Improving Address Translation Performance in Virtualized Multi-Tenant Systems

Artemiy Margaritov

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2021

Abstract

With the explosive growth in dataset sizes, application memory footprints are commonly reaching hundreds of GBs. Such huge datasets pressure the TLBs, resulting in frequent misses that must be resolved through a page walk – a long-latency pointer chase through multiple levels of the in-memory radix-tree-based page table. Page walk latency is particularly high under virtualization where address translation mandates traversing two radix-tree page tables in a process called a nested page walk, performing up to 24 memory accesses. Page walk latency can be also amplified by the effects caused by the colocation of applications on the same server used in an attempt to increase utilization. Under colocation, cache contention makes cache misses during a nested page walk more frequent, piling up page walk latency. Both virtualization and colocation are widely adopted in cloud platforms, such as Amazon Web Services and Google Cloud Engine. As a result, in cloud environments, page walk latency can reach hundreds of cycles, significantly reducing the overall application's performance. This thesis addresses the problem of the high page walk latency by ① identifying the sources of the high page walk latency under virtualization and/or colocation, and ② proposing hardware and software techniques that accelerate page walks by means of new memory allocation strategies for the page table and data which can be easily adopted by existing systems.

Firstly, we quantify how the dataset size growth, virtualization, and colocation affect page walk latency. We also study how a high page walk latency affects performance. Due to the lack of dedicated tools for evaluating address translation overhead on modern processors, we design a methodology to vary the page walk latency experienced by an application running on real hardware. To quantify the performance impact of address translation, we measure the application's execution time while varying the page walk latency. We find that under virtualization, address translation considerably limits performance: an application can waste up to 68% of execution time due to stalls originating from page walks. In addition, we investigate which accesses from a nested page walk are most significant for the overall page walk latency by examining from where in the memory hierarchy these accesses are served. We find that accesses to the deeper levels of the page table radix tree are responsible for most of the overall page walk latency.

Based on these observations, we introduce two address translation acceleration techniques that can be applied to any ISA that employs radix-tree page tables and nested page walks. The first of these techniques is Prefetched Address Translation

(ASAP), a new software-hardware approach for mitigating the high page walk latency caused by virtualization and/or application colocation. At the heart of ASAP is a lightweight technique for directly indexing individual levels of the page table radix tree. Direct indexing enables ASAP to fetch nodes from deeper levels of the page table without first accessing the preceding levels, thus lowering the page walk latency. ASAP is fully compatible with the existing radix-tree-based page table and requires only incremental and isolated changes to the memory subsystem.

The second technique is PTEMagnet, a new software-only approach for reducing address translation latency under virtualization and application colocation. Initially, we identify a new address translation bottleneck caused by memory fragmentation stemming from the interaction of virtualization, application colocation, and the Linux memory allocator. The fragmentation results in the effective cache footprint of the host page table being larger than that of the guest page table. The bloated footprint of the host page table leads to frequent cache misses during nested page walks, increasing page walk latency. In response to these observations, we propose PTEMagnet. PTEMagnet prevents memory fragmentation by fine-grained reservation-based memory allocation in the guest OS. PTEMagnet is fully legacy-preserving, requiring no modifications to either user code or mechanisms for address translation and virtualization.

In summary, this thesis proposes non-disruptive upgrades to the virtual memory subsystem for reducing page walk latency in virtualized deployments. In doing so, this thesis evaluates the impact of page walk latency on the application's performance, identifies the bottlenecks of the existing address translation mechanism caused by virtualization, application colocation, and the Linux memory allocator, and proposes software-hardware and software-only solutions for eliminating the bottlenecks.

Lay summary

The average size of a dataset processed by modern applications is growing. However, not all existing hardware mechanisms exploited by modern processors are designed in a way to tolerate the dataset size growth without additional performance loss. One of such mechanisms is virtual memory. Virtual memory is a technique that makes programming easier by providing an abstraction for managing memory. For each memory location, virtual memory assigns a virtual address that is translated to the actual physical address on each access to the memory location. The process of translating a virtual address to a physical is called address translation. Modern hardware is optimized for performing fast address translation for a small set of frequently-used memory locations whose addresses are accumulated in a hardware buffer. For all the data addresses beyond the size of that buffer, the translation is performed through a page walk – a long-latency chain of multiple serialized accesses to memory, slowing down the application execution. Thus, the larger the application's dataset size, the more translations are performed by time-consuming page walks, increasing the overhead of virtual memory. As a result, the performance of virtual memory poorly scales with increasing dataset sizes.

Meanwhile, independently of the dataset size growth, page walk latency is considerably increased by two techniques whose popularity is currently growing. These techniques are ① the virtualization technology allowing to partition hardware resources in a security-preserving manner and ② application colocation used to increase utilization of the hardware. Both techniques are widely adopted by cloud computing platforms.

As a result, a combination of the growth in dataset size and the expansion of virtualization and application colocation poses a serious challenge for low-overhead virtual memory. In response, this thesis aims to design page walk latency reduction techniques that can be easily incorporated into the existing systems. To that end, we analyze how a change in page walk latency affects the overall performance and propose two new low-effort techniques for reducing page walk latency of large-dataset applications running under virtualization.

Acknowledgements

First and foremost, I would like to express my gratitude and appreciation to my advisor, Prof. Boris Grot. Without his guidance, technical expertise, and unconditional support, this thesis would not have been written. Being a remarkable researcher, Boris helped me to develop skills essential for doing research. Being always available for a brainstorming session, Boris patiently listened to all my ideas, taught me how to critically analyze them, and helped to shape them into meaningful research. I am very happy (and proud) that I was his student.

I am grateful to Prof. Vijay Nagarajan, Prof. Hugh Leather, Prof. Pramod Bhatotia, and my second advisor Prof. Nigel Topham for providing insightful feedback on my works in the annual review panels. I thank Prof. Edouard Bugnion for thought-provoking discussions and valuable comments on my research. I am also extremely grateful to my viva examiners, Prof. Dan Tsafrir and Prof. Antonio Barbalace, for their very helpful comments and suggestions.

I am fortunate to have had the opportunity to work with and learn from my fellow students and collaborators, Dmitrii Ustiugov and Amna Shahab, who assisted me in writing papers. Together we shared a lot of memorable moments, went through stressful paper submissions, ranted about one thing and another during lunch hours, and argued with a reviewer which claimed that our proposal is an unnecessarily complex solution.

My sincere thanks must also go to my mentors from Arm Ltd., Rekai Gonzalez-Alberquilla and Reiley Jeyapaul, who supervised me as part of the Arm Industrial CASE programme.

I would be remiss if I did not thank my friend, Priyank Faldu, who went through hard times with me, cheered me on, and celebrated each accomplishment. I will certainly remember the never-ending discussions, not necessarily technical, from the beginning to the end of this laborious but pleasant path.

I thank the faculty members, staff, and students of the School of Informatics for their help and support. I would like to especially thank Antonios Katsarakis, Siavash Katebzadeh, and Vasilis Gavrielatos for their valuable help and support, both technical and otherwise.

I would like to express my gratitude to Arm Ltd., the School of Informatics, EPSRC, and Google, who funded me throughout my studies.

Finally, and most importantly, my deep and sincere gratitude goes to my family for their continuous and unparalleled love, care, and support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- A. Margaritov, D. Ustiugov, E. Bugnion, B. Grot. ”Prefetched Address Translation”. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO).* 2019 [1]

- A. Margaritov, D. Ustiugov, S. Shahab, B. Grot. ”PTEMagnet: Fine-grained Physical Memory Reservation for Faster Page Walks in Public Clouds”. In *Proceedings the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2021 [2]

(*Artemiy Margaritov)*

To my mother, Margarita. Her support, encouragement, and constant love have sustained me throughout my life.

Contents

Chapter 1

Introduction

1.1 Virtual Memory

Virtual memory is a technique that simplifies programming and provides isolation and security in modern computer systems. For each memory page, virtual memory creates a mapping between the virtual and physical addresses of each application. The mapping is stored in a separate auxiliary data structure called *page table (PT)*. A PT resides in memory together with the data. On modern x86-64 and AArch64 systems, a PT is organized as a radix-tree. The root node of such a radix-tree is called *root of the PT*, leaf nodes are called *page table entries (PTEs)*, and the rest of the nodes in the radix-tree are called *intermediate page table nodes*. PTEs store the actual physical addresses while other nodes store a pointer to a node in the next radix-tree level. x86-64 processors, which currently dominate the server market, employ a 4-level radix-tree PT organization and support a 5-level radix-tree PT [3] for large-memory setups (e.g. enabled by emerging memory technologies such as Intel's 3D XPoint [4]). Meanwhile, emerging AArch64-based server processors also support a 4-level PT. Both x86-64 and AArch64 architectures employ 4KB base pages and support several different additional large page sizes. In this thesis, we focus on systems with 4KB pages as large pages don't fit the needs of all applications (see Section 2.1.2.1 for more details).

