my research and writing my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis was completed at the National and Kapodistrian University of Athens during the 2023 - 2024 academic year under the supervision of assistant professor Vasileios Karakostas.

1. INTRODUCTION

Virtual memory is present in almost all classes of modern computing systems. The benefits of enhanced isolation, security between processes, as well as increased programmer productivity, have constituted its presence ubiquitous. However, virtual memory provides all the aforementioned benefits by introducing an abstraction layer between the virtual address space that the applications see, and the physical memory that the computing system is equipped with. This abstraction layer requires translation from the virtual to the physical address space and comes with adequate software and hardware support. At the software level, the operating system manages the memory and stores mappings from the virtual to the physical address space in the page table. At the hardware level, the translation lookaside buffer (TLB) accelerates this translation step by caching recently used virtual-to-physical mappings.

The impact of virtual memory on application performance depends on various application characteristics (e.g., use of memory management library and system calls, memory access patterns), system characteristics (e.g., memory management library, operating system, processor model), and conditions (e.g., memory fragmentation). Several recent works [32, 28, 23, 31, 25, 29, 24, 19, 21, 20, 9] have shown that memory-intensive applications that operate on large working sets, tend to stress the limits of the TLB and spend a significant amount of their execution time in page table walks due to TLB misses.

Prior works that have focused on reducing the latency of page table walks (and that are closely related to this thesis) have mainly followed two approaches: (i) they either preserve page table structures in the cache hierarchy [32, 23, 31, 28, 25], or (ii) directly store TLB entries within the cache memory hierarchy [29, 24, 19, 22, 18]. However, most of those works require changes at the architectural (e.g., modifications at the organization of the page table) or at the microarchitectural level, and hence cannot improve the performance of existing real systems.

1.1 Goal & Motivation

The goal of this thesis is to reduce the latency of the page table walks by improving the locality of the page table in the cache hierarchy. We target commodity, off-the-shelf systems, avoiding the need for any microarchitectural modifications.

To motivate our approach, we perform an analysis of the locality of the memory references that the page table walks induce in the memory hierarchy using hardware performance counters for a set of memory intensive applications. We find that a significant percentage of the memory references due to page walks miss in the cache hierarchy and end up accessing the main memory (DRAM). More specifically, we observe that 10.3% on average and up to 32.4% of page walk accesses are fetched from DRAM in native execution. In a memory-pressured system, we observe that 33% on average and up to 51.5% of page walk accesses are fetched from DRAM. Under virtualized execution, these percentages change to 16.3% on average and up to 33.4%, respectively. Hence, there is significant opportunity for improving the application performance by converting those DRAM accesses into cache hits.

1.2 Approach

To improve the locality of the page table walks in the cache hierarchy, we propose PT-Baker. PT-Baker is a software, system-level approach for reducing the latency of page table walks for applications that suffer from high TLB misses and insufficient page table walks locality. PT-Baker introduces a helper thread that periodically iterates through one or more target application data structures that are responsible for the majority of data TLB misses, to preserve their corresponding page table entries (PTEs) warm within the cache hierarchy.

The proposed method requires modification of the application code to introduce the helper thread. To keep the PTEs warm in the cache hierarchy, we implement a user-level and a kernel-level approach. The user-level approach touches the allocated space using specific page strides to improve the cache locality of the page table itself without thrashing application data in the cache hierarchy. However, this method also fetches the corresponding data, thus increasing the pressure in the cache hierarchy. In contrast, the kernel-level approach introduces a new system call that directly accesses only the page table itself. While this approach avoids fetching any application data in the cache hierarchy, it requires modifying also the operating system kernel. To throttle the aggressiveness of the PT-Baker helper thread in the memory hierarchy, we insert delay events (i..e, NOP instructions and sleep system calls) after every iteration round and control the placement of the helper thread, taking into consideration the multicore organization.

We design and implement PT-Baker in Linux v6.6.10 and evaluate it using various memory intensive benchmarks on a multicore AMD Ryzen 9 5900X Zen 3 CPU. Our results indicate that the kernel-level approach offers significantly better results and it is also less sensitive to the parameters that control aggressiveness. On a native environment, the user-level approach improves the workloads performance by up to 5.9%, while the kernel-level approach improves performance by up to 18%. Testing on a memory-pressured system provides up to 11.2% performance improvement with the user-level approach and 22.4% with the kernel-level approach. In virtualized environments, performance improvements reach up to 38.2% using the user-level approach, though due to 2D nested page walks that add more pressure to the cache hierarchy compared to a native system, the average performance improvement is limited and requires further parameter exploration to adjust the aggressiveness of the helper thread. Finally, our performance improvements also come with slightly higher CPU power consumption. The user-level approach increases power by 5.4% on average, while the kernel-level approach can increase it by up to 9.4%; however by adjusting the PT-Baker thread's aggressiveness, we can reduce power usage while still achieving similar performance benefits.

1.3 Thesis Contributions

In summary, the main contributions of this thesis are:

- We analyze the locality of page walks in the memory hierarchy for several memory intensive applications and quantify the opportunity for reducing the latency of page walks.
- We propose a system-level, microarchitecture-aware approach that improves the locality of the PTEs in the memory hierarchy by touching periodically the PTEs. Our

approach works in both user and kernel levels.

- We comprehensively evaluate our proposal executing several memory intensive applications in both native and virtualized execution setups, as well as executing them on a memory-pressured system in conditions of cache contention due to resource interference.
- We systematically analyze the impact of several parameters in the effectiveness of the proposed approach.

1.4 Organization

The rest of this document is structured as follows. In Chapter 2 we provide essential background information regarding caches and virtual memory. In Chapter 3 we describe our motivation for this work. In Chapter 4 we present our approach, while in Chapters 5 and 6 we describe our evaluation methodology and present results, respectively. In Chapter 7 we describe the related work around optimization methods for virtual memory. Finally, in Chapter 8 we conclude this work and give an outlook on future work.

2. BACKGROUND

In this chapter we describe the basic functionality and organization of caches. We also analyze the address translation process in native and virtualized systems, present the necessary architectural/microarchitectural support, and briefly describe the optimization mechanism of huge pages for improving the performance of address translation.

2.1 Cache Memory

Cache memory is a small memory hardware component located in the microprocessor that is used to reduce the average cost of accessing data and instructions from main memory and minimize the CPU's stalled cycles. In modern microprocessors, caches play a crucial role in the performance and energy efficiency. The effectiveness of caches mainly depends on two important principles: the temporal locality and the spatial locality. Temporal locality means that if a particular memory address is referenced, it is likely to be referenced again in the near future. On the other hand, spatial locality means that when a particular memory address is referenced, nearby locations are also likely to be referenced in the near future. To reduce the total DRAM accesses, modern CPUs are equipped with a multi-level cache hierarchy, i.e., commonly level 1 (L1), level 2 (L2) and level 3 (L3 or last-level cache – LLC), that are respectively larger but require higher access latency.

2.2 Virtual Memory

Commodity modern operating systems and microprocessors typically support virtual memory. Virtual memory separates the address space of every process from each other, by providing a transparent and secure way for processes to manage (i.e., allocate and free) and access memory. Virtual memory introduces mappings between virtual addresses and physical addresses in page-size (commonly 4KB or 2MB) granularity. The above address mapping mechanism demands both operating system and architectural support to make this address translation process efficient.

2.2.1 Address Translation

The page table is a software data structure that is managed by the operating system and holds all the mappings of each application from the virtual to the physical address space. To perform virtual to physical address translation for a memory request, the operating system (and the hardware depending on the ISA, as explained next) walks the page table. This process is called a page table walk or page walk.

The organization of the page table depends on the instruction set architecture (ISA). A common organization that is employed by multiple ISAs, e.g., x86, ARM, RISC-V, is the radix tree. The radix tree is usually organized in multiple levels, depending on the size of the supported address space. The page table walk requires multiple memory references to obtain the physical address from the last level leaf node, known as page table entry (PTE). For example, the three-level and four-level radix trees provide support for 39-bit and 48-bit address spaces, and the page table walk requires three and four memory accesses, respectively. Although the page table can be cached within the memory hierarchy, its

locality can significantly affect the address translation process resulting in long-latency page table walks. With the introduction of Intel's five-level page tables [1], which extends the virtual addresses size from 48 bits to 57 bits to increase the maximum capacity of physical memory, the number of memory references due to page table walks increases, resulting in even longer delays during the address translation process.

Figure 2.1 shows a 48 bit, four-level radix tree of page tables. There are four levels of page tables, i.e., PGD, PUD, PMD, and PT, each one pointing to the next page level. The address translation process is sequential and starts from the PGD directory which is pointed by the CR3 register, goes through all page table levels, and completes upon reaching the last leaf node, i.e., the PTE. The final physical address is generated by combining the value of the corresponding PTE with the page offset of the virtual address.

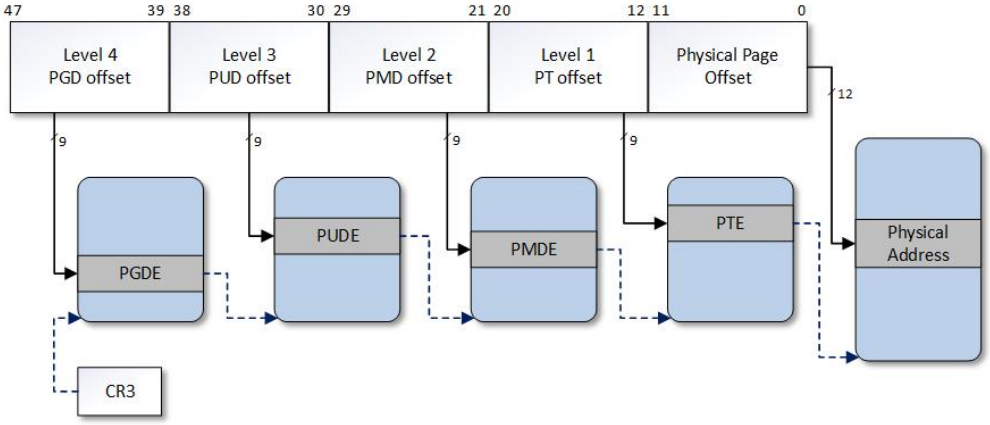


Figure 2.1: The four-level radix page table organization of the x86-64 ISA.

2.2.2 Accelerating Address Translation

To accelerate the address translation process, microprocessors employ a Memory Management Unit (MMU) that consists of two main caching components: (i) the Translation Lookaside Buffer (TLB), and (ii) the Page Walk Caches (PWCs).

The TLB stores the most recently translated virtual addresses as entries in page granularity, and provides directly the corresponding physical address, without the need to trigger a page walk, reducing the memory address translation latency. The growing tendency towards bigger and more TLB intensive workloads has driven CPU manufacturers to enhance the TLB by including separate TLBs for data and instruction accesses and introducing a two-level TLB hierarchy to avoid the excessive amount of page walks. For example, modern AMD Zen 3 microprocessors (which we use in this thesis) implement a two-level TLB organization per core, with separate L1 Data and Instruction TLBs being fully associative and a unified L2 TLB with 16-way associativity. In this thesis we focus on L2 TLB misses due to data accesses that trigger page walks and access the memory hierarchy to retrieve the missing translation entry.

Figures 2.2 and 2.3 show the virtual address translation mechanism with one-level and two-level TLBs, respectively. Upon missing in the L2 TLB, a page walk process starts. There are two types of TLB miss handling: hardware and software. In case of hardware-managed TLB, a hardware state machine named as page table walker walks the page table directly using the CR3 register to translate the missing address. In case of software-managed TLB, a TLB miss exception is triggered and the operating system code (i.e., the

TLB miss handler) is responsible to complete the address translation process and install the entry in the TLB. In this thesis we focus on the x86-64 ISA which requires hardware-managed TLBs.

In addition, a private low latency cache called Page Walk Cache (PWC) [12] is implemented within modern microprocessors' MMUs. This cache is dedicated to storing address translations for intermediate levels of the radix page table that have been previously walked. Therefore, when a TLB miss occurs, the page walk process is accelerated by skipping some lookups for specific page table levels, reducing the overall memory address translation overhead of an application.

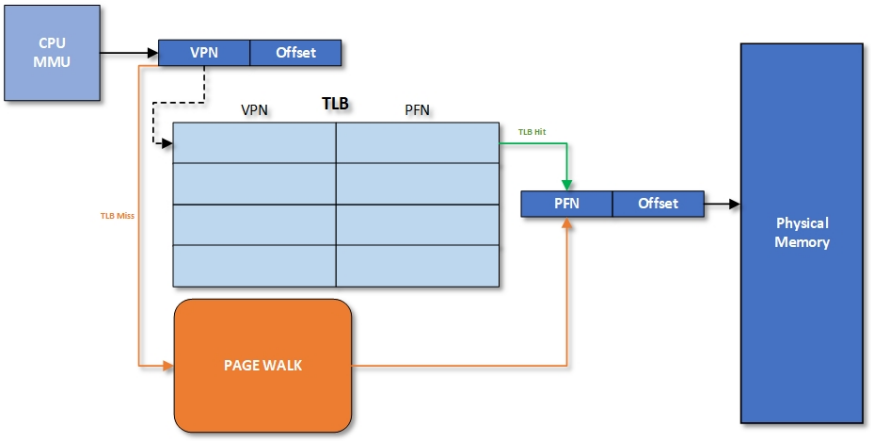


Figure 2.2: Virtual address translation using a one-level TLB.