While simplifying programming, virtual memory introduces a tax on performance due to the need for virtual-to-physical address translation on each memory access. Address translation is performed by a *page walk* – a long-latency pointer chase through the in-memory radix-tree-based PT. During a page walk, the levels of the radix-tree-based PT must be traversed one by one, incurring high latency due to serialized accesses to the memory hierarchy. To reduce the number of page walks, today's processors tend to employ several hardware features to accelerate address translation, including hardware page walkers and caching translations in Translation Lookaside Buffers (TLBs) or intermediate PT nodes in page walk caches (PWCs).

Address Translation under Virtualization. Virtualization is a technique allowing to run the software in a virtual instance of a system in a layer abstracting from the actual hardware, the operating system (OS) installed on that hardware, and the current state of the execution environment. A virtual instance of a system is called *a virtual machine*. An OS running in a virtual machine is called *a guest OS* whereas the OS installed on the actual hardware is called *a host OS*. Virtualization is very useful for ① simplifying software deployment (e.g. by providing a snapshot of the whole system with pre-installed software) and ② partitioning the compute resources in a security-preserving manner (e.g. allowing multi-tenancy among clients that cannot be trusted).

While simplifying deployment and enabling multi-tenancy in a security-preserving manner, virtualized setups incur a particularly high overhead of address translation. Under virtualization, resolving a TLB miss mandates traversing two guest and host PTs in a process called a *nested page walk*. In a nested page walk on a CPU architecture with 4-level PTs, resolving each access to the guest PT requires traversing the entire host PT, resulting in up to 24 accesses to the memory hierarchy (versus up to 4 accesses in native execution).

Hardware Acceleration of Address Translation. To reduce the number of page walks, today's processors tend to employ several hardware features to accelerate address translation, including hardware page walkers and caching translations in TLBs or intermediate PT nodes in PWCs.

1.2 Challenges for Virtual Memory

The core of the address translation mechanisms employed by modern x86-64 and AArch64 processors has not changed for a few decades. In contrast, the software and the execution environment have significantly evolved. Below we list the key trends changing the software and the execution environment and affecting the requirements for the address translation mechanism.

- **Growing dataset sizes.** Massive in-memory datasets are a staple feature of many applications, including databases, key-value stores, data, and graph analytics frameworks. Recent research article [5] has shown that large dataset sizes of modern applications make hardware caching techniques inefficient. The large – and rapidly growing – dataset sizes, coupled with irregular access patterns, in many of these applications result in frequent TLB misses, triggering a large number of page walks.

- **Ubiquitous use of virtualization.** The benefits provided by virtualization, such as enhancing security and simplifying deployment, have accelerated the adoption of virtualization, in particular in cloud platforms. However, these benefits come at the price of having longer page walks which is critical for applications experiencing a large number of page walks.

- **Aggressive application colocation.** To increase utilization of available hardware resources, virtual machines and/or applications are aggressively colocated on the same server for better CPU and memory utilization [6], [7]. As a result of colocation, the applications experience Last-Level Caches (LLC) contention which lowers the chances of having a hit into LLC during a page walk and increases page walk latency.

One can see that the combination of these trends significantly affects the overhead of the address translation mechanism. While large dataset sizes increase the frequency of page walks, the use of virtualization and workload colocation increases page walk latency. As a result, in a system employing both virtualization and application colocation, a data-intensive application can face a large address translation overhead as the negative effects amplify each other.

1.2.1 Cloud Computing: a Case Where the Challenges Emerge

Cloud computing is a service providing access to on-demand computing resources. Cloud computing offers great flexibility through resource scaling, high resource utilization, and low operating costs. Businesses are deploying their services in the public cloud to enjoy these benefits. The global cloud computing market is anticipated to grow from $350 billion in 2020 to $800 billion by 2025 [8].

Cloud platforms offer a great variety of virtual server options for executing all types of applications, including data-intensive applications such as graph analytics and key-value stores. To ensure security, isolation, and to hide the complexity of managing physical machines, cloud resources operate under virtualization and rent to cloud users as virtual machines with virtual CPUs. Moreover, in cloud environments, applications run inside virtual machines that are aggressively colocated on the same server for better CPU and memory utilization [6], [7]. State-of-the-art deployments, such as those at Google, colocate applications even on a single SMT core [9]. While enhancing security and improving utilization, such virtualized setups with aggressive colocation suffer from a particularly high latency of address translation on each TLB miss. As a result, one can see that many trends adversely affecting page walk latency emerge in the cloud environment, amplifying each other. Thus, reducing page walk latency in virtualized systems opens up opportunities for improving performance (e.g. increasing the throughput in cloud deployments).

1.3 Previous Research

Existing research proposals seeking to ameliorate the high cost of address translation tend to fall into one of two categories: *disruptive* changes to the virtual memory system and *incremental* improvements to existing designs.

Disruptive proposals advocate for a radical restructuring of the virtual memory subsystem. For instance, replacing the radix-tree-based PT with a hash-table-based PT can reduce the cost of a TLB miss to a single memory access in the best case [10], [11]. Problematically, such a replacement entails a complete overhaul of the entire PT data structure and TLB miss handler logic in both software and hardware – a daunting task involving extensive code changes in the kernel memory management module ① which has a lot of interdependences with other kernel modules [12] and ② implementation of which spans more than 68K lines of kernel code in Linux [13].

Another possibility is adding segment-based memory management to the existing page-based memory subsystem [14]. Regrettably, such an addition significantly complicates the virtual memory subsystem due to the need to develop, integrate and maintain a second translation mechanism in addition to the existing one. Recent work has also suggested allowing applications to install custom page walk handlers in the OS to enable application-specific address translation [15]. The downside of this approach is that the burden of accelerating address translation falls on application developers.

The challenge for future virtual memory systems is to enable low-overhead address translation for big-memory applications running in the cloud environment without disrupting existing system stacks.

1.4 Thesis Contributions

Aims of this thesis are:

- To characterize the performance of page walks under virtualization: quantify the effect of a high page walk latency on the overall application's performance, and
- To design non-disruptive mechanisms for reducing the overhead of page walks when running a data-intensive application under virtualization.

The rest of the chapter describes a motivating characterization study of page walks and provides a brief overview of proposals of two new mechanisms for reducing the overhead of page walks.

1.4.1 Characterization of Page Walks

1.4.1.1 Measuring Page Walk Latency

We quantify the effect of virtualization, application colocation, and growth in dataset size on page walk latency. We find that both virtualization and application colocation considerably increase page walk latency. Moreover, we find that increasing the dataset size also noticeably contributes to a growth in page walk latency.

In addition, we study what part of the nested page walk causes most of the total page walk latency. The results show that accesses to deeper levels of the PT attribute for a higher latency than accesses to the PT levels closer to the root of the PT. Moreover, we observe that with application colocation, accesses to the leaf PT nodes of the host PT on average take longer than accesses to the leaf PT nodes of the guest PT.

1.4.1.2 Quantifying Performance Cost of Page Walks

To show that long page walk latency can cause a significant reduction in overall performance, we quantify the overhead of page walks on real x86-64 hardware. In particular, we conduct a study to answer a question: how much performance does an application gain if page walks would be made shorter?

We design a methodology for varying the average page walk latency under virtualization on real hardware. Using this methodology, we measure the number of execution cycles on configurations with different average page walk latencies. By evaluating a representative set of data-intensive applications, we find that removing page walks can significantly reduce the overall execution time (up to 68%, 34% on average).

Moreover, the results show that reducing page walk latency by increasing the chances of page walk accesses hitting in LLC can reduce the overall execution time by up to 15% (12% on average).

1.4.1.3 Key Findings

Key observations of our characterization are:

- Under virtualization, page walk latency is significantly higher than in native standalone execution,
- Page walk latency increases with increasing dataset size,
- There are low- and high-latency memory accesses during a nested page walk; most of the latency comes from two memory accesses – accesses to the leaf nodes of the host and guest PTs,
- Without colocation, accesses to the leaf PT nodes of the guest and the host PTs take approximately the same time; however, with colocation, accesses to the host leaf PT nodes are significantly longer than to the guest leaf PT nodes, and
- Overall performance is sensitive to page walk latency. As a result, reducing page walk latency can improve performance.

1.4.2 Prefetching Page Table Nodes during a Page Walk

Based on the observation that the reason for the high page walk latency in a radix-tree-based PT is the serialization of memory accesses during a page walk, we propose *Address Translation with Prefetching (ASAP)* – a new paradigm for slashing the latency of page walks. ASAP can be applied to any ISA that employs radix-tree-based PTs and nested page walks (e.g. x86-64 or AArch64). ASAP breaks the dependency between consecutive PT nodes by fetching them ahead of the page walker via *direct indexing* of the PT. ASAP lowers page walk latency in a non-disruptive manner, retaining full compatibility with the PT and the existing address translation machinery (TLBs, page walk caches, etc.). ASAP is also non-speculative, which means it avoids costly prediction tables and does not introduce new security vulnerabilities.

To introduce direct indexing as a lightweight addition to the existing virtual memory subsystem, we exploit the observation that a process typically operates on just a few contiguous virtual memory ranges (further referred to as VMAs). One important example of such a range is the heap, which forms a large contiguous region in the virtual space of a process. Each allocated virtual page inside a VMA has a PT entry at the leaf level of the PT, reached through a chain of PT nodes, one node per PT level. Thus, there exists a one-to-one mapping between a virtual memory page and a corresponding PT node at each level of the PT. However, this correspondence exists only in the virtual space, but not in the physical space, due to the buddy allocator that scatters the virtual pages, including those of the PT, across the physical memory.

To enable direct indexing of PT nodes, there must be a function that maps a virtual page to the physical page containing the corresponding PT node. Our insight is that if the PT nodes for a given level of the PT in physical memory follow the same order as the virtual pages they map to, then a direct index into the PT array is possible using simple base-plus-offset computation.

To support direct indexing, the OS must induce the required ordering for the PT nodes in physical memory. This requires minimal and localized changes in the Linux kernel, in particular in the allocator for the PT (i.e., in `pmd_alloc()` and `pte_alloc()` functions), and absolutely no modifications to the actual radix-tree PT structure and the existing page walk routine.

ASAP non-disruptively extends the TLB miss-handling logic. For each VMA that supports direct indexing, ASAP requires a VMA descriptor consisting of architecturally-exposed *range registers* that contain the start and end addresses of the VMA, as well as the base physical addresses of the contiguous regions containing PT levels mapping the VMA. Our evaluation shows that tracking 2 VMAs is enough to provide direct indexing for 98% of the memory footprint for the studied big-memory benchmarks.

On each TLB miss, the virtual address of the memory operation is checked against the range registers. On a hit, the physical addresses of the target PT nodes are determined via a simple base-plus offset computation. Then, ASAP issues prefetched requests by calculated target addresses. PT prefetches generated by ASAP travel like normal memory traffic through the memory hierarchy. Prefetched cache blocks are placed into the L1-D cache to serve the upcoming requests from the page walker. We highlight that there is no modification required to the cache hierarchy or the page walker.

ASAP delivers a significant reduction in page walk latency, with an average improvement of 45% across the evaluated big-memory applications under virtualization on an x86-64-based system. This is an expected result as ASAP can overlap memory accesses during the page walk along both the host and guest dimensions of the nested walk, thus exposing the latency of only one memory access. By quantifying the fraction of cycles spent in page walks on a real processor, we can obtain a lower-bound of ASAP's performance improvement by multiplying this fraction with ASAP's reduction in page walk latency.

1.4.3 Improving Caching Efficiency of Page Table under Virtualization and Colocation

Further analyzing the result of the characterization of page walks, we perform a root cause analysis of the difference in average latencies of accesses to the leaf nodes of the guest and host PTs under virtualization and application colocation. We find that while guest PTEs corresponding to nearby virtual addresses reside in the same cache block, host PTEs corresponding to these guest virtual addresses are often scattered among multiple cache blocks.

To understand why PTEs may reside in different cache blocks, let's consider a simple case where applications are running natively.

A PT is indexed through virtual addresses, where the PTEs for two adjacent virtual pages A and $A+1$ sit in neighboring leaf nodes and within the same cache block. While the A and $A+1$ are adjacent in virtual address space, their virtual-to-physical mappings are determined by the memory allocator. If the memory allocator is allocating memory for a single application, adjacent virtual pages are likely to be mapped to adjacent physical pages, carrying over their spatial locality to physical address space. However, if the memory allocator is allocating memory for multiple applications, the allocations for virtual pages A and $A+1$ may be interleaved by memory allocation requests for co-running application(s). In this case, A and $A+1$ are unlikely to be mapped to adjacent physical pages and their spatial locality is lost in the physical address space. In the worst case, a set of pages that are contiguous in the virtual space may be allocated to physical pages that are entirely non-contiguous. This results in the application's memory being fragmented in the physical address space.

Under virtualization, when the memory allocator in a VM is stressed by colocated applications, each individual application's memory is fragmented in the guest OS's physical address space. The host OS deals with the VM like another process and treats the guest physical address space as the VM process' virtual memory. Problematically, the fragmentation in the guest physical memory carries over to the host virtual memory. Guest virtual pages A and $A+1$ which were mapped to non-adjacent guest physical pages will now be non-adjacent in host virtual address space. As a result, they will not occupy neighboring PTEs in the host PT, and will not reside in the same cache block. This increases the footprint of the host PT nodes corresponding to each application running inside the VM.

In response to these observations, we introduce PTEMagnet, a legacy-preserving software-only technique to reduce page walk latency in cloud environments by improving the locality of the host PTEs. Cache locality of host PTEs can be improved by limiting memory fragmentation in the guest OS. We show that prohibiting memory fragmentation within a small contiguous region greatly increases locality for host PTEs. PTEMagnet uses this observation and employs a custom guest OS memory allocator which prohibits fragmentation within small memory regions by fine-grained memory reservation. As a result, PTEMagnet improves the locality of the host PTEs and thus accelerates nested page walks. PTEMagnet can be implemented on any system that employs radix-tree PTs and nested page walks.

1.5 Published Work

During my PhD programme, I contributed to preparing four conference papers and one workshop submission that are listed below. Some of the contents of this thesis have appeared in the first two publications on the list.

- ***A.* Margaritov**, D. Ustiugov, E. Bugnion, B. Grot. ”Prefetched Address Translation”. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2019 [1]

- ***A.* Margaritov**, D. Ustiugov, S. Shahab, B. Grot. ”PTEMagnet: Fine-grained Physical Memory Reservation for Faster Page Walks in Public Clouds”. In *Proceedings the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2021 [2]

- **A. Margaritov**, D. Ustiugov, E. Bugnion, B. Grot. ”Virtual Address Translation via Learned Page Table Indexes”. In *Proceedings of the 2nd Workshop on Machine Learning for Systems at the Conference on Neural Information Processing Systems (MLSys).* 2018 [16]

- ***A.* Margaritov**, S. Gupta, R. Gonzalez-Alberquilla, B. Grot. ”Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores”. In

Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA). 2019 [17]. **Best Paper Award**

- A. Shahab, M. Zhu, **A. Margaritov**, B. Grot. ”Farewell My Shared LLC! A Case for Private Die-Stacked DRAM Caches for Servers”. In *Proceedings of the 51st International Symposium on Microarchitecture (MICRO). 2018 [18]*

1.6 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background on virtual memory subsystem mechanisms. Chapter 3 presents our evaluation of the overhead of page walks under virtualization and motivates the need for reducing page walk latency. The next two chapters describe our proposals for reducing page walk latency under virtualization. Chapter 4 presents ASAP, a new technique for shortening page walk latency by prefetching intermediate PT nodes. In Chapter 5, we study the effect of memory fragmentation stemming from application colocation on page walk latency, and present PTEMagnet, a simple software-only approach to prevent fragmentation of the PT through a fine-grained reservation-based allocation in the guest OS. Finally, in Chapter 6, we summarize our findings and provide potential future directions for research on accelerating page walks under virtualization.

Chapter 2

Background and Related Work

This chapter presents the background material necessary to understand the main contributions of this thesis.

2.1 Virtual Memory Basics

Virtual memory provides each process with the illusion of having access to a full address space, thus mitigating the reality of limited physical memory. The operating system (OS), using a combination of hardware and software, maps memory addresses used by a program (i.e., virtual addresses) into physical addresses in memory. The OS assigns memory on a page granularity. Translation of virtual-to-physical address happens on every memory access. In this thesis, we focus on systems having x86-64 or AArch64 architecture and using Linux as an OS. Such systems are common in cloud platform infrastructure. The remainder of this chapter describes Linux address translation mechanisms and memory allocation in detail.

2.1.1 Address Translation Mechanisms

Linux keeps the mapping of the virtual address space of a running process onto physical memory in a per-process structure, called a Page Table (PT). The PT maintains the mappings at a page granularity: for each page in the virtual space, there is a PT entry (PTE) that contains a virtual-to-physical address translation for all virtual addresses within a single page, as well as other important metadata including page access permissions. Before accessing any memory location in physical memory, the CPU must look up the address translation of the virtual address and check its validity with respect to access permissions.