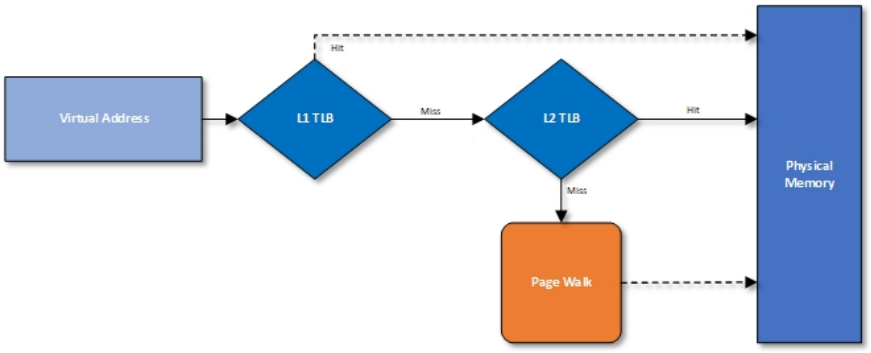


Figure 2.3: Virtual address translation using a two-level TLB hierarchy.

2.2.3 Improving Address Translation through Huge Pages

The TLB reach can be improved by using large or huge pages, i.e., 2MB or 1GB in the x86-64 ISA. Huge pages require support from both the processor and the operating system vendors. On the processor side, the MMU is extended to cache translations of huge pages. On the OS side, the memory management subsystem is extended to manage mappings at huge page granularity. For example, the Linux kernel provides support for transparent hugepages (THP) that is a feature of allocating anonymous memory mappings of huge 2MB pages, transparently to the applications [6].

Huge pages decrease the number of TLB misses radically since every TLB entry provides translation information for a larger memory region. Huge pages also reduce the page table walking latency because page table leafs are located within the third level of page tables

on the radix tree structure (PMD entries). However, using larger page sizes results in the following disadvantages. First, applications often experience higher latency during page fault exceptions. Second and most important, system memory fragmentation increases over time making it more difficult for the OS to allocate huge pages at runtime. Third, the mechanism of transparent huge pages is not controlled directly by the users. Therefore, 4KB pages can be promoted to larger pages even if it is not beneficial for the application, leading to higher memory usage.

2.2.4 Virtual Memory in Virtualized Environments

In virtualized environments, the address translation latency increases significantly. This increase occurs because the address translation process consists now of a two-dimensional page walk due to the nested page tables that are handled by the guest operating system and the host operating system. Figure 2.4 shows a memory address translation from a guest virtual address (gVA) to a host physical address (hPA) on the x86-64 architecture with a four-level page table structure. Guest's CR3 and page table levels must also be translated from the host's virtual to the host's physical address space, resulting in this nested two-dimension search process. The aforementioned mechanisms for accelerating address translation, i.e., TLB and PWCs, can be used in virtualized environments as well. The address translation process for a single guest virtual address, in case of a TLB miss, may require up to 24 memory accesses in the memory hierarchy, depending on the PWCs. Finally, huge pages can also be used in the guest OS, in the host OS, or in both the guest and host OS.

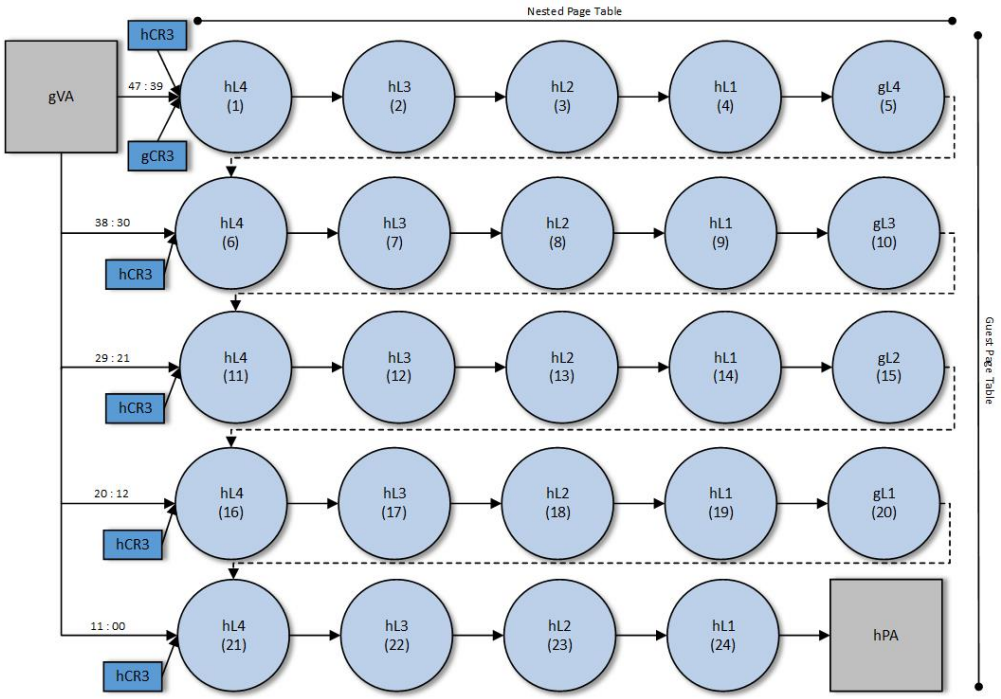


Figure 2.4: Virtual address translation in a virtualized system using four-level nested page tables.

3. MOTIVATION

Excessive TLB misses and high latency page walks are a major performance issue for many emerging workloads that demand large amounts of memory. This issue becomes particularly important for workloads that exhibit irregular access patterns, which additionally increases the amount of cache misses. In such cases, the PWCs and the CPU cache hierarchy fail to reduce the latency of the page walks effectively, leading to time consuming DRAM accesses.

In this chapter we conduct performance analysis on a series of workloads in order to measure the page table walks locality. To put our findings into perspective, we compare the frequency of main memory accesses due to page walks with the frequency of main memory accesses for fetching application data. We perform our analysis in both native and virtualized environments as well as on a native system under high memory pressure. We find that a significant percentage of page walk memory references end up accessing the main memory. We also find that the frequency of LLC misses is much worse for application data than for page walks. We thus get confidence that there is a high chance of reusing cached PTEs during page table walks compared to reusing cached application data, by leveraging also the packed organization and reduced size of the page table.

3.1 Quantifying the Frequency of Misses in the Memory Hierarchy

To better understand the locality of page walks, we first conduct a series of performance analysis experiments on an AMD Zen 3 microprocessor using the hardware performance counters and the workloads described in Chapter 5. We measure the frequency of misses in the memory hierarchy, i.e., misses per kilo-instructions (MPKI) for (i) L2 TLB due to data accesses, (ii) DRAM accesses during page walks, and (iii) data DRAM accesses (LLC misses). Figure 3.1 summarizes the results.

We observe that all workloads except Graph500 have very high data DRAM accesses due to their irregular access patterns that result in low data locality in the LLC. They also experience a high number of DRAM accesses due to page walk references. Given that the size of the page table is significantly smaller with respect to the actual dataset of an application, the application performance could improve by prioritizing the contents of the page table in the LLC, reducing in this way the latency of page table walks. Although Graph500 has significantly lower DRAM accesses compared to the other workloads, it still suffers from a high number of DRAM accesses due to page walk memory references. Hence, keeping its PTEs warm inside the cache hierarchy could also help speed up the page table walks for that application as well.

3.2 Quantifying the Locality of Page Walks in the Memory Hierarchy

We also conduct a performance analysis for the same workloads to measure the percentage of accesses in the memory hierarchy for the memory references due to page table walks in a native system and in a virtualized environment. In addition, we perform a profiling analysis under memory-pressure conditions due to resource interference in the LLC, using the stress-ng tool with one stressor thread and the *cache* parameter, and with two stressor threads and the *stream* parameter.

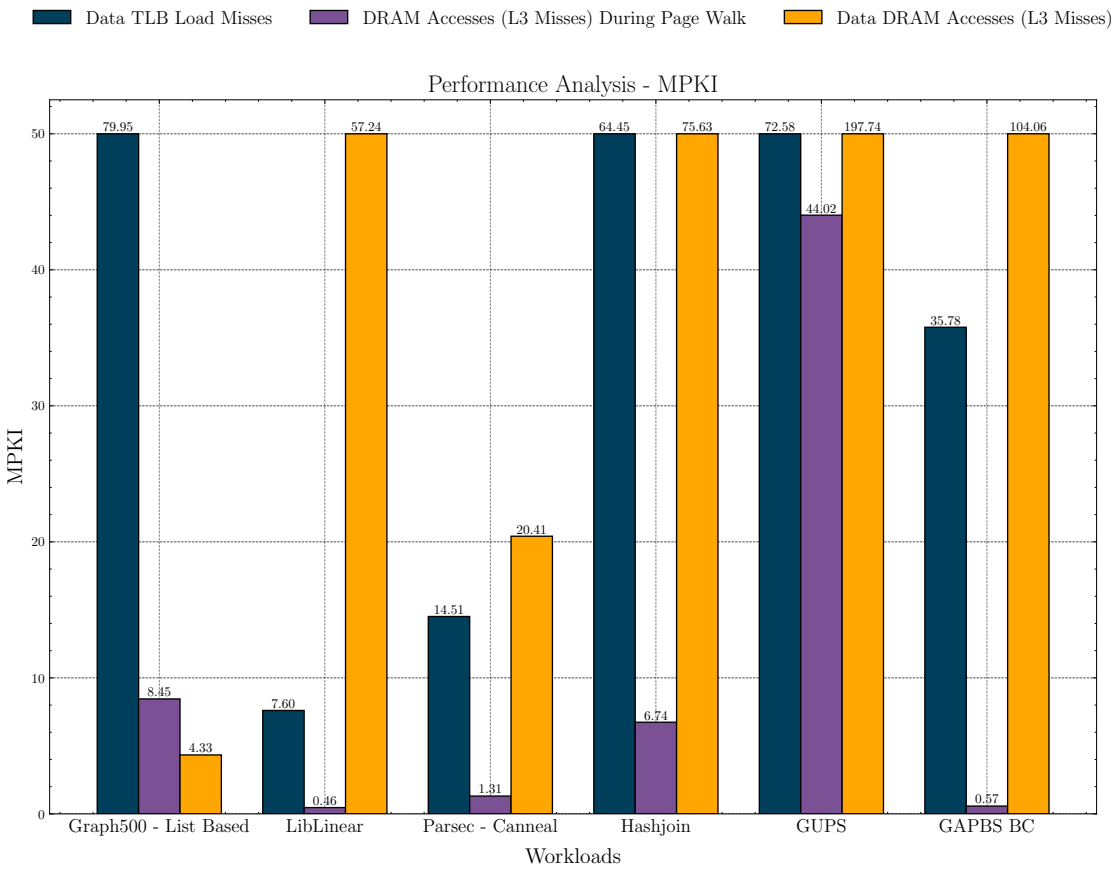
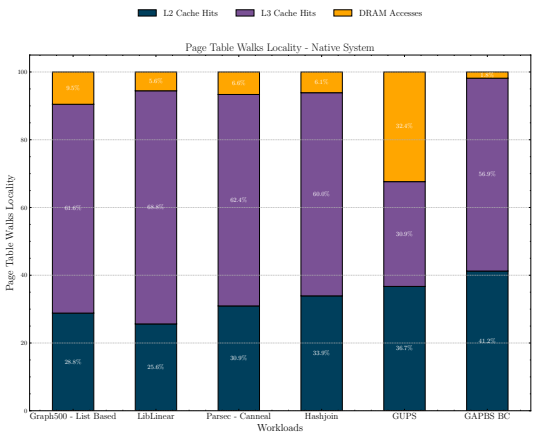


Figure 3.1: Misses per kilo instructions (MPKI) for L2 data TLB load misses, DRAM accesses (LLC misses) during page walk, and data DRAM accesses (LLC misses) in native execution.

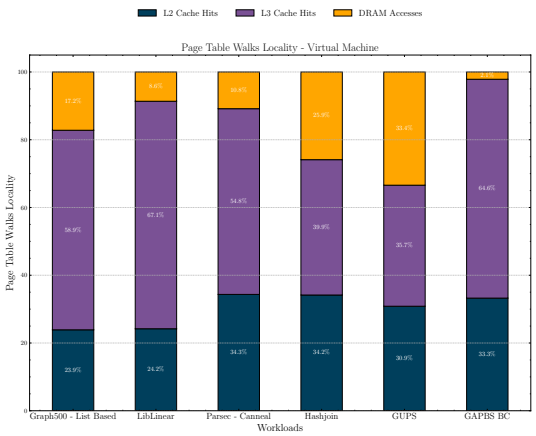
Figures 3.2 and 3.3 illustrate the results for the system scenarios mentioned above. In native execution, on average, 10.3% (up to 32.4%) of page walk accesses retrieve data from DRAM. In memory-pressured systems using one *cache* stressor thread, this average increases to 33% (up to 51.5%). With two *stream* stressor threads, the average percentage of accesses during the page walk process rises to 33.6% (up to 55%). In virtualized environments without memory pressure, on average 16.3% (up to 33.4%) of page walk accesses end up accessing DRAM. For example, on a native system, the LibLinear workload executes 2.3×10^{12} instructions with 5.1×10^{9} L2 cache hits, 13.7×10^{9} LLC hits, and 1.1×10^{9} main memory accesses during the page table walks. Moreover, in a virtual environment where the address translation consists of nested page walks, the accesses to main memory have increased significantly. The LibLinear workload now executes 2.3×10^{12} instructions with 11×10^{9} L2 cache hits, 30.5×10^{9} LLC hits, and 3.9×10^{9} main memory accesses during the page walks.

3.3 Opportunity for Improvement

Based on our performance analysis we can clearly notice that a significant percentage of accesses during page walks are fetched from the main memory across all workloads. We also notice that the MPKI for LLC data accesses is very high, which highlights the low data locality and the opportunity for techniques that can efficiently maintain the page table structures within the cache hierarchy, since cached page table entries are denser and likely to be used again soon than data during periods of high data cache misses.

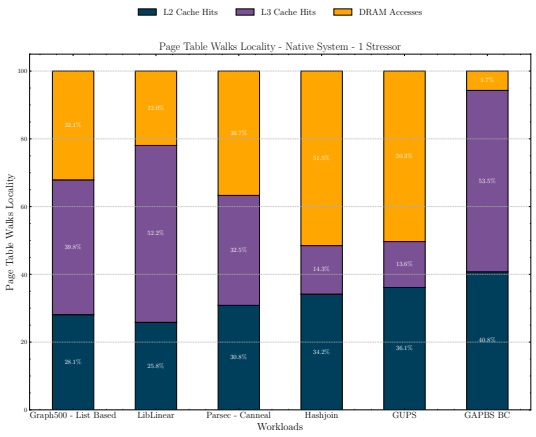


(a) Native System

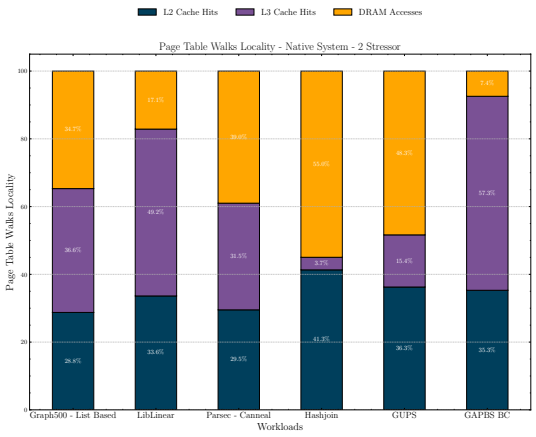


(b) Virtual Machine

Figure 3.2: Breakdown of the page walks locality, i.e., percentage of memory accesses due to page table walks that are fetched from the L2 cache, L3 cache, and DRAM, in native (left) and virtualized (right) execution.



(a) Native - 1 Stressor thread



(b) Native - 2 Stressor threads

Figure 3.3: Breakdown of the page walks locality, i.e., percentage of memory accesses due to page table walks that are fetched from the L2 cache, L3 cache, and DRAM, on a native memory-pressured system using 1 *cache* stressor thread (left) and 2 *stream* stressors threads (right) with the stress-ng tool.

4. PT-BAKER: A HELPER-THREAD APPROACH

In this chapter we present the main idea of PT-Baker and describe its design and implementation in user-level and kernel-level, respectively. We also present a thorough examination of the parameters that affect its performance, and finally we discuss alternative implementations.

4.1 Overview

Our main approach is to create a software helper thread named Page Table Baker or simply PT-Baker, that periodically touches the page table structures of the main thread's workload in order to preserve them within the cache hierarchy, reducing the number of the main memory accesses due to page table walks. Algorithm 1 and Figure 4.1 show the necessary application modifications to create and setup the PT-Baker thread and the workflow of PT-Baker, respectively. Initially, before the application's main algorithm starts, the PT-Baker thread must be spawned and provided with pointers to the application memory regions that are responsible for the majority of TLB misses. Then, the PT-Baker starts touching periodically the page table entries of the corresponding application data in an infinite loop until the application's main algorithm thread finishes, while the application's main algorithm thread starts its execution and proceeds as normally. After the application's main algorithm thread completes, the PT-Baker thread is stopped and the execution ends.

PT-Baker requires modifications in the application code to start, execute, and stop the helper thread in user-level, accordingly. To touch the page table entries, we design and implement a user-level and a kernel-level approach that offer different trade-offs. The user-level PT-Baker approach touches the page table entries by loading the corresponding data of one or more memory allocated regions of the application, and triggering on purpose page table walks. The user-level approach does not require a modified kernel to be installed, but by accessing the allocated memory regions to trigger page walks, it also fetches the corresponding data and adds pressure to the system's memory hierarchy. In contrast, the kernel-level PT-Baker approach also introduces a user-level thread. However, instead of loading the corresponding data, the kernel-level PT-Baker approach introduces a new system call that directly accesses the page table structures without fetching any application data. Thus, the kernel-level PT-Baker approach introduces less memory pressure, but it requires kernel modifications.

To manage the aggressiveness of PT-Baker based on each workload's pressure to the system's memory hierarchy, PT-Baker uses a series of configurable parameters. In addition, PT-Baker is microachitecture-aware, taking into consideration the core and cache organization of the system.

Algorithm 1: User/Kernel-level PT-Baker thread design within a workload.

```
1 init_workload()
2 create_pt_baker_thread(workload_data)
3 start_workload_main_task()
4 stop_pt_baker_thread()
```

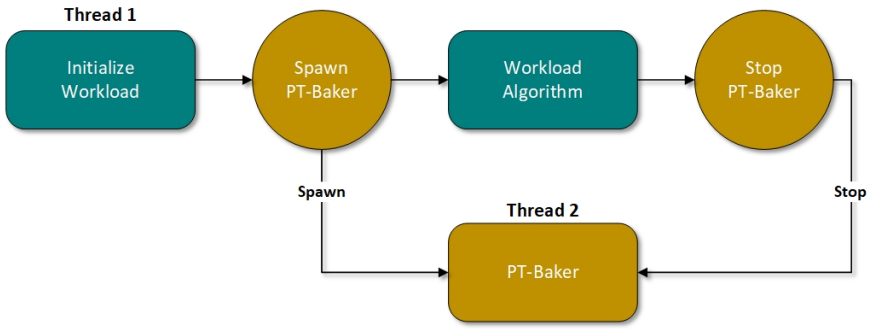


Figure 4.1: PT-Baker design and synchronization within a workload.

4.2 User-level Approach

When designing the user-level approach, we must first consider how to interact with the application's memory regions, as direct access to the page table radix tree is not possible in user-space. For PT-Baker to access the application memory regions that cause a high number of TLB misses, it must be able to access both a pointer to the target memory region and the size of the memory region. As a result, it is necessary that the memory allocations have been issued and any necessary parameters have been initialized accordingly, before the PT-Baker thread is spawned.

Algorithm 2 shows the design of the user-level PT-Baker thread iterating on a single memory region. PT-Baker provides support for applications that use multiple memory regions, i.e., more than one large data structures. However, for the workloads that we use in this thesis, we observe that there is always a single memory region that suffers from frequent TLB misses and thus page walks locality issues. Apart from spawning and executing the helper thread, PT-Baker does not require any further application modifications. We only observed in Graph500 that the introduction of the helper thread may affect the behavior of the memory allocator, leading to variability in execution times. To address that issue, we modified Graph500 to use the *mmap* system call instead of glibc's *malloc* function, avoiding such inconsistencies.

To ensure that the helper thread's access pattern is preserved at the executable program, the compiler optimizations are disabled for this function by specifying to the GCC compiler not to use any compiler optimizations on the helper thread function source code file or by using a GCC attribute keyword on helper thread that disables optimizations for a specific function. We use NOP instructions and sleep system calls to manage, i.e., throttle, the aggressiveness of the helper thread (analyzed in Section 4.4). Finally, user-level PT-Baker does not require any synchronization mechanism to access the application's allocated memory region, as the only concern is to trigger page table walks to preserve the page table structures within the cache.

4.3 Kernel-level Approach

Although the user-level PT-Baker thread can trigger page table walks and fetch page table entries within the cache hierarchy in user-space, direct access to the page table radix tree is not possible without fetching the corresponding application data. To avoid fetching additional data within the memory hierarchy we have to access the page table structures directly from the kernel-space, i.e., the operating system. Therefore, we design and imple-

Algorithm 2: User-level PT-Baker thread function

```
1  Function user_baker_thread(workload_array, array_length, page_step,
     prefetch_enabled, nops, microseconds):
2      offset ← page_step / sizeof(workload_array[0])
3      while true do
4          for i ← 0 to array_length , i ← i + offset do
5              if prefetch_enabled then
                   // Prefetch instruction method
6                  prefetch(workload_array[i])
7              else
                   // regular load instruction
8                  access_byte = workload_array[i]
9              end
               // nops and microseconds can be 0
10             for j ← 0 to nops do
11                 asm("nop")
12             end
13         end
14         sleep(microseconds)
15     end
16 End Function
```

ment a new system call that takes one or more address memory regions of the application and efficiently traverses all the page table level entries that are responsible for translating those memory regions.

The implementation of the system call function is based on the Linux kernel 6.6.10 *page-walk.c* source file under the kernel's *mm* directory. The system call traverses the page table structures with a page stride value as a step size to achieve greater cache locality and reduce memory pressure on the system. To efficiently access the page table levels once, the system call traverses the page table from the initial PGD to PUD, PMD, PT and then to PTE using the page stride mentioned above. After accessing all the PTEs of a specific PT index, it moves to the next PT structure index. After finishing with all the PTs it moves to the next PMD and so on. The system call can traverse both 4-level and 5-level page tables.

To safely iterate over the page table structures, we lock the memory descriptor *mm_struct* in read mode, to ensure that no other OS components modify the contents of the application's page table while the system call traverses it. The page walk algorithm of the system call is also able to recognize and iterate over transparent huge pages by temporarily locking a specific page table lock that prevents splitting the pages into regular 4KB size. Algorithm 3 is a simple pseudocode that shows the traversal of PTEs within the kernel-space using a specific page stride access pattern in kernel.

Algorithm 4 shows the implementation of the PT-Baker thread that calls iteratively the system call that we described above. The parameters *starts_addrs*, *end_addrs* are arrays that store the addresses (start and end) of the address spaces that we want to iterate over. The parameter *length* stores the number of address spaces that we are going to iterate over and it is the size of the arrays *starts_addrs* and *end_addrs*. The parameter *offset* consists the page stride that we mentioned above, and finally *nops* and *microseconds* are

Algorithm 3: Walk PTE range function

```
1  Function walk_pte_range(pte, addr, end, walk, offset, nops):
2      page_offset ← offset × page_size
3      pte_offset ← offset
4      while true do
5          if pte then
6              pte_access ← value of pte
7          end
           // nops can be 0
8          for z ← 0 to nops do
9              asm("nop")
10         end
11         if addr ≥ end − page_offset then
12             break
13         end
14         addr ← addr + page_offset
15         pte ← pte + pte_offset
16     end
17     return 0
18 End Function
```

parameters that manage the aggressiveness of the PT-Baker by adding a delay between accesses and they are described in further detail in Section 4.4.

Algorithm 4: Kernel-level PT-Baker thread function

```
1  Function syscall_pt_baker_thread(starts_addrs, end_addrs, length, offset, nops,
     microseconds):
2      while true do
           // nops and microseconds can be 0
3          syscall_pt_baker(starts_addrs, end_addrs, length, offset, nops)
4          sleep(microseconds)
5      end
6  End Function
```

4.4 Configuring the Helper Thread

As already mentioned above, the user-level and kernel-level approach configurations have the following common parameters to configure: page step/PTE stride, NOP instructions, and sleep system calls. In addition, the user-level approach can touch the application's memory region using either prefetch instructions or regular load instructions, as shown in Algorithm 2. The PT-Baker thread in general is very sensitive to those parameters, especially the user-level approach that also fetches additional application data within the cache hierarchy. Therefore, selecting an optimal configuration requires experimenting with various parameters, as the behavior of PT-Baker also depends on the system workload, i.e., on applications that runs alongside with the target application. The above parameter series are described in the following sections and evaluated in Chapter 6.

4.4.1 Selecting Page Stride

Since the user PT-Baker thread performs an iterative access to the application memory region, it firstly needs to calculate the loop offset to access and trigger page table walks minimizing redundant data fetches within the cache hierarchy. In a 4KB paging system, a typical value for the loop offset can be $8 \times 4KB$ to utilize the cache block size while fetching the PTEs; the size of each PTE is 64-bits and the cache block size is 64 bytes, hence each cache block contains 8 PTEs.

Another interesting page stride value could be $512 \times 4KB$ (i.e., 2MB granularity), where we only access the first PTE of each page directory entry (PDE) on 4KB pages, restricting the page table structures that are going to be cached to a specific page table level depth. This value can be effective for applications that specifically use 2MB pages for the target allocated structure and still experience frequent TLB misses that cause long-latency page table walks [9].

Similar to the user-level approach, an optimal stride for PTE accesses in kernel PT-Baker is 8, to utilize the cache block size and spatial locality.

4.4.2 Triggering Page Walks

There are two methods for triggering address translation during the user-level PT-Baker thread's execution: using prefetch instructions or using regular load instructions by performing a variable assignment. However, while the prefetch instructions seems a valid method to access an application memory region, we first need to ensure that it also triggers page table walks. Next we perform relevant experiment and show that prefetch instruction can indeed trigger page table walks in the AMD Zen 3 architecture. For the kernel-level PT-Baker approach, we only experimented with using regular load instructions in the implementation of the system call for accessing directly the page table structures, although prefetch instructions could be used as well.