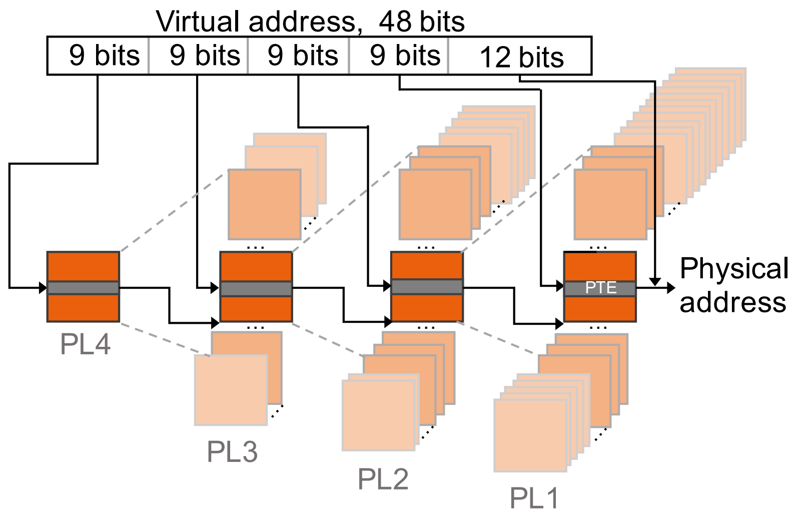


Figure 2.1: Linux x64-64/AArch64 page table organization as a four-level radix tree [1].

On x86-64 and AArch64 architectures, a PT is commonly organized as a multi-level radix tree, where the leaf level contains the actual PTEs. Upon a memory access, the CPU needs to perform a page walk, i.e., traverse the PT level-by-level from its root to the appropriate leaf with the corresponding translation. On both architectures, Linux uses a four-level PT (Linux supports a five-level PT on x86-64 [3]). In this thesis, we focus on a four-level PT. Figure 2.1 shows the organization of a four-level PT. Due to a large number of levels in a PT, a page walk, which is a sequence of serialized memory accesses to the nodes of a radix-tree PT, can be a long-latency operation especially if an application has a large memory footprint [1], [19], [20].

To reduce address translation overheads, modern CPUs deploy a wide range of machinery that includes both address-translation-specific and general-purpose caching. First, multi-level Translation-Lookaside Buffers (TLBs) cache address translations to recently accessed memory locations. Upon a TLB miss (if no TLBs have the required translation), a hardware-based page walker performs a PT look-up, bringing the corresponding translation from memory, potentially raising a page fault if the translation does not exist, and installing it in the TLB for future reuse. PTs are located in conventional physical memory so that page walks can benefit from the regular CPU cache hierarchy that accelerates accesses to the recently used parts of the PT. Besides the conventional caches, CPUs deploy Page-Walk Caches (PWCs) that hold recently accessed intermediate nodes of the radix-tree PTs, allowing the page walker to skip one or more PT levels in a page walk.

In a virtualized scenario, which is typical in a public cloud, guest and host OSs

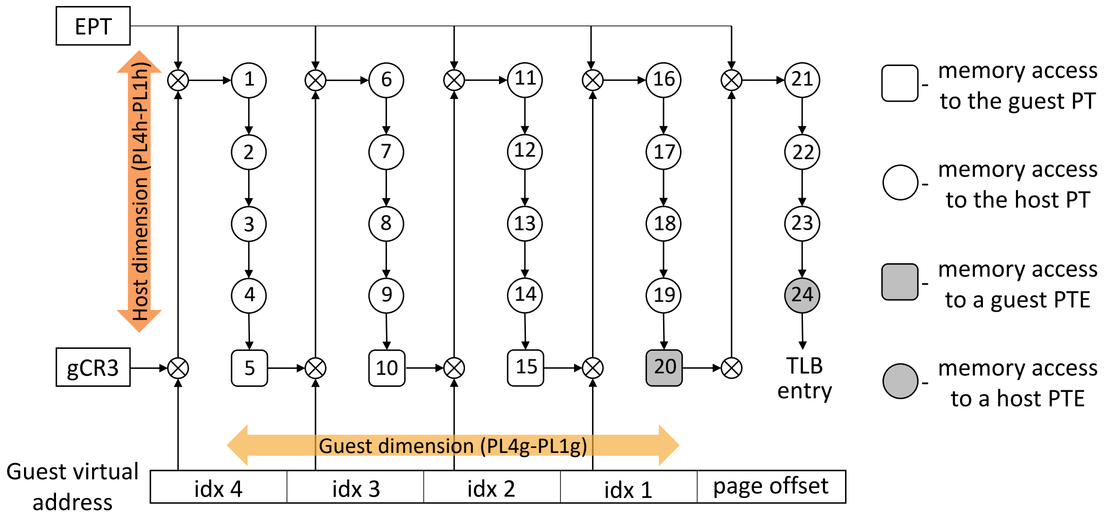

EPT
gCR3
Host dimension (PL4h-PL1h)
Guest dimension (PL4g-PL1g)
Guest virtual address
idx 4
idx 3
idx 2
idx 1
page offset
TLB entry
memory access to the guest PT
memory access to the host PT
memory access to a guest PTE
memory access to a host PTE

Figure 2.2: Nested page walk memory accesses. Accesses are enumerated according to their order in a nested (2D) page walk [1].

manage their PTs that together comprise so-called nested PTs [21]. Upon a TLB miss, a virtualized process has to perform a 2D (or *nested*) page walk. A 2D page walk involves a conventional 1D page walk in the guest PT where each access to the nodes of the guest PT triggers a complete 1D page walk in the host PT. The latter is required to find the location of the node in the next level of the guest PT in the host physical memory. Thus, to translate a guest virtual address to a guest physical address, in addition to the 4 memory accesses to the guest PT, the page walker needs to perform up to 4 memory accesses to the host PT for each access to the guest PT. Moreover, after getting a guest physical address, the page walker needs to perform one more 1D page walk in the host PT to figure out the location of the data in the host physical memory. Thus, to translate a guest virtual address to a guest physical address, on top of the 4 memory accesses to the guest PT, the page walker needs to perform up to 4 memory accesses to the host PT every time it accesses the guest PT (or up to 20 memory accesses to the host PT in total). To point to a particular memory access during a nested page walk, we assign a unique serial number from 1 to 24 to all memory accesses of a nested page walk. Figure 2.2 represents the mechanism of calculating and accessing PT nodes during a nested page walk. Figure 2.2 also represents the mapping between the assigned numbers and the nested page walk memory accesses. For example, an access to a guest PTE is marked with the number 20.

2.1.2 Memory Allocation Granularity: Page Sizes

The four-level PT employed by x86-64 is designed to provide a translation for a 4KB (small) page. However, the hierarchical structure of the PT radix tree naturally allows for a technique to reduce the overheads of address translation – shortening the page walk by storing one translation for a group of pages in intermediate PT nodes. This approach allows replacing 512 contiguous small pages with a large page of the equivalent size. For example, in Linux/x86-64, a large page of 2MB is managed by a single entry in the PL2 level of the PT, replacing the corresponding 512 independent PT entries in the PL1 level. Large pages are one of the least intrusive, with respect to the PT radix tree, innovations in memory management, requiring modest hardware and software extensions.

The x86-64 architecture supports different page sizes: 4KB (small), 2MB and 1GB (large). The AArch64 architecture is more flexible: it supports the same page sizes as the x86-64 architecture along with other page sizes. In this thesis, we configure AArch64-based systems to use only the page sizes available on the x86-64 architecture (i.e., set the base page size to 4KB).

2.1.2.1 Problems of Large Pages

Unfortunately, even the least intrusive changes in the virtual memory mechanisms carry numerous implications on the overall system behavior and introduce performance pathologies. On x86-64, the introduction of 2MB and 1GB large pages revealed an ensemble of unexpected problems. Araujo et al., for example, show that the use of large pages leads to memory fragmentation in multi-tenant cloud environments [22]. Under high memory fragmentation, Linux often has to synchronously compact the memory before a memory chunk of the necessary size can be allocated, introducing high average and tail latencies. Kwon et al. showcase a problem in unfair large pages distribution among multiple applications sharing a single server, as well as point to increasing memory footprint due to the internal fragmentation [23]. For instance, the authors show that `redis` increases its memory footprint by 50% when using large pages and may start swapping even with a carefully provisioned physical memory.

While being a simple and natural idea, the systems community has struggled with the wide adoption of large pages. The root of the problem is the lack of memory management flexibility (e.g., copy-on-write in OS, deduplication, and ballooning optimizations in hypervisor [24], [25]) with large pages as compared to fine-grain paging.

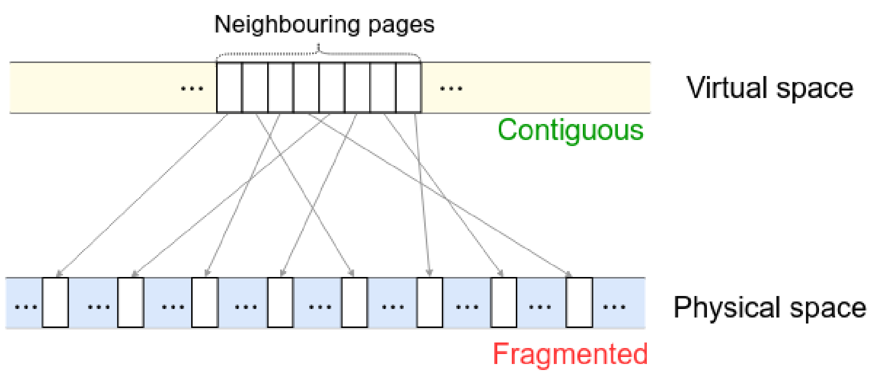


Figure 2.3: Difference in contiguity in virtual and physical address spaces [2].