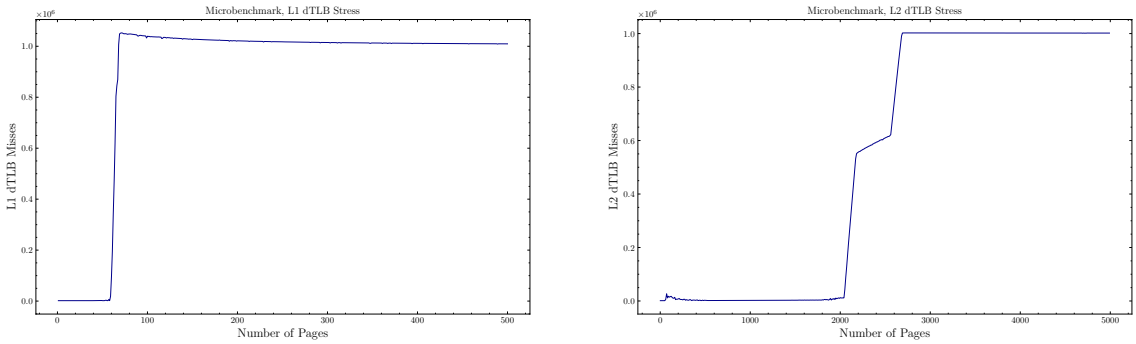
4.4.3 Crafting a TLB microbenchmark

To explore and understand the microarchitectural details of the MMU design (particularly with respect to whether prefetch instructions trigger page table walks) of the AMD Zen 3 (that we use in this thesis), we implement a custom microbenchmark that thrashes the TLB hierarchy due to data accesses. Algorithm 5 shows the TLB thrashing microbenchmark using a simple iterative loop. In each iteration, the microbenchmark accesses the first byte of every page of a configurable allocated array structure. As we can see in Figure 4.2, when the array allocation size becomes equal to 64 pages of 4KB each, the L1 TLB is thrashed. This behavior confirms the documented L1 TLB size of our system that is 64 entries. Similarly, the L2 TLB begins to exhibit a high number of TLB misses as the array is reaching the 2048 pages size, and is ultimately thrashed when using 2560 4KB pages. This behavior confirms the documented L2 TLB size of our system that is 2048 entries, while the difference in L2 TLB misses between using 2048 and 2560 entries can be attributed to the TLB Coalescing [30] support that allows the L2 TLB to map more pages than the default size (depending on the contiguity of memory allocations). Overall, using our simple microbenchmark we verify the sizes of the TLB hierarchy, and thus we can use it to reverse engineer more properties with respect to the MMU, as explained next.

Algorithm 5: TLB Thrashing Algorithm Experiment

Input: $Byte\ Indexed\ Allocated\ Array$, $page\_step > 0$, $N > 0$, $array\_size > 0$

$page\_index \leftarrow 0$
$offset \leftarrow page\_size \times page\_step$
for $i \leftarrow 0$ **to** $N - 1$ **do**
    $array\_index \leftarrow page\_index \mod array\_size$
    $access \leftarrow array[array\_index]$
    $page\_index \leftarrow page\_index + offset$
end



(a) L1 DTLB (b) L2 DTLB

Figure 4.2: L1 (left) and L2 (right) DTLB thrashing experiment using the TLB thrashing algorithm.

4.4.3.1 Reverse Engineering the MMU behavior of the Prefetch Instruction

We now determine whether the prefetch instruction triggers page table walks. We review the generated assembly for the *_mm_prefetch* built-in function, and we observe that GCC compiler uses the x86 *prefetcht0*, *prefetcht1* and *prefetcht2* assembly instructions. The choice among these assembly instructions depends on the argument provided to the builtin instruction, that defines in which level of the cache (and above) should the data be loaded. However, depending on the CPU hardware design, the prefetch instruction might or might not trigger the MMU to walk the page table. In order to examine the behaviour of the assembly instruction, we run the TLB microbenchmark in two ways and profile its execution using the PAPI library [26]. First, we use the TLB microbenchmark with an array of 4224 4KB pages allocated (double the size of L1 and L2 DTLB entries), but we only touch half of these pages using regular load instructions (i.e., the variable assignment method), and measure the L2 TLB misses. Then, we use the TLB microbenchmark with the same array size, touching half of the pages using the load instruction and additionally touching the other half, which were not previously accessed, using the prefetch instruction. If the prefetch instruction fills the L2 DTLB, we are going to observe increased L2 DTLB misses on the second run compared to the first experiment. Figure 4.3 results clearly indicate that the prefetch instruction triggers TLB fills in our system, thus page walks are triggered and the instruction can be used in our user-level approach.

4.4.4 Throttling PT-Baker

Furthermore, to be able to manage the aggressiveness of PT-Baker for fetching PTEs in the cache hierarchy and their respective data in the case of the user-level approach, we

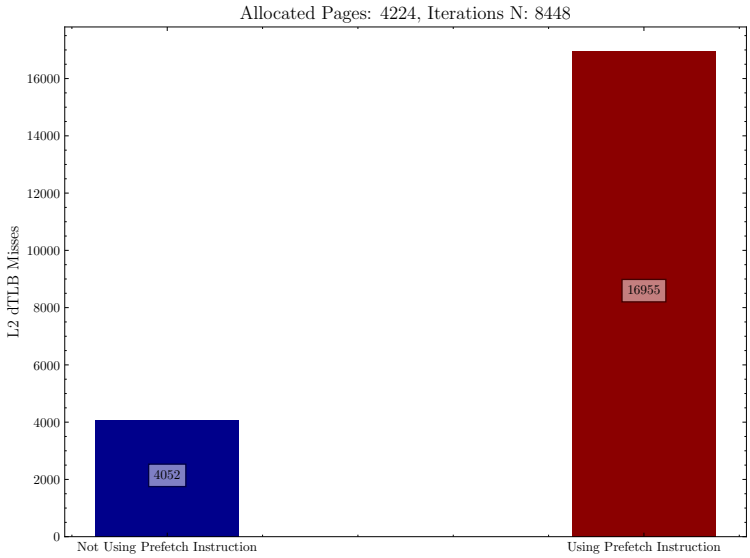


Figure 4.3: Reverse engineering the prefetch instruction's page table walk behavior. We observe increased L2 DTLB misses, indicating that the prefetch instruction triggers page walks on the AMD Zen 3 architecture.

insert NOP instructions between the PT-Baker 's array accesses or issue sleep system calls for several microseconds before the next iteration round starts executing. Using NOPs and sleeps allows us to control any cache pollution and memory bandwidth issues. In addition, the sleep system call allows reducing the energy consumption that the PT-Baker thread adds. Algorithm 2 code illustrates the user-level PT-Baker thread function design in a simple manner.

Similarly, in the case of the kernel-space PT-Baker, we insert NOP instructions between each leaf PTE accesses or a sleep instruction (or both) for a certain amount of time in microseconds before the next system call execution, to control cache pollution and energy consumption. Algorithms 3 and 4 show the design of the kernel-level thread PT-Baker approach.

4.4.5 Placing PT-Baker Thread

Another critical issue that needs to be addressed is PT-Baker's core placement within the microprocessor's chip. PT-Baker has to be microachitecture-aware and its thread placement needs to take into consideration the core and cache organization of the system. If the PT-Baker thread resides on the same core with the application main thread, utilizing Simultaneous Multithreading (SMT), it can preserve PTEs in the L2 cache and potentially even filling entries within the SMT shared TLB hierarchy. Otherwise, if the PT-Baker thread is placed on another core, they only share the LLC, and the PT-Baker thread can preserve PTEs inside it without affecting the private caches of the workload's thread. Figure 4.4 shows the CPU core and memory layout and PT-Baker thread's affinity options. As described above, page table entries can be found in LLC as well as in the L2 cache in AMD processors.

Figure 4.5 shows the cache and core layout of Ryzen 5900X processor. Each core has a private Data/Instruction L1 cache and a unified L2 cache. Depending on the number of cores, these are organized in groups called Core Complexes (CCX) that share a bank of LLC [7]. The Ryzen 9 5900X CPU (see Section 5.1 for details) has 2 core complexes

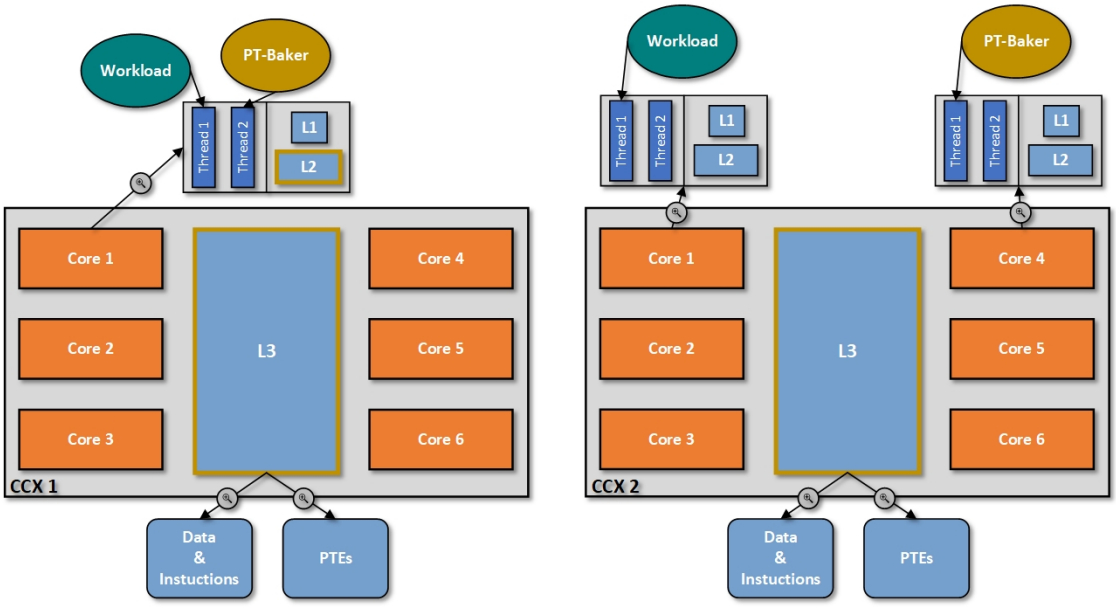


Figure 4.4: AMD CPU Zen 3 PT-Baker thread placement scenarios: PT-Baker can be placed on the same core using SMT to fill private caches, or on another core to utilize only the LLC.

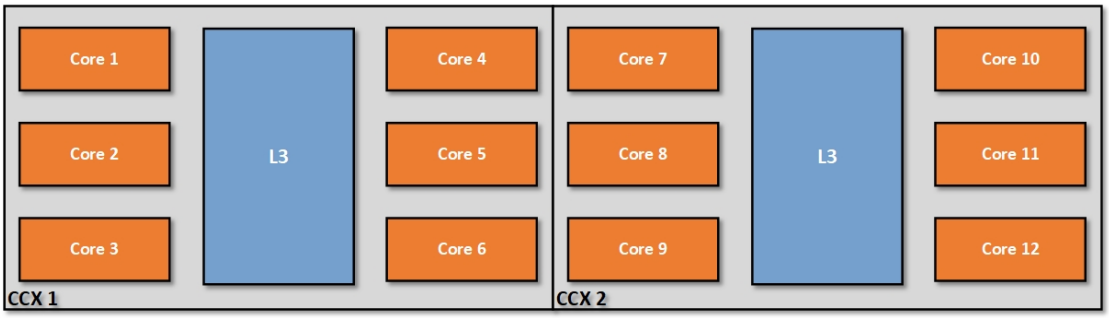


Figure 4.5: AMD Ryzen 5900X LLC and cores partitioning. Each group of 6 cores share 32MB LLC, thus PT-Baker must be on the same CCX chip to utilize the cache hierarchy.

with 6 cores each, with every CCX sharing 32MB of LLC. Therefore, a secondary helper thread can be either placed on a different core inside a core complex utilizing the shared LLC, or in the same core (SMT) with the main thread's execution, preserving the PTEs on both L2 and LLC, and even filling entries within the shared TLB hierarchy.

4.4.5.1 Reverse Engineering SMT Sharing in TLBs

To determine whether the TLB entries are shared between the two hardware threads, i.e., hyperthreads, of an SMT core we need to reverse engineer the lookup process of the TLB. AMD reports that TLB entries have an additional thread ID tag for validation so that TLB entries can be used only from the hardware thread that created the TLB entry [10]. Similarly to the previous reverse-engineering method, first we execute the TLB microbenchmark (Algorithm 5) with 2112 4KB pages (i.e., 2048 + 64 pages to fill the TLB) and count the total L2 TLB misses using the PAPI library. Then, we run the TLB microbenchmark with the same memory area, but now with two hardware threads placed on the same physical SMT core, to observe whether the number of TLB misses will remain the same between the two different executions. Ultimately, as we notice in Figure 4.6, the number of L2 TLB misses has increased significantly in the second experiment when

the two parallel threads are running. We confirm that TLB entries are indeed marked with a hardware thread id and are not shared between the two hardware threads on the same core, even if they belong to the same application or process. We conclude that PT-Baker cannot preserve TLB entries within a core's TLB hierarchy by spawning the helper thread on the same core using hyperthreading.

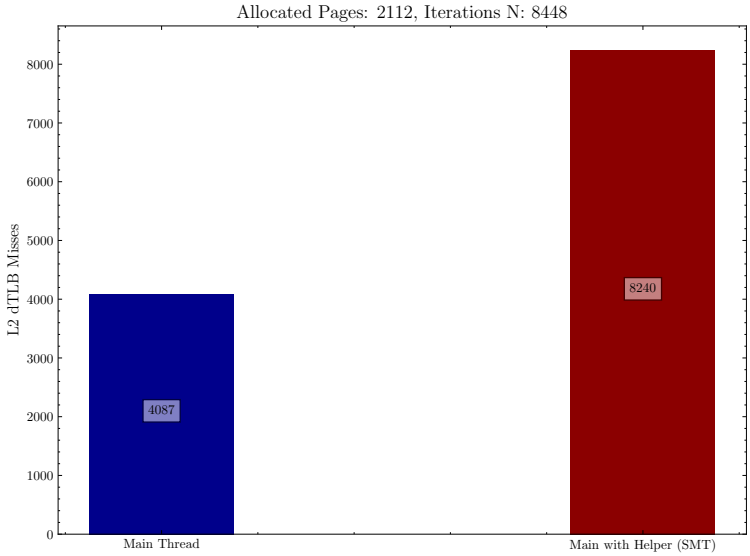


Figure 4.6: Reverse engineering thread ID tag validation of TLB entries. We observe increased L2 DTLB misses while accessing a shared allocated structure within the two hyperthreads of a core, thus TLB entries are not shared within the same core (SMT).

4.5 Discussion

As mentioned earlier, both user-level and kernel-level approaches require simple modifications at the application code. The proposed technique could also be designed and implemented in a more automated way at compile time; however, the user should provides hints to the compiler regarding when to start and stop the helper thread and which application memory regions should be touched by the helper thread.

Another approach could be creating an OS daemon thread which automatically identifies applications that experience a high number of TLB misses and main memory accesses due to page walks using hardware performance counters, and touches the application's memory regions to preserve the page table structures inside the cache hierarchy. This approach does not require any application modifications and avoids the need for recompiling applications.

Finally, another important factor is the power consumption especially when the PT-Baker operates in an aggressive manner; our results in Chapter 6 show that PT-Baker does not increase power consumption significantly.

5. EVALUATION METHODOLOGY

5.1 System Configuration

The system hardware we use for the evaluation includes a 12-core AMD Ryzen 9 5900X Zen 3 CPU with 64 GB of DDR4 memory. On the software side, we use the Linux kernel version 6.6.10, as provided by the Pop!_OS distribution, with the *gcc/g++* compiler versions 11.4.0, and the Linux *Perf* [16] version matching the Linux kernel. Additionally, we conduct evaluation experiments in a virtualized system, using a virtual machine with QEMU [14] version 6.2.0 and Kernel Virtual Machine (KVM) [3] enabled.

We use two setups in terms of memory pressure. The first one is a clean setup where only the target workload is running. The other one is a memory-pressured setup in which the stress-ng tool [5] runs concurrently with the target workload. We model three different levels of memory-pressure by configuring stress-ng with: (i) having 1 stress thread with *cache* parameter, (ii) having 2 stress thread with the *stream* parameter, and (iii) having 3 stress thread with the *stream*. Table 5.1 provides the overall system specifications in further detail.

While PT-Baker can reduce the latency of page table walks that may occur to memory regions mapped with any page size, i.e., base 4KB or huge 2MB that are supported transparently in the x86-64 architecture, in the evaluation of PT-Baker in this thesis we mainly use 4KB pages. This scenario resembles common execution conditions that may occur in long running servers in which huge pages are unavailable due to system aging and memory fragmentation [33]. Hence, applications eventually use base 4KB page mappings and thus experience frequent TLB misses that trigger long-latency page walks. We performed experiments using PT-Baker with 2MB pages without any fragmentation and we did not observe any performance improvements. As future work, we plan to perform experiments using PT-Baker and 2MB pages while varying memory fragmentation. We also plan to evaluate PT-Baker on systems with larger main memories using larger datasets, as prior work [9] has shown that for such systems and applications, page walks can occur frequently even when 2MB pages are used without any fragmentation.

Table 5.1: System hardware and software configuration.

CPU	AMD Ryzen 9 5900X
L1 I/D Cache	$12\ Cores \times 32KB$, 8-way set associative
L2 Cache	$12\ Cores \times 512KB$, 8-way set associative
L3 Cache	$2\ CCX \times 32MB$, 16-way set associative
RAM	64GB, DDR4 4000MHz
L1 DTLB	64 entries, fully associative
L2 DTLB	2048 entries, 16-way set associative
Linux Kernel	6.6.10
gcc/g++	11.4.0
Perf tool	6.6.10
QEMU	6.2.0

5.2 Workloads

For the evaluation of our approach we experiment with the following workloads:

- Graph500, List-based version (scale 24) [27],
- LibLinear, a linear classification workload with the *HIGGS* input set [17, 2],
- GUPS, the *RandomAccess* benchmark from HPCC (32 GB allocation) [4]
- Hashjoin, an implementation [8] of the well-known algorithm (Hashtable Size: 10^8, Iterations: 100, Outer Table Size: 10^7, Inner Table Size: 9×10^8),
- Canneal, a kernel from PARSEC (Input Nets: 6×10^7, Swaps: 3×10^4, Start Temperature: 2000, Temperature Steps: 15000) [15], and
- BC, a Betweenness Centrality algorithm from the GAPBS suite [13] with the twitter graph input set that the suite provides.

For all of the aforementioned workloads, PT-Baker thread only touches one memory region, which is the main allocated structure for each algorithm that produces the majority of TLB misses that trigger page walks. For those workloads that use more than one allocated data structures, such as Graph500, Hashjoin and GAPBS BC, we implemented the PT-Baker to access multiple data structures sequentially or even touch more than one structures in each iteration. However, we did not notice any additional performance improvement when touching more than one allocated data structure. Therefore, in the reported results PT-Baker touches a single memory region for those workloads as well.

Table 5.2: Workloads - Memory footprints of workloads and PT-Baker

Workload	Total Memory Footprint	PT-Baker Memory Footprint
Graph500 List Based	9.1GB	4.25GB
LibLinear	5.5GB	4.5GB
Parsec - Canneal	20.6GB	5.1GB
Hashjoin	16.5GB	13.4GB
GUPS	32GB	32GB
GAPBS BC	14GB	5.47GB

5.3 Hardware Performance Counters & Metrics

The metrics used in our analysis are generated by collecting results with the Linux Perf tool. This tool offers a range of pre-mapped events suitable for our measurements such as the number of page walks for data requests, L2 data TLB misses and total instructions executed. Apart from these predefined events that the Linux Perf tool provides, we can also use raw performance counters that are not directly mapped and offered by the tool. The AMD Zen 3 architecture contains performance counters that measure page walk hits in the cache hierarchy and main memory [11]. They are officially documented by AMD and can be measured using Perf. Table 5.3 shows all the counters used for the motivation

analysis presented in Chapter 3, as well as for the evaluation and performance analysis experiments presented in Chapter 6.

Table 5.3: Performance events used in *perf-stat* tool

ls_tablewalker.dside	Total data page table walks
dTLB-load-misses	Total data TLB load misses that trigger page walks
cpu/instructions/	Total instructions executed
power/energy-pkg/	Energy consumption of the CPU package
event=0x5B, mask=0x01	L2 local cache hit during page walks
event=0x5B, mask=0x12	L3 Cache hit on local or remote CCX during page walks
event=0x5B, mask=0x48	Access from DRAM or IO during page walks

6. RESULTS

In this chapter we measure the performance improvement of PT-Baker on a native system in both non-memory-pressured and memory-pressured execution conditions. Furthermore, we evaluate the performance improvement on a virtualized environment for various parameter configurations of PT-Baker. We then experiment with the PT-Baker core placement and perform a parameter sensitivity analysis using specific workloads. Finally, we provide power consumption measurements for some workloads on configurations that show performance improvement.

6.1 Native Execution

6.1.1 Native Execution in a Non-Memory-Pressured System

First, we tested PT-Baker on the native system without any LLC memory interference, both for the user-level and kernel-level approach, using the NOP and sleep instruction parameters. The PT-Baker helper thread runs on a different core than the application main thread, but within the same CCX. Figure 6.1 shows the performance improvement and Figure 6.2 shows percentage decrease for DRAM accesses during page table walks when using 4KB pages with PT-Baker. The results are normalized to the scenario of using 4KB pages without PT-Baker.

We observe that the PT-Baker thread speeds up most workloads. The Graph500 and LibLinear benchmarks show an increase in performance up to 5.9% and 4.1%, respectively using the user-level approach, with the kernel PT-Baker providing a greater performance improvement of up to 18% and 5.4%. For the hashjoin workload, we observed a performance improvement of 5.4% with the kernel-level PT-Baker. Finally, we also noticed a performance improvement for the GAPBS BC algorithm when we used the PT-Baker at user and kernel level, but only when using the NOP instructions parameter. However, PT-Baker does not provide performance improvements for Canneal from PARSEC and GUPS. We observe that Canneal does not experience a significant speedup even with 2MB pages. For GUPS, which is a very memory intensive microbenchmark, we observe that PT-Baker slightly degrades the application performance due to cache pollution by even attempting modestly to keep PTEs warm in the LLC. Finally, we compare our results with the execution time with 2MB pages (THP) without PT-Baker. While 2MB pages provide important performance improvements for almost all applications due to significant TLB miss reduction, we observe that PT-Baker is able to bridge the performance gap between 4KB and 2MB pages, particularly for Graph500.

6.1.2 Native Execution in a Memory-Pressured System

We now perform experiments on a memory-pressured system with LLC contention due to cache interference. Again, the PT-Baker helper thread is running on a different core than the application main thread, but within the same CCX. To put pressure on the memory system, we used the stress-ng tool with one thread with the cache parameter and also two and three threads with the stream parameter; with the latter configuration parameters, the tool allocates more memory and is overall more memory intensive on the cache hierarchy of the system. Figures 6.3 and 6.4 show the performance improvement with PT-Baker at

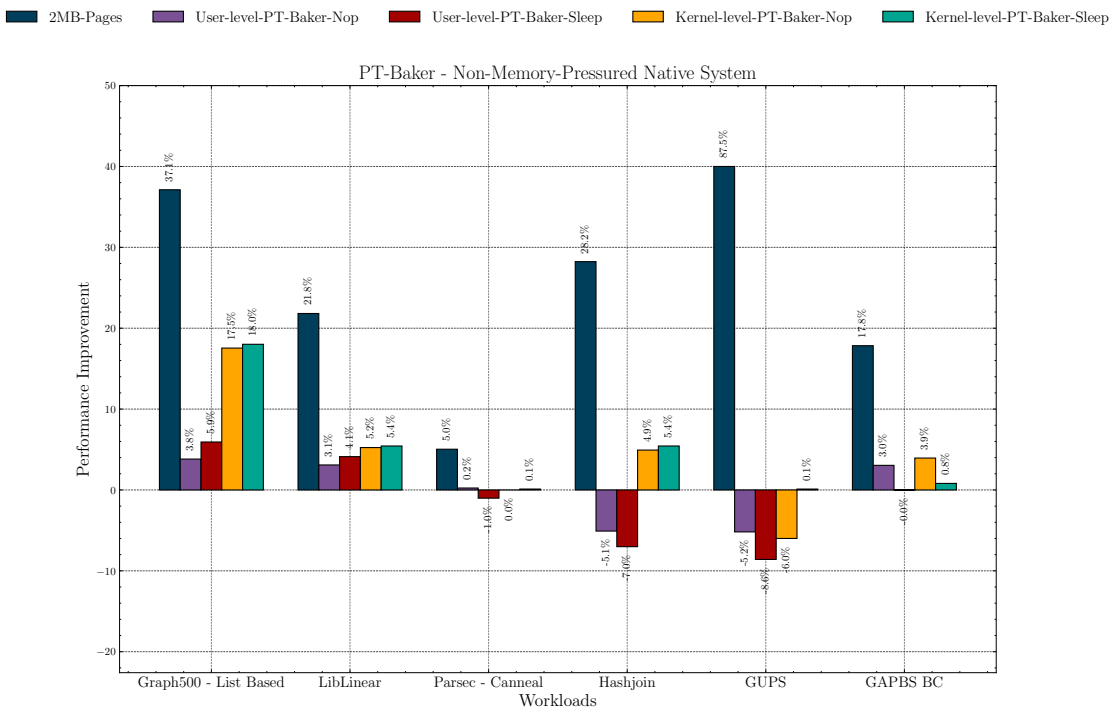


Figure 6.1: PT-Baker native execution performance improvement for both user-level approach and kernel-level approach with NOP instructions (greater than or equal to zero) and sleep instruction parameters.

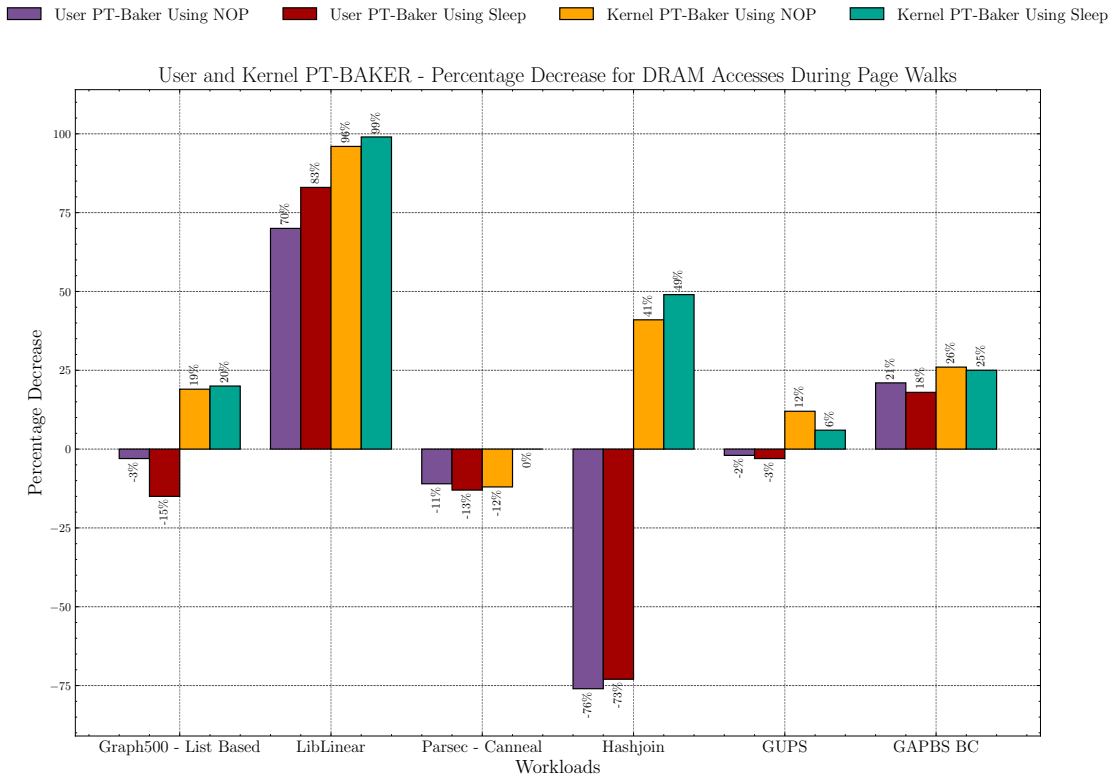


Figure 6.2: DRAM percentage decrease during page table walks on a native system using user-level and kernel-level PT-Baker with NOP and sleep instruction parameters.

user-level and at kernel-level using the sleep command parameter. It can be clearly seen that PT-Baker at the kernel-level again provides a greater performance improvement compared to PT-Baker at the user-level. We also perform additional measurements for the BC algorithm using the NOP command parameter for PT-Baker to reduce the aggressiveness

between page walk triggering. As shown in Figure 6.5, we found that the NOP approach, especially for the kernel-level method, leads to a performance improvement compared to the sleep parameter.