Large pages are also not used for code due to security concerns. Indeed, large pages increase the chances of breaking address space layout randomization due to much lower randomness among physical addresses for code. With large pages, a functional dependency between physical addresses is higher as the number of large pages needed to allocate code is much lower than the number of small pages.

Due to the combination of issues above, large pages are often disabled in production systems [26]. As a result, in this thesis, we evaluate systems using only 4KB small pages.

2.1.3 Memory Allocation Mechanism

In Linux, a process can request memory from Linux, by executing `mmap()` or `brk()` system calls. After executing these calls, the process immediately (eagerly) receives the requested virtual memory region. In contrast to the allocation of virtual space, physical memory allocation is performed lazily: the OS assigns physical memory to a process on-demand and on a page-by-page basis. The OS allocates memory for the process with a page of physical memory upon the first access that triggers a page fault and inserts the corresponding virtual-to-physical mapping, that is a PTE, into the PT.

2.1.4 Memory Fragmentation

Eager allocation of virtual address space in conjunction with lazy physical address space allocation can create a drastic difference in the spatial locality of the two address spaces. In Linux, both `mmap()` and `brk()` system calls are implemented to return contiguous virtual memory regions (e.g., two consecutive `mmap` calls from one application return contiguous virtual addresses) whereas physical memory pages are allocated ad

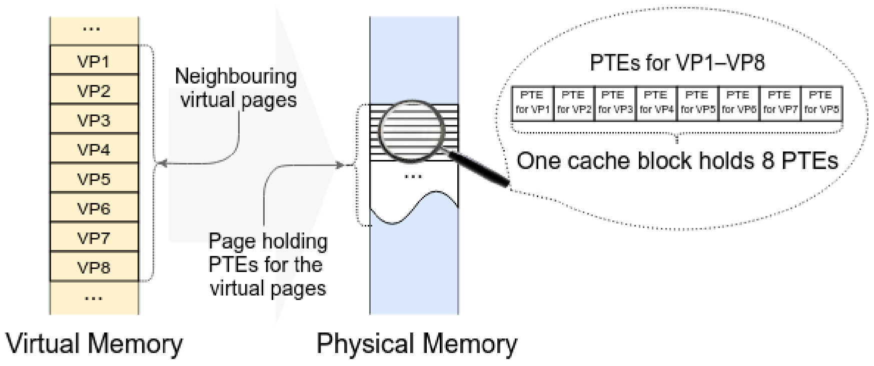


Figure 2.4: Packing of PTEs of neighbouring virtual pages in one cache block [2].

hoc, as the Linux buddy allocator is optimized for fast physical memory allocation instead of contiguity. In a multi-tenant system, due to the fact that page faults from different applications are coming asynchronously, the buddy allocator fragments the physical memory space. As a result, addresses that are contiguous in the virtual address space often correspond to pages arbitrarily placed in physical memory [1], [27], [28]. Thus, under aggressive colocation, which is typical in the public cloud, buddy allocator interspersedly allocates physical memory to different applications, destroying the contiguity present in the virtual space. Figure 2.3 demonstrates the difference in contiguity between virtual and physical address spaces.

2.1.5 Spatial Locality in the Page Table

Prior work has demonstrated that abundant spatial locality exists in access patterns of many cloud applications [29]. Spatial locality means that nearby virtual addresses are likely to be accessed together. In native execution, PT accesses also inherit the spatial locality of accesses to data. Indeed, being indexed by virtual addresses, PTEs corresponding to adjacent pages are tightly packed in page-sized chunks of memory. As a result, TLB misses to adjacent pages in the virtual space result in page walks that traverse the PT radix-tree to adjacent PTEs. Dense packing of such PTEs amplifies the efficiency of CPU caches that manage memory at the granularity of cache blocks: up to eight adjacent 8-byte PTEs (which are likely to be accessed together due to applications' spatial locality) may reside in a single cache block. Figure 2.4 represents the layout of the PT, highlighting the tight packing of PTEs of neighbouring pages in a cache block.

Figure 2.5 shows trajectories of consecutive page walks for ⓐ contiguous pages and ⓑ non-contiguous pages. One can see that page walks for contiguous virtual

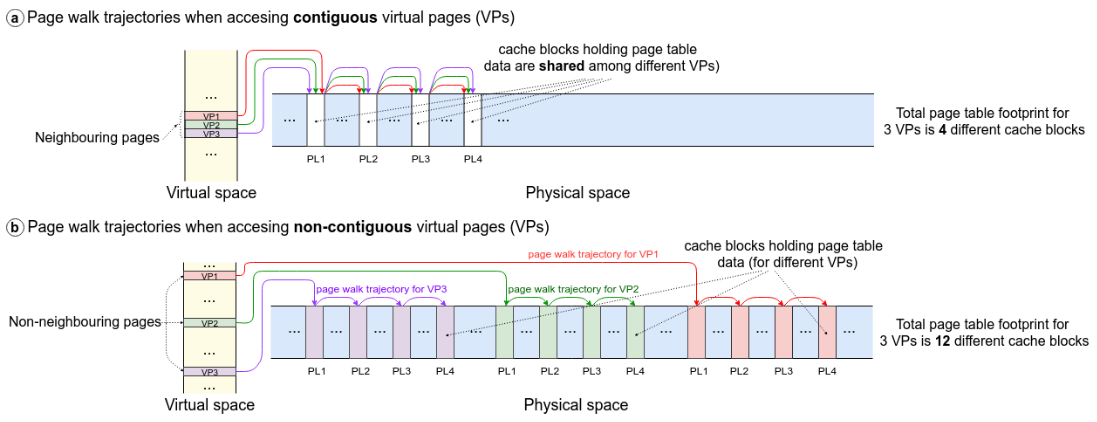

(a) Page walk trajectories when accesing **contiguous** virtual pages (VPs)
cache blocks holding page table data are **shared** among different VPs)
Neighbouring pages
VP1
VP2
VP3
PL1
PL2
PL3
PL4
Total page table footprint for 3 VPs is **4** different cache blocks
Virtual space
Physical space
(b) Page walk trajectories when accesing **non-contiguous** virtual pages (VPs)
page walk trajectory for VP1
cache blocks holding page table data (for different VPs)
page walk trajectory for VP3
page walk trajectory for VP2
Non-neighbouring pages
Total page table footprint for 3 VPs is **12** different cache blocks
Virtual space
Physical space

Figure 2.5: Page walk trajectories in physical memory when accessing *a* – non-contiguous and *b* – contiguous virtual pages [2].

pages experience spatial locality as page walks for all pages access the same cache blocks. In contrast, page walks for non-contiguous pages go through different cache blocks, which necessarily increases the cache footprint of the PT and increases the page walk latency. As a result, the locality in access patterns and contiguity in the virtual space can make page walks faster. Thus, in the absence of virtualization, by creating contiguity in the virtual space, the memory allocation mechanism naturally helps to make address translation faster. Note that fragmentation in the physical space has no effect on page walk latency because the locality in the PT stems *only* from contiguity in the virtual space and not physical.

2.2 Previous Research on Address Translation

Prior attempts at accelerating address translation are either disruptive – requiring a radical re-engineering of the virtual memory subsystem – or incremental to the existing mechanisms. The following subsections discuss both types of proposals in detail.

2.2.1 Disruptive Proposals

With rapidly increasing application dataset sizes and the associated growth in translation overheads [30], some of the recent work argues for a complete replacement of the PT radix tree and the address translation mechanisms that are designed around it. One set of proposals has suggested using large contiguous segments of memory instead of fine-grained paging for big-data server applications [14], [31], [32]. These proposals are based on observations that server applications tend to allocate all of their memory at start-up and make little use of page-based virtual memory mechanisms, such as swapping and copy-on-write [14], [31], [32]. Hence, the authors suggested using segments of contiguous memory for these applications instead of pages. Managing memory in segments of an arbitrary size allows to considerably reduce translation overheads that arise due to the increasing number of fixed-size pages required to cover growing datasets.

While tempting from a performance perspective, there are several major issues with the segment-based approach that prevent its adoption. First, segment-based memory relies on specific characteristics of a single class of applications, whereas an OS has to be general-purpose. Hence, the OS needs to efficiently support both the conventional page-based approach and the segment-based one. Supporting two separate translation mechanisms makes the memory subsystem heterogeneous and more complicated to manage. Second, not all big-memory applications can operate on a small number of segments, as noted by [33], while supporting a large number of segments is impractical in hardware, as noted in the original paper [31].