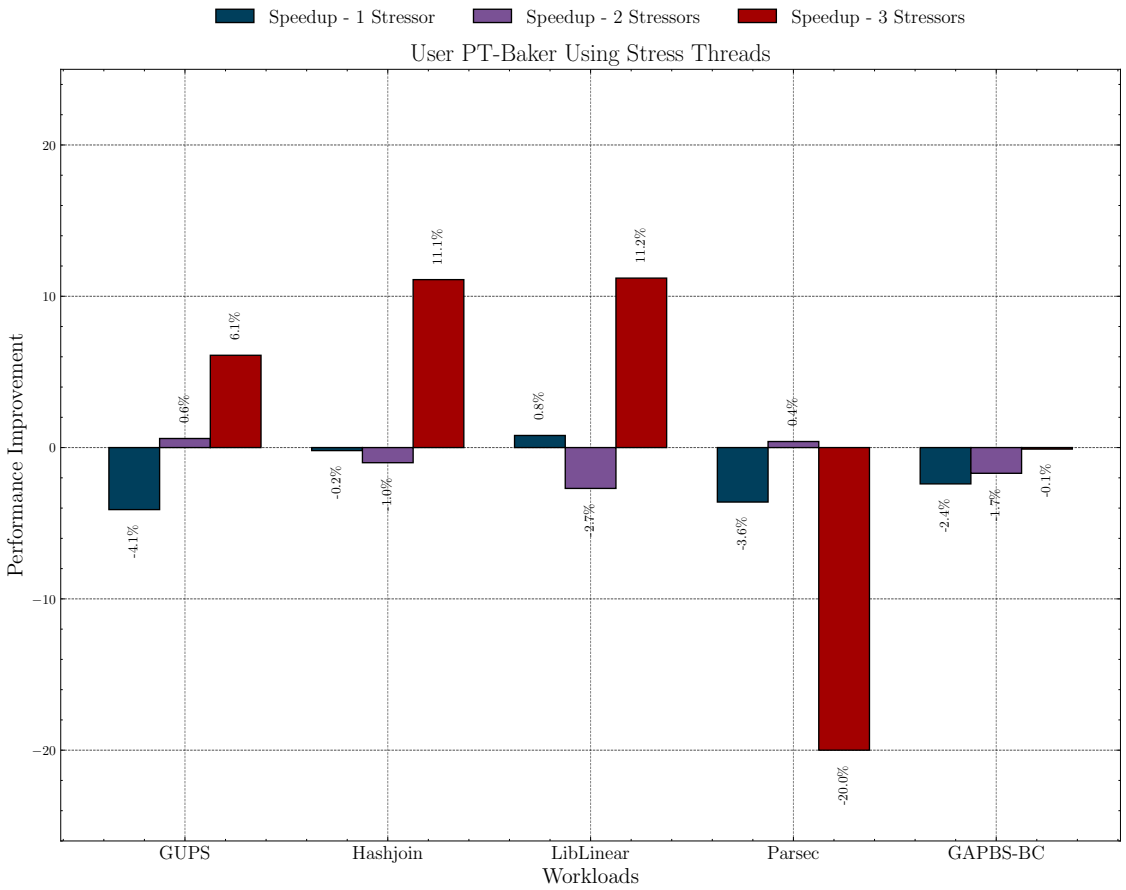


Figure 6.3: User-level PT-Baker performance improvement on a native memory-pressured system using the sleep instruction parameter.

6.2 Virtualized Execution

We also test our approach on a virtual machine using QEMU by running both the workloads and PT-Baker entirely at the guest level. Figure 6.6 shows the performance improvement achieved with both user and kernel PT-Baker for sleep and NOP instructions on different cores within the same CCX CPU chip. Figure 6.7 illustrates the performance improvement when running with two *stream* stress-ng stress threads. The results show that on a virtual machine, both in non-memory-pressured and memory-pressured systems, the percentage improvement is not as significant as with native execution for most workloads, except for Graph500 and LibLinear. While our thread attempts to keep the PTEs warm in the cache, it adds additional pressure in the LLC. In a native system, a page table walk requires up to 4 memory references, but in a virtual machine, this process requires up to 24 references, as shown in Figure 2.4. Further investigation of the sleep and NOP instruction parameters may be necessary to reduce cache pollution and ultimately accomplish additional memory address translation accelerations. Finally, we observe that the user-level PT-Baker approach for Graph500 with sleep system calls provides the most significant performance improvement, likely due to its additional assistance in prefetching data in the LLC and reducing data cache misses.

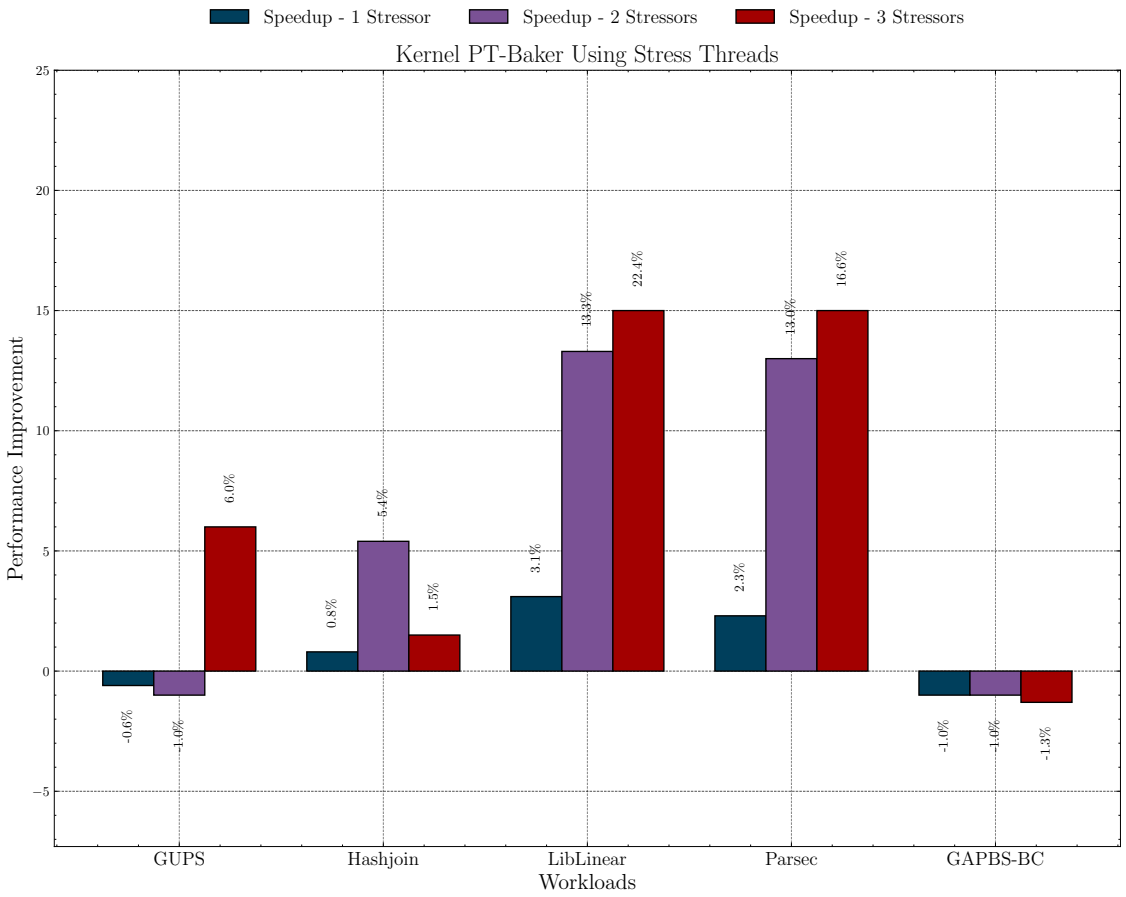
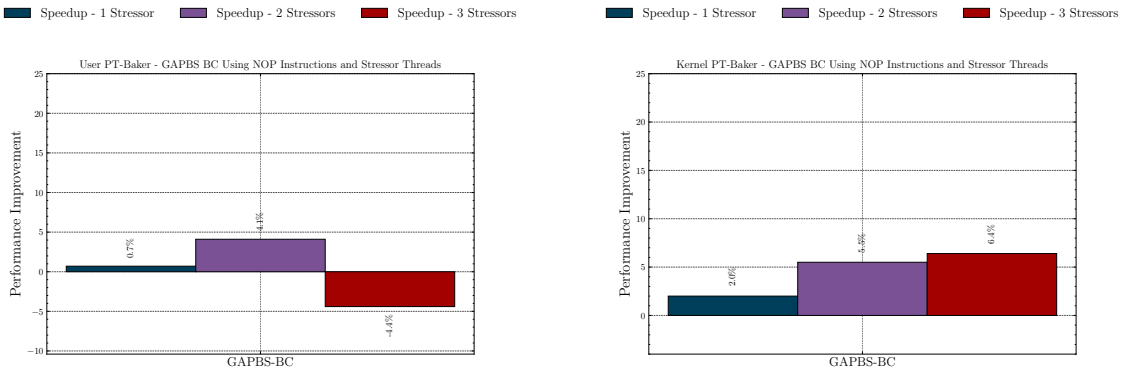


Figure 6.4: Kernel-level PT-Baker performance improvement on a native memory-pressured system using sleep instruction parameter.



(a) User PT-Baker (b) Kernel PT-Baker

Figure 6.5: GAPBS - BC workload using the NOP instruction parameter for user-level (left) and kernel-level (right) PT-Baker on a native memory-pressured system.

6.3 Sensitivity Analysis

In this section, we experiment with specific workloads using Simultaneous Multithreading (SMT), evaluate the effectiveness of the prefetch instruction for the user-level approach, and assess the sensitivity of NOP and sleep instruction parameters.

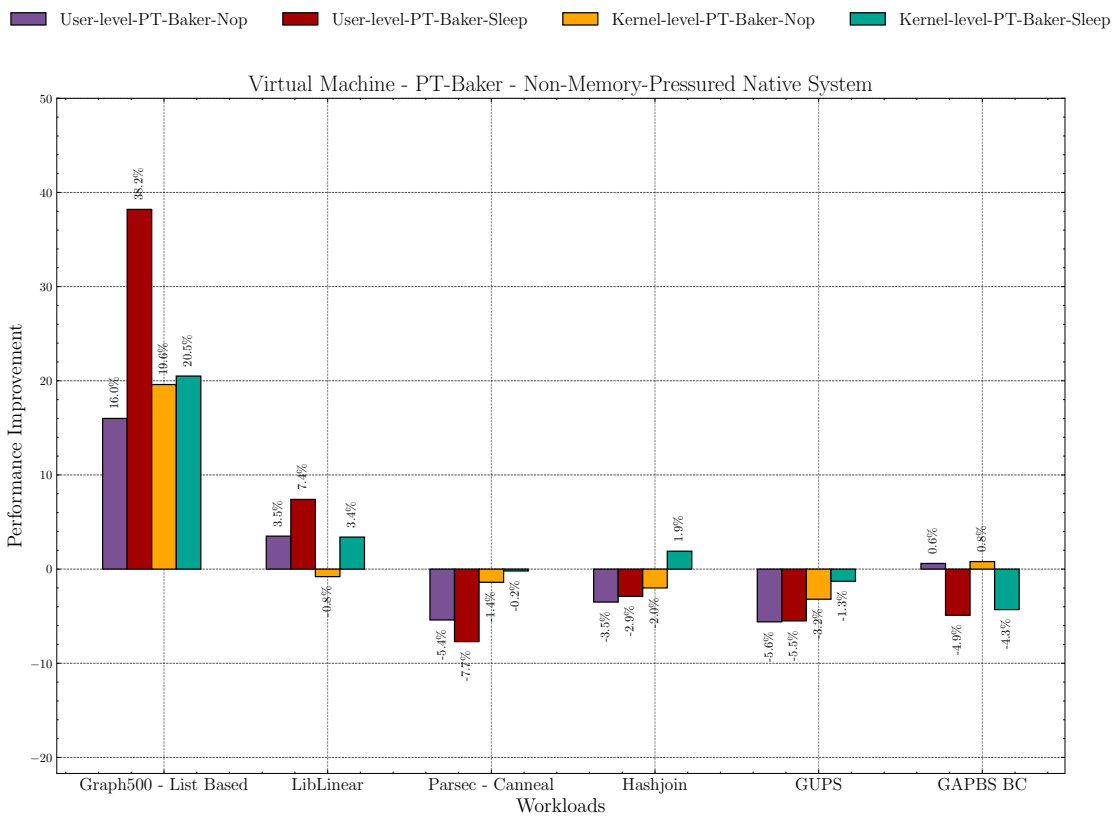


Figure 6.6: PT-Baker virtual execution performance improvement for both user-level approach and kernel-level approach with NOP instructions (greater than or equal to zero) and sleep instruction parameters.