Another challenge with segment-based memory management is that, in practice, finding large contiguous physical memory regions can be challenging in the presence of memory cell failures. Such failures are exposed to the OS, which monitors the health of the available physical memory, detects the faulty cells, and *retires* the affected physical memory at granularity of small pages [34], [35]. While a recent study at Facebook already reports increasing memory error rates due to DRAM technology scaling to smaller feature sizes [36], emerging memory technologies, such as 3D XPoint, are likely to have a higher incidence of memory cell failures due to lower endurance as compared to DRAM [37], [38]. Hard memory faults dramatically complicate memory management not only with segments but also with large pages; e.g., by introducing an extra level of abstraction [39] and new software and hardware machinery (e.g., per-segment Bloom filters for pages with hard faults [32]).

To reduce PT access latency, prior work proposes replacing the PT radix tree with lookup-latency optimized data structures, such as hash tables [10], [11], [40] and hashed inverted PTs as in IBM PowerPC [41]. While promising, such designs require a complete overhaul of address translation and memory allocation mechanisms in both software and hardware. Making a lot of changes in both software and hardware components of a system at the same time is a complex task. Therefore, adopting any of these designs for conventional usage is a daunting prospect. In addition, traditionalhash-table-based designs may suffer from performance pathologies including long-chain traversals due to hash collisions [10], [20]. SPARC architecture handles TLB misses in software while accelerating TLB miss handling by introducing a software-managed direct-mapped cache of translations, called TSB [42]. However, prior work shows that larger TSB entries exhibit poorer cache locality making TSB less efficient than the conventional PT radix tree [20].

Other prior work investigated application-specific address translation, allowing the developers to choose address translation methods appropriate for their applications [15], e.g., using segment- [14] or a hashtable-based PT organization [10]. However, these approaches expose the complexity of virtual memory programming to the application developers or, if implemented at the system level, lead to an increase in OS memory management complexity. Both greatly impede the adoption of these ideas in the wild.

2.2.2 Incremental Approaches

To avoid the disruption to existing system stacks, some researchers have proposed microarchitectural and OS-based techniques to accelerate address translation while retaining compatibility with the conventional PT and radix tree mechanisms. One such group of mechanisms seeks to leverage existing contiguity in both virtual and physical space to coalesce multiple PTEs into a single TLB entry so as to increase TLB reach [43], [44]. Indeed, thanks to the ease of their integration into existing system stacks, the industry has recently adopted this idea [45].

Problematically, the effectiveness of coalescing-based approaches is fundamentally constrained by the size of a TLB structure. Another limitation is that coalescing relies on having contiguity in the physical memory space of an application; however, ensuring such contiguity is not an objective of existing Linux memory allocators [33]. As a result, generating physical memory allocation contiguity is not the purpose of the existing mechanisms, such as Linux buddy allocator, but rather a side-effect. As a result, on a default OS, application contiguity characteristics can vary greatly across different runs of the same application, especially under application colocation, making solutions that rely on PTE coalescing unreliable from a performance perspective. (One of our contributions is an analysis of the sources of the variability in application contiguity characteristics which is discussed in Chapter 5.)

Another set of techniques focus on improving TLB efficiency in the presence of multiple page sizes. A straight-forward TLB design that statically partitions capacity across a set of supported page sizes will suffer from poor utilization when one-page size dominates. Moreover, because the size of the page that belongs to a particular memory access is unknown before a TLB look-up, all of the TLB structures need to be checked, increasing TLB hit latency and energy consumption. While recent works have attacked these problems [46]–[48], their effectiveness is fundamentally limited by the capacity of the TLB structure. To combat these issues, Misel-Myrto et al. propose page size prediction to reduce TLB look-up latency and energy consumption [46]. Other researchers explored using a unified indexing function for all the page sizes while keeping a single hardware structure for all the translations [47], [48]. While effective, these techniques are all fundamentally limited by the capacity of the TLB structure. The advent of variable page sizes significantly increased the complexity of the TLB machinery, often introducing multiple hardware structures and concurrent indexes that increase the look-up time and waste energy.

Thanks to the backward compatibility of these approaches, the industry has already started their adoption [45]. However, none of these approaches (to the best of our knowledge) offer much-desired scalability: while datasets continue to increase, the efficiency of these techniques would inevitably decrease due to limited capacity of the physical TLB structures as well as coalescing opportunities exposed by the software.

2.2.3 Summary

Improving the performance of virtual memory has been a focus of much recent work. While some prior work seeks for radical re-design of conventional PTs and replacing them with a scalable segment-based or hash-table approach, such solutions are not general enough and introduce additional complexity into the already sophisticated virtual memory subsystem. Other researchers take a pragmatic side focusing on incremental and evolutionary techniques. However, none of these techniques offer the much-desired scalability required due to the unrelenting growth in application dataset sizes.

Chapter 3

Characterizing Page Walks

3.1 Introduction

In this chapter, we characterize page walk latency and its effect on performance. We start the characterization by evaluating how the dataset growth, virtualization, and colocation affect the average page walk latency. Next, we analyze the sources of high page walk latency under virtualization. We measure the latency of each memory access during a nested page walk with and without colocation. To understand what increases page walk's latency under colocation, we study which accesses to the main memory during a nested page walk cause most of the latency. Lastly, we study how page walk latency affects the overall performance under virtualization. To measure the effect of page walk latency on performance, we design a methodology for varying the average page walk latency of an application executed on real hardware under virtualization. By measuring execution time with different page walk latencies, we identify a performance drop attributed to page walks. Moreover, we show that shortening nested page walks by bringing PT nodes close to a CPU in the memory hierarchy can considerably lower execution time.

3.2 Measuring Page Walk Latency

In this study, we measure the average page walk latency. We study four configurations: native and virtualized executions both with and without application colocation. As modern hardware does not provide a performance counter reporting the average page walk latency, we conduct a simulation study. We simulate the memory hierarchy of an Intel-Broadwell-like processor using DinamoRIO [49] (details on the simulation

Name	Description
mcf	SPEC'06 benchmark (ref input)
canneal	PARSEC 3.0 benchmark (native input set)
bfs	Breadth-first search, 60GB dataset (scaled from Twitter)
pagerank	PageRank, 60GB dataset (scaled from Twitter)
mc{80,400}	Memcached, in memory key-value cache, 80GB and 400GB datasets
redis	In-memory key-value store (50GB YCSB dataset)

Table 3.1: Workloads used in the characterization.

Parameter	Value/Description
Processor	Dual socket Intel(R) Xeon(R) E5-2630v4 (BDW) 2.40GHz 20 cores/socket, SMT
Memory size	160GB/socket (768GB/socket for `mc400`)
Hypervisor	QEMU 2.0.0, 128GB RAM guest
Guest/host operating system	Ubuntu 14.04.5, Linux kernel v4.4

Table 3.2: Parameters of the hardware platform used for characterization studies.

5× larger dataset	SMT colocation	Virtualization	Virtualization + SMT colocation	Virtualization + SMT colocation + 5× larger dataset
1.2×	2.7×	5.3×	12.0×	15.8×

Table 3.3: Increase in `mc` page walk latency under various scenarios. The data is normalized to native execution in isolation with a 80GB dataset.

methodology are described in Section 4.3). We measure the average page walk latency for applications listed in Table 3.1. Table 3.2 lists the parameters of the simulated system and processor.

Figure 3.1 demonstrates that, despite all existing hardware support, page walk latency may climb to hundreds of cycles. Table 3.3 summarizes the trend by focusing on the `mc` workload. Various factors can affect the page walk latency, including the size of the dataset, interference caused by a co-running application, and virtualization. If running on a core in isolation and in native mode, `mc` has average page walk latency of 44 cycles. Increasing the dataset size of `mc` from 80GB to 400GB causes the average page walk latency of `mc` to climb by 1.2×. Despite the fact that this value

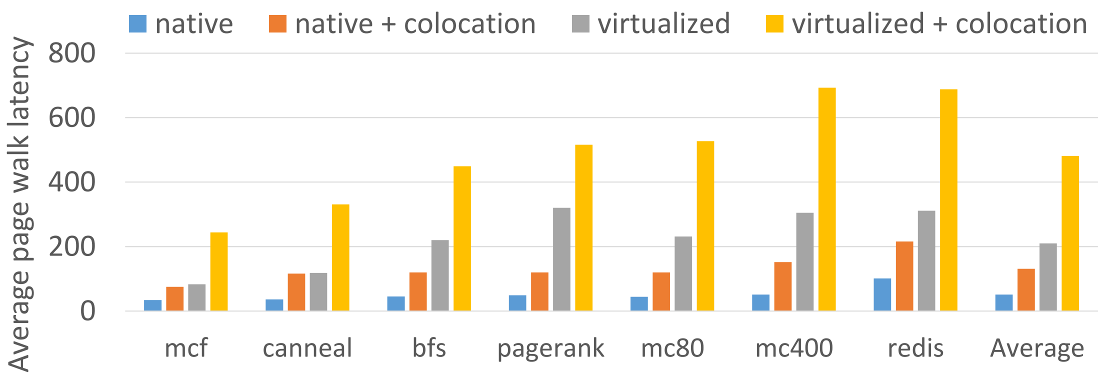


Figure 3.1: Average page walk latency in various scenarios [1].

might look modest, if projected to multiple-terabyte datasets, the dataset size will impact page walk latency much more. Colocation with another application on the same core increases the page walk latency by 2.7× (to 120 cycles). Virtualization (without colocation) increases the page walk latency of `mc` by 5.3×. Virtualization with colocation propels the page walk latency of `mc` to 527 cycles that comprises a 12× increase. Finally, virtualization, colocation, and the 5× dataset size growth increase the page walk latency to 697 cycles, a whopping 15.8× increase.