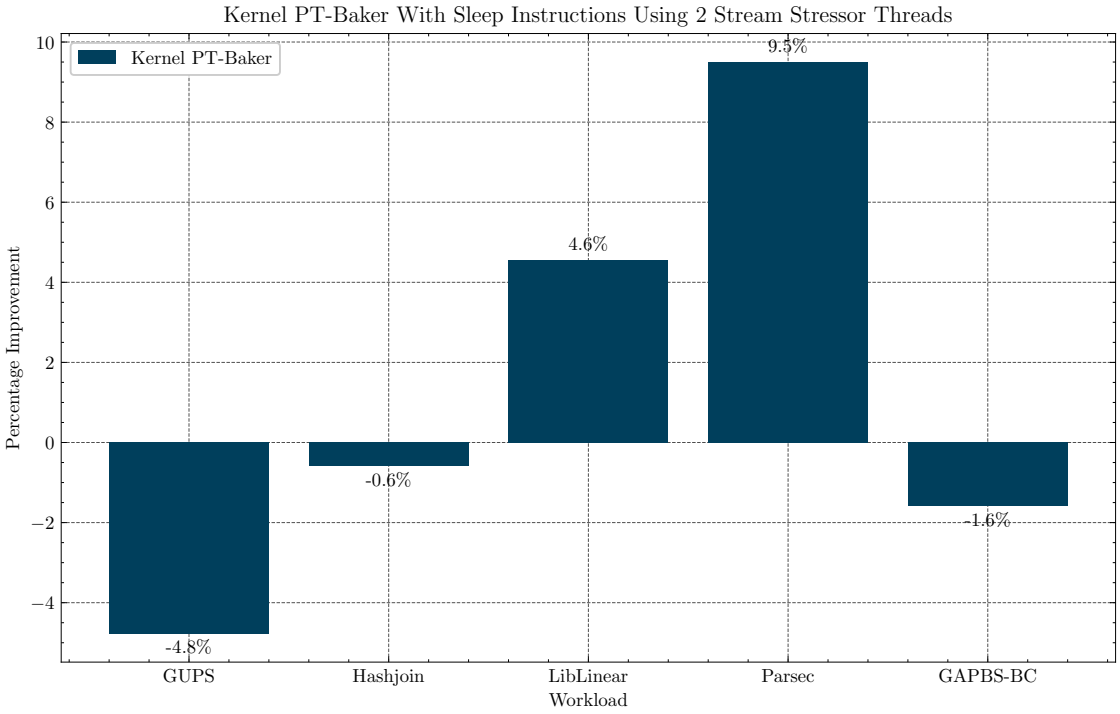


Figure 6.7: Kernel-level PT-Baker, using sleep instructions parameter, performance improvement on a virtual memory-pressured system.

6.3.1 Running PT-Baker using SMT

We now test the placement of the PT-Baker thread with sleep instruction on the same core with the application's main thread, using the microprocessor's SMT capabilities. Figure 6.8 summarizes the performance improvement results, normalized to when using 4KB pages without PT-Baker. We observe that the kernel-level approach provides similar but slightly smaller performance improvements compared to placing the helper thread on a different core within the same CCX. In contrast, the user-level approach provides less performance benefits with respect to placing the helper thread on a different core. This occurs because the user-level approach introduces additional L1 and L2 cache pollution by fetching also application data. This behavior is more pronounced for the GAPBS BC workload, as we observe that placing the PT-Baker thread on the same core with the application's main thread does not bring any performance improvement.

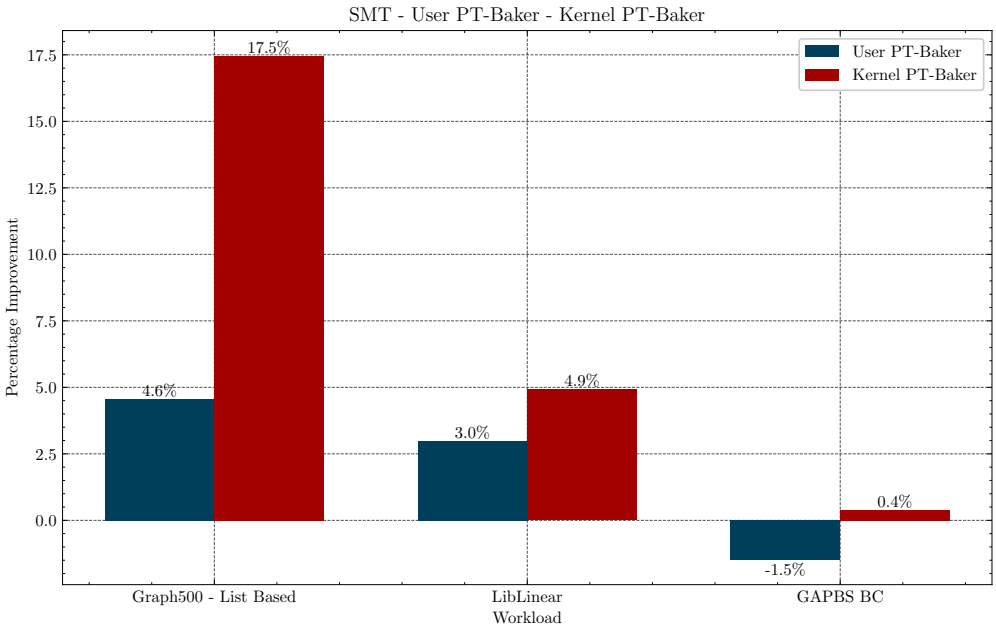


Figure 6.8: Simultaneous Multithreading (SMT) evaluation on Graph500 - List Based, LibLinear and GAPBS BC for user-level and kernel-level PT-Baker with sleep instruction parameter.

6.3.2 Testing User PT-Baker using the Builtin prefetch Instruction

So far we have used regular load instructions, i.e., the variable assignment method, for accessing the workload's allocated memory region with the user-level PT-Baker approach. We now test the user-level approach using the prefetch instruction that we described in Chapter 4. Figure 6.9 shows the performance improvement results, normalized to when using 4KB pages without PT-Baker. We observe that Liblinear has reduced performance improvement compared to using regular load instructions. However, the performance of Graph500 and GAPBS BC is further improved, with Graph500 showing an increase from a maximum of 5.9% to 7.9%, while the maximum improvement for GAPBS BC remains at 3%. Overall, we notice that the prefetch instruction is more memory-aggressive compared to using regular load instructions through a typical variable assignment mode.

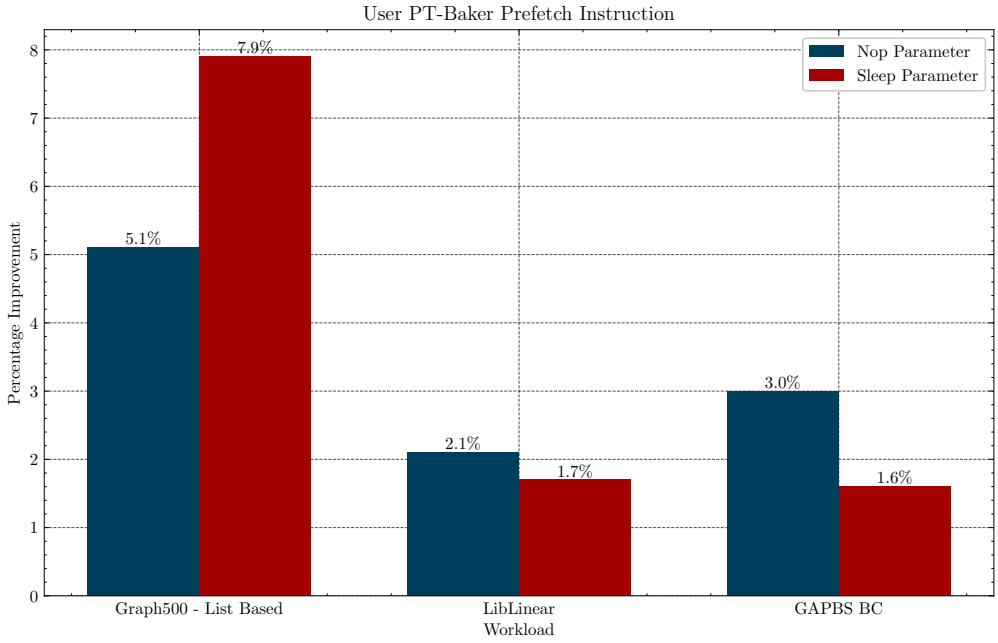


Figure 6.9: Using prefetch instruction for user-level approach on LibLinear, Graph500 and GAPBS BC workloads with the Sleep instruction parameter.

6.3.3 Testing PT-Baker using NOP and Sleep Instructions

We conduct a sensitivity analysis regarding the number of NOP instructions and the duration of the sleep system calls in microseconds. Figures 6.10 and 6.11 show the performance improvement results, normalized to when using 4KB pages without PT-Baker. We observe that the user-level approach is much more sensitive to the NOP and sleep parameters in comparison to the kernel-level approach, because the user-level approach fetches additional application data.

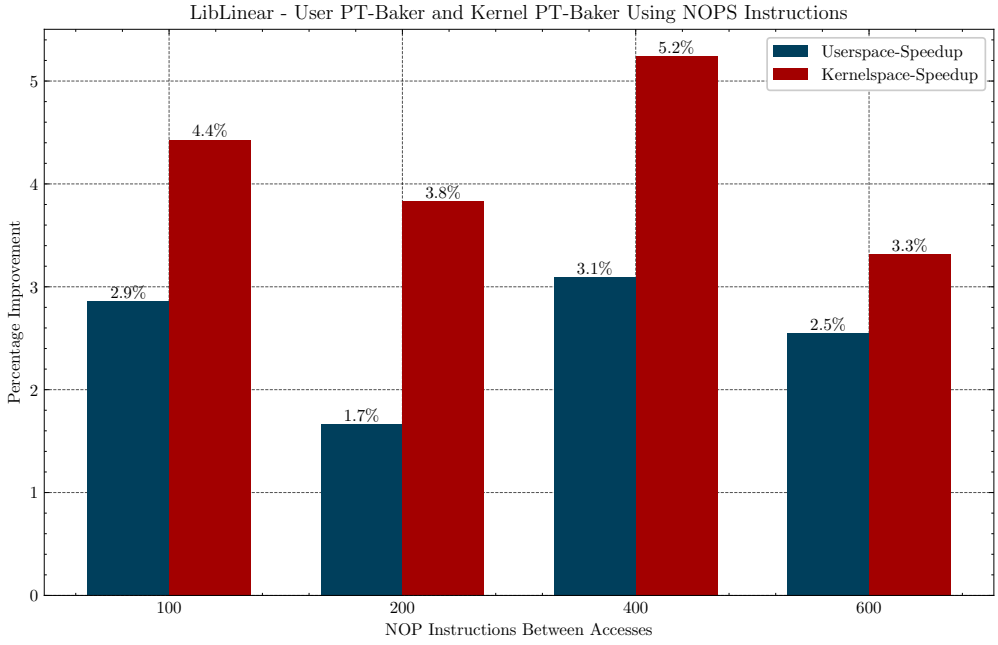


Figure 6.10: Sensitivity analysis for number of NOP instructions parameter on LibLinear using user-level and kernel-level PT-Baker.

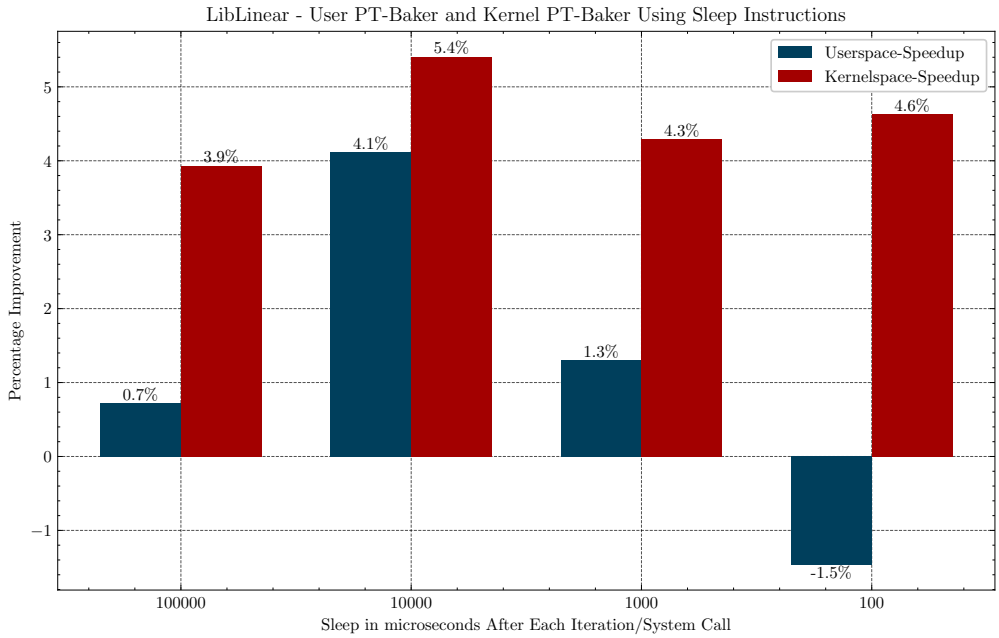


Figure 6.11: Sensitivity analysis for sleep instruction parameter in microseconds on LibLinear using user-level and kernel-level PT-Baker.