3.3 Analyzing Sources of High Page Walk Latency

In this section, we analyze the sources of high page walk latency in virtualized systems with and without colocation. To that end, we study which memory accesses during a page walk incurs the longest latency.

Under virtualization, a nested page walk consists of up to 24 memory accesses. Figure 2.2 represents all memory accesses of a nested page walk and assigns a number to each memory access. During a simulation, for each memory access during a nested page walk, we measure its average latency. We perform this study for `mcf` running under virtualization alone and in colocation with a data-intensive benchmark.

Figure 3.2 represents the results of the study. The legend lists the numbers of individual memory accesses during a nested page walk. To simplify the result presentation, we show the latency of the first 19 memory accesses as one aggregated value. One can see that for standalone execution, memory accesses with serial numbers 20 and 24, which correspond to accesses to PTEs of the guest PT and the host PT respectively, account for the largest part of the total page walk latency if running alone. High latency of accesses to PTEs of both PTs can be explained by the fact that the PT level con-

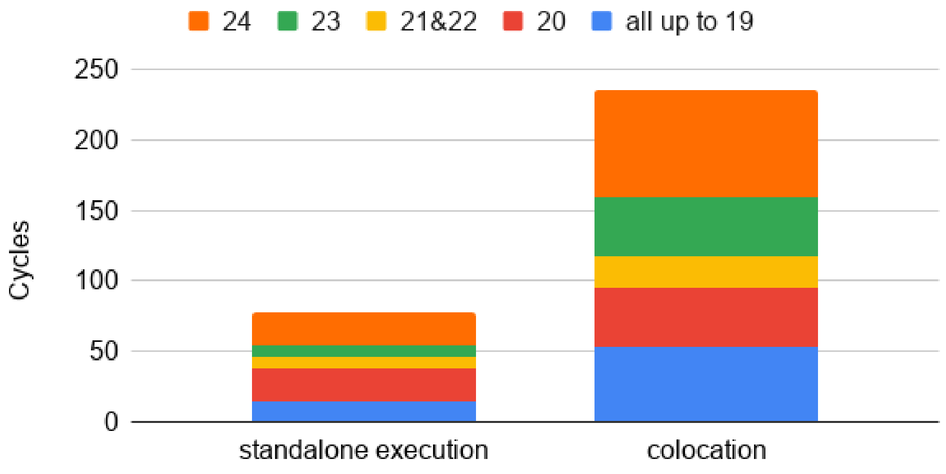


Figure 3.2: `mcf`'s average page walk latency breakdown by page walk memory accesses. The legend lists the serial numbers of individual memory accesses according to their order in a nested page walk (see Figure 2.2 for more details). Accesses with numbers 20 and 24 correspond to accesses to guest and host PTEs, respectively.

taining PTEs has the largest footprint among all PT levels. As a result, accesses to PTEs are more likely to result in a miss in the cache hierarchy and trigger a read from the main memory, increasing the average latency of the accesses to PTEs. One can see that the average latencies of accesses to guest and host PTEs are equal if running alone. The parity can be explained by the fact that each page walk accesses both cache lines holding guest and host PTEs. Thus, on average, cache lines holding guest and host PTEs have the same reuse distance, and, as a result, the same probability to be found in the cache hierarchy.

In theory, with application colocation, the parity between average latencies of accesses to guest and host PTEs should remain unchanged as the co-runner affects the reuse distance of all cache blocks similarly. However, the results show that with application colocation, `mcf`'s average latency of the access to the guest PTE is not equal to the average latency of the access to the host PTE. Indeed, Figure 3.3 shows the comparison of these latencies among both configurations. One can see that in colocation, the average latency of the access to the host PTE (serial number 24) is almost twice higher than the latency of the access to the guest PTE (serial number 20).

As a result of this study, we conclude that in colocation, there is an unknown effect that causes the discrepancy between the average latencies of accesses to the guest and host PTEs. We study this effect in detail in Chapter 5 where we discover its root cause

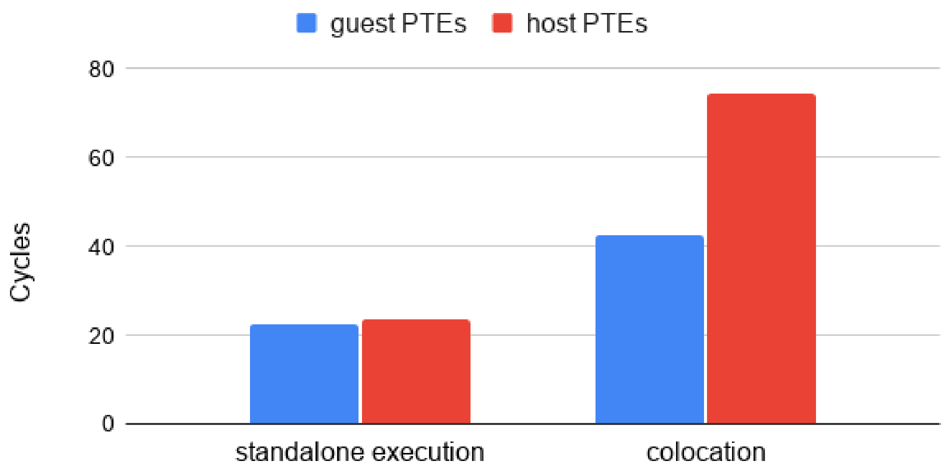


Figure 3.3: Comparison of average latencies of memory accesses to the guest and host PTEs during a nested page walk.

and propose a method to remove the discrepancy, reducing the average page walk latency.

3.4 Measuring How Page Walks Affect Performance

In this section, we quantify what part of the overall performance is wasted due to the high page walk latency under virtualization. To that end, we design a methodology that allows quantifying the effect of reduction in the average page walk latency on an application running on real hardware. By applying the methodology, we estimate how changing the average page walk latency affects the application's performance. Note that while our methodology helps to quantify the performance change, it won't be efficient if applied to production systems as it involves locking some hardware resources, namely LLC capacity.

3.4.1 Methodology for Varying Page Walk Latency

We design a methodology allowing to vary the average page walk latency experienced by an application running under virtualization. We change the base page walk latency by two approaches.

Elimination of page walks. The first approach is to remove TLB misses by forcing an application to allocate only large pages using *libhugetlbfs* [50]. Using large pages allows increasing the TLB reach – the size of a dataset for which address translation can take place without performing page walks. In this study, we use an Intel Broadwell processor. As the capacity of the second-level TLB is considerably (more than 24×) larger than that of the first-level TLB on the evaluated processor, we approximate the TLB reach as a size of the dataset whose translations can be accommodated in full by the second-level TLB. The second-level TLB of the evaluated processor has 1536 entries where each entry can store a translation for one page. With 4KB pages, the TLB reach is $4KB * 1536 = 6MB$. In contrast, with large pages, the TLB reach is $2MB * 1536 = 3GB$. As a result, with 4KB pages, an application with an irregular memory access pattern experiences a significant number of TLB misses if an application's dataset size is considerably larger than 6MB. In contrast, with large pages, an application can run with a dataset of up to 3GB in size while experiencing a considerably lower (≈100×) number of TLB misses than with 4KB pages. Consecutively, one can say that forcing using large pages effectively eliminates TLB misses of an application with a dataset size of up to 3GB. The absence of TLB misses means no page walks. Elimination of a page walk can be considered keeping a page walk but zeroing out its latency. Thus, one can say that using large pages effectively turns all page walks observed with 4KB pages to zero-latency page walks.

Neglecting the difference in overheads of memory allocation between 4KB and large pages, execution with large pages corresponds to execution with 4KB but with zeroed out average page walk latency.

To conclude, by forcing using large pages on an Intel Broadwell processor with an application using a dataset smaller than 3GB, one can achieve the effect of zeroing out the average page walk latency.