6.4 Power Consumption

Finally, we measure the power consumption of the CPU package for specific workloads and configurations that provide significant performance improvement. More specifically, we evaluate the power consumption when executing LibLinear, GAPBS BC and Hashjoin. Figure 6.12 shows the power consumption in Watts and the percentage increase on the AMD Ryzen 9 5900X package, which has a TDP of 105 watts. Power consumption is measured using the perf-stat tool with the *power/energy-pkg/* event, as described in Table 5.3. Overall, the kernel-level PT-Baker consumes slightly more power than the user level approach. The use of NOP instructions only serves to reduce the aggressiveness of PT-Baker thread's pressure to the memory cache hierarchy. In contrast, the sleep system calls can also assist to lower the power consumption of the CPU, enabling other processes to execute during the time quantum that the PT-Baker thread is in a blocked state.

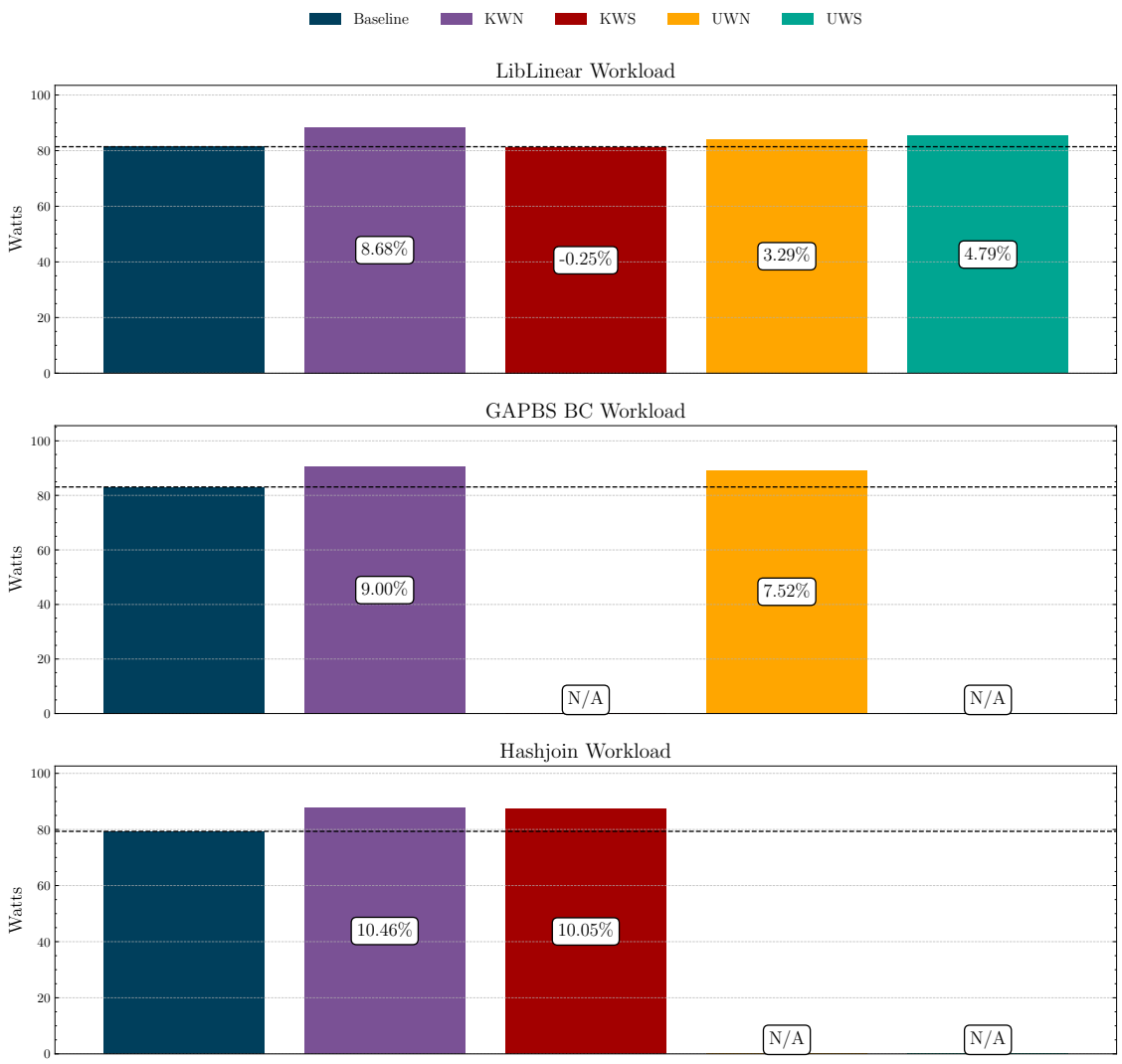


Figure 6.12: Power Consumption using kernel-level approach PT-Baker thread and user-level PT-Baker thread. Kernel-level PT-Baker thread using NOP instructions KWN. Kernel-level PT-Baker thread using sleep KWS. User-level PT-Baker thread using NOP instructions UWN. User-level PT-Baker thread using sleep instruction UWS.

7. RELATED WORK

In the last few years, there has been a massive amount of research aiming at reducing the overheads of address translation. In this chapter we focus on closely related work that focuses on reducing the latency of page table walks by either preserving page table structures in the cache hierarchy or directly storing TLB entries within the cache memory hierarchy. In contrast to PT-Baker that is a system-level approach, prior approaches typically require architectural and microarchitectural design modifications and thus cannot directly improve the performance of existing, real systems.

7.1 Prioritizing PTEs in the Cache Hierarchy

To manage better user-level and system-level memory references in the LLC, Wu and Martonosi [32] proposed a hardware methodology in which a different cache replacement policy applies to each memory reference type. That approach ensures that cache entries are managed based on their origin, resulting in more effective LLC locality.

Kwon et al. [23] proposed to use a dynamically reserved LLC space for pinning page table entries that are most frequently fetched from the main memory. That approach also requires hardware support and uses the LLC misses and L2 TLB MPKI events to determine the space to be allocated in the LLC during the execution of a workload, thus balancing the page table entries and the data in the cache memory.

Vasudha et al. [31] proposed several optimizations to address translation cache policies aimed at extending the retention of page table entries in both the L2 and LLC caches, alongside the introduction of hardware prefetching for address translations. For the L2 cache, the authors modified the DRRIP eviction policy to insert page table leaf nodes at the lowest eviction priority, while all the previous page table nodes of the corresponding page walk are assigned the highest eviction priority into the L2 cache. Similarly, the authors proposed enhancements to the SHiP and Hawkeye cache policies in the LLC to preserve page table entries in the cache for more time.

Park et al. [28] proposed flattening the tree structure of the page table to reduce the number of memory accesses during page walks. That approach requires hardware modifications in the page table walkers and the page walker caches (PWCs), while on the software side it modifies the kernel so that the 3rd and 2nd levels of the page table are flattened accordingly. In addition, the authors proposed a cache management technique that prioritizes the page table entries in the LLC. Similarly to the previous approaches, the proposed technique requires hardware support to enforce the prioritization and detect phases with high TLB and cache misses. To gain confidence about the performance potential of prioritizing the page table entries in the LLC, the authors state that they implemented an approach similar to PT-Baker and only mention that they managed to achieve 5% speedup for Graph500 on a real Intel microprocessor, without providing any design and implementation details, or evaluation results. In contrast, in this thesis we comprehensively present and evaluate the PT-Baker approach, taking into consideration various important implementation aspects.

Finally, Margaritov et al. [25] proposed PTEMagnet, a software technique for reducing the latency of page walks in cloud environments, by improving locality of host PTEs in public cloud environments. Their approach involves a custom memory allocator that reserves

eight-pages of continuous guest physical memory. As a result, the host's memory footprint is drastically reduced, resulting in better cache locality and reduced latency during page walks. In contrast, our approach can be beneficial for both native and virtualized environments, and avoids any changes to the memory allocator.

7.2 Storing TLB Entries in the Cache Hierarchy

Ryoo et al. [29] introduced an addressable, four-way set associative L3-TLB structure within the main memory, known as *POM-TLB*, for storing TLB entries. That approach takes advantage of caching TLB entries in the L2 and L3 caches as well, by substantially decreasing the frequency of page walks and reducing the virtual address translation latency.

Similarly, Marathe et al. [24] proposed *CSALT*, a hardware-based cache partitioning design, to maximize the hit rate of data and TLB entries. Using the Mattson's Stack Distance algorithm to estimate the hit/miss ratio and a formula to measure the total hit ratio in the cache called Marginal Utility, *CSALT* attempts to optimize the capacity of data and TLB entries inside the cache memory.

Kanellopoulos et al. [19] presented an approach named *Victima*, that stores clusters of TLB entries in the L2 cache. By using a predictor mechanism that recognizes slow address translations processes based on frequency and cost weight along with a TLB-aware cache policy, *Victima* constructs TLB clusters within the L2 cache, enabling direct access to the corresponding TLB entry.

Finally, recent work by Kotra et al. [22] and Jaleel et al. [18] demonstrated that GPUs also suffer from high TLB misses and poor page walk latency, which degrade the overall GPU performance for memory intensive workloads. They proposed techniques that enable TLB entries to be stored within the memory hierarchy of the host and the device to reduce the of latency of page walks.

8. CONCLUSION AND FUTURE WORK

The growing memory requirements in today's applications are putting extreme pressure on the TLB, leading to a significant increase in costly address translations in virtual memory. In Section 3 we conducted a performance analysis for a series of workloads and showed that a significant percentage of page walks result accessing the main memory. In Section 4 we proposed a user-level and a kernel-level approach called PT-Baker thread that periodically iterates through the page table levels in order to keep the page table structures warm within the cache hierarchy and reduce the latency of page table walks. We tested our approach on both native and virtualized systems, as well as on a memory pressured system. We showed that PT-Baker can bring significant performance improvements for some applications. Finally, we performed a sensitivity analysis regarding the thread placement, the use of the prefetch instruction to trigger page table table walks at the user level, and the use of NOP instructions and sleep system calls to throttle the aggressiveness of PT-Baker.

In our future work, we intend to implement a dynamic system-level daemon that can access other processes' page tables without requiring users to integrate this technique into their workloads (as described in Section 4.5). Furthermore, we plan to evaluate PT-Baker on other systems, such as Intel and ARM processors. We will also conduct experiments on systems with larger memory capacities, using 2MB pages and varying levels of fragmentation. While this thesis is focused on single-threaded workloads, we will also consider multi-threaded workloads with a broader range of applications. Finally, the improvements of PT-Baker in virtualized execution are less pronounced, due to the 2D page walks that introduce more memory references and result in increased cache pressure. The execution of PT-Baker in virtualized execution requires further exploration through fine-tuning the various parameters. An alternative promising technique to improve the performance in virtualized environments is implementing PT-Baker at the hypervisor level, in which case PT-Baker will transparently manage and access the host's virtual memory and trigger only one-dimensional page walks.

ABBREVIATIONS - ACRONYMS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
TLB	Translation Lookaside Buffer
DTLB	Data Translation Lookaside Buffer
MMU	Memory Management Unit
CR3	Control Register 3
PGD	Page Global Directory
PMD	Page Mid-level Directory
PUD	Page Upper Directory
PDE	Page Directory Entry
PTE	Page Table Entry
gVA	guest Virtual Address
hPA	host Physical Address
DRAM	Dynamic Random-Access Memory
LLC	Last Level Cache
CCX	Core Complex
NOP	No Operation
PWC	Page Walker Cache
SMT	Simultaneous Multithreading
TDP	Thermal Design Power

APPENDIX A. SOFTWARE ARTIFACT

The code and the scripts that were used in this thesis can be found in the following link:
`https://github.com/Aggelos561/Thesis-PT-Baker`

BIBLIOGRAPHY

[1] Five-level page tables. https://lwn.net/Articles/717293/.

[2] Libsvm data: Classification, regression, and multi-label. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#HIGGS.

[3] Qemu kernel virtual machine. https://wiki.qemu.org/Features/KVM.

[4] Randomaccess - giga updates per second (gups). https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html.

[5] Stress-ng, a tool to load and stress a computer system. https://manpages.org/stress-ng.

[6] Transparent hugepage support. https://docs.kernel.org/admin-guide/mm/transhuge.html.

[7] Amd ccx definition. https://www.tomshardware.com/reviews/amd-ccx-definition-cpu-core-explained,6338.html, 2021.

[8] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 515–528. IEEE Press, 2020.

[10] AMD. White paper, speculation behavior in amd micro-architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf.

[11] AMD. Open-source register reference for amd family 17h processors models 00h-2fh. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/56255_OSRR.pdf, 2018.

[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *SIGARCH Comput. Archit. News*, 38(3):48–59, jun 2010.

[13] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[15] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[16] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[17] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.

[18] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems. *ACM Trans. Archit. Code Optim.*, 16(1), mar 2019.

[19] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1178–1195, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 66–78, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2014.

[22] Jagadish B. Kotra, Michael LeBeane, Mahmut T. Kandemir, and Gabriel H. Loh. Increasing gpu translation reach by leveraging under-utilized on-chip resources. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1169–1181, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Osang Kwon, Yongho Lee, and Seokin Hong. Pinning page structure entries to last-level cache for fast address translation. *IEEE Access*, 10:114552–114565, 2022.

[24] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. Csalt: Context switch aware large tlb. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–462, 2017.

[25] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. Ptemagnet: fine-grained physical memory reservation for faster page walks in public clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 211–223, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[27] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19(45-74):22, 2010.

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

[29] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 469–480, 2017.

[30] Eliot H Solomon, Yufeng Zhou, and Alan L Cox. An empirical evaluation of pte coalescing. In *Proceedings of the International Symposium on Memory Systems*, pages 1–16, 2023.

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

[32] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–11, 2011.

[33] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.