Improving the page walk's hit rate to caches. The second approach to lower the average page walk latency is to force PT nodes to remain on-chip. If PT nodes stay on-chip, a page walk never reads PT nodes from the main memory, which is expected to reduce the average page walk's latency. We force PT nodes to stay on-chip by guaranteeing that PT nodes are in the last level cache (LLC). We place PT nodes in LLC by constantly accessing them in a loop by a custom kernel module running in parallel with the application on the same processor. To increase the chances of a PT node staying on-chip, we guarantee no evictions of PT cache blocks from LLC using LLC partitioning. We partition LLC between the application and the kernel module using Intel's Cache Allocation Technology [51]. We create two LLC partitions and assign the first partition to an application and the second partition to the PT-fetching kernel module. We allocate LLC partitions in a way that the kernel module's LLC

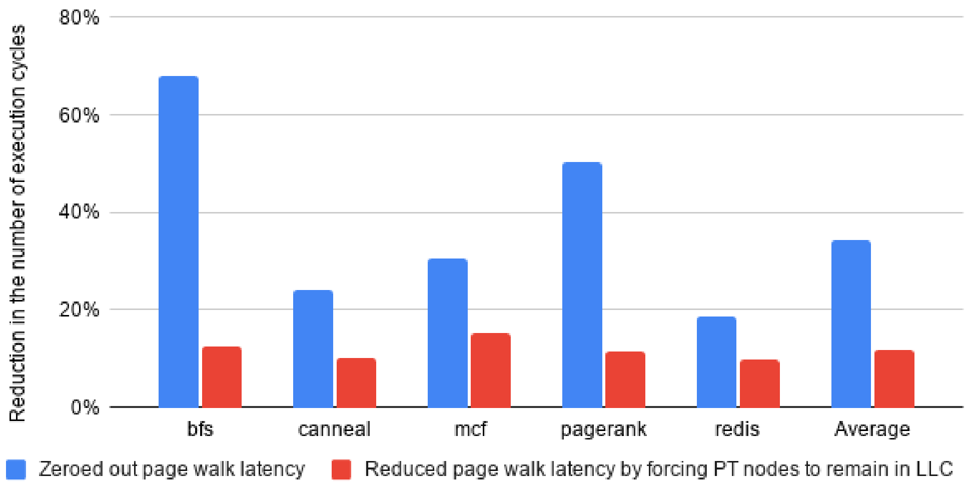


Figure 3.4: Reduction in the number of execution cycles achieved by ① zeroing out the average page walk latency and ② reducing the average page walk latency by forcing PT nodes to remain in LLC.

partition is capable to accommodate the whole application's PT while the rest of the LLC capacity goes to the application. To conclude, we can reduce the average page walk latency by forcing PT nodes to remain in LLC by a combination of ① fetching PT nodes with a custom kernel module and ② partitioning LLC to reduce the chances of PT nodes being evicted from LLC.

3.4.2 Evaluation of Performance with Various Page Walk Latencies

Firstly, we study how eliminating page walks affects the application's performance under virtualization. We use the number of execution cycles reported by a corresponding performance counter as a metric of performance. We study two configurations: ① 4KB pages, and ② with large pages forced by the *libhugetlbfs*. The configuration with 4KB pages has the base average page walk latency while the configuration with large pages has no page walks (that is equivalent to zeroed out latency).

Secondly, we analyze how reducing page walk latency by keeping PT nodes on-chip affects the application's performance under virtualization. In this part of the study, we use 4KB pages for all configurations. We study two configurations: ① normal execution and ② execution with the PT nodes forced to be on-chip. The first configuration features the base average page walk latency. In comparison to the first configuration, the configuration with PT nodes in LLC is expected to have a lower average page walk latency. On both configurations, we limit LLC capacity available to an application in a way that the remaining part of the LLC can accommodate the whole application's PT.

We evaluate all applications listed in Table 3.1 except `mc`, which is unaffected by *libhugetlbfs*. We run all applications with datasets of 3GB or smaller. We report the number of execution cycles saved thanks to a reduction in the average page walk latency. We normalize the results to the number of execution cycles on the configuration with normal execution (and 4KB pages). Figure 3.4 shows the results of the study. Zeroing out the average page walk latency reduces the number of execution cycles by 18-68% (34% on average) compared to a configuration with the base page walk latency. The largest reduction is observed on graph applications – 68% on `bfs` and 50% on `pagerank` that corresponds to whopping speedups of 3.1× and 2.0×, respectively.

Reducing the average page walk latency by forcing PT to remain on-chip results in smaller savings in the number of execution cycles than zeroing out the average page walk latency. Keeping PT nodes in LLC results in saving 10-15% (12% on average) of execution cycles in comparison to execution without the guarantee that PT nodes are always in LLC. Forcing PT nodes to remain on-chip delivers the largest saving in the number of execution cycles of 15% on `mcf` whereas `bfs` and `redis` demonstrate the second-largest reduction in in the number of execution cycles of 12%.

3.5 Conclusion

In this chapter, we characterize page walks. We measure how various factors – ① the dataset growth, ② virtualization, and ③ application colocation – affect the average page walk latency. The results show that virtualization significantly increases the average page walk latency, especially when an application has a large dataset and its memory accesses are irregular. Moreover, we find that under virtualization, a high page walk latency results in a significant performance overhead of 34% on average. Thus, we conclude that reducing page walk latency under virtualization can boost performance considerably.

To find opportunities for page walk latency reduction under virtualization, we analyze the average latencies of accesses to each PT node during a nested page walk. The results of our analysis show that accesses to PTEs have higher latency than accesses to other PT nodes. Moreover, under virtualization and colocation, we discover a discrepancy between average latencies of accesses to the PTEs of the guest and host PTs: accesses to the host PTEs on average take much longer than accesses to the guest

PTEs. To the best of our knowledge, we are the first to report such a discrepancy. We show that under virtualization, page walks account for a significant performance overhead: eliminating page walks would result in 34% performance improvement on average (68% max). Finally, we find that bringing PT nodes closer to a CPU can result in a considerable performance improvement (12% on average if PT nodes are kept in LLC).

Our findings discussed in the previous paragraph show opportunities to reduce page walk latency under virtualization and improve performance. In the following two chapters, we present two techniques for reducing page walk latency based on our characterization findings.

Chapter 4

Address Translation with Prefetching

4.1 Introduction

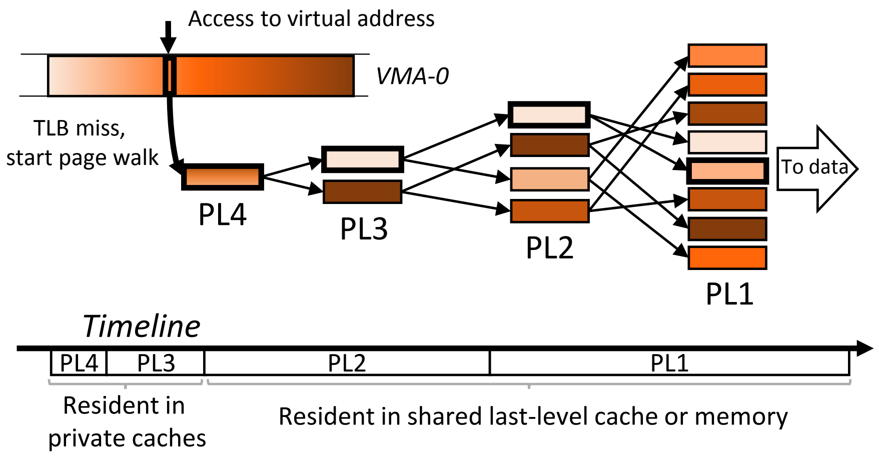
The challenge for future virtual memory systems is to enable high-performance address translation for terabyte-scale datasets without disrupting existing system stacks. As a step in that direction, in this chapter, we introduce *Address Translation with Prefetching (ASAP)* – a new paradigm for reducing the latency of the iterative pointer chase inherent in PT walks through a direct access to a given level of the PT. With ASAP, under virtualization, a TLB miss typically exposes the latency of just two accesses to the memory hierarchy regardless of the depth of the PT.

To introduce ASAP as a minimally-invasive addition to the system stack, we exploit the observation that applications tend to have their virtual memory footprint distributed among only a handful of contiguous virtual address ranges, referred to as Virtual Memory Areas (VMAs). Each allocated virtual page has a PTE, sitting at the leaf level of the PT, and a set of intermediate PT nodes that form a pointer chain from the root of the PT to that PTE. Due to lazy memory allocation, PTEs associated with a given VMA tend to be scattered in machine memory, with no correlation between their physical addresses and that of the associated virtual pages. Our insight is that if the PTEs were to reside in contiguous physical memory and follow the same relative order as the virtual pages that map to them, then there would exist a direct mapping between the virtual page numbers in a VMA and the physical addresses of their corresponding PTEs. Given such a mapping, finding a PTE can be done through simple base-plus-offset addressing into the PTE array, avoiding the need to access preceding levels to find the PTE location. Similar logic applies to intermediate levels of the radix tree, which also can be directly indexed using base-plus-offset addressing provided the entries are in contiguous memory and in sorted order with respect to virtual addresses they map. Sorted order means that if a virtual page number X comes before virtual page number Y, then the radix tree entry for X resides at a physical address less than that of the radix tree entry for Y.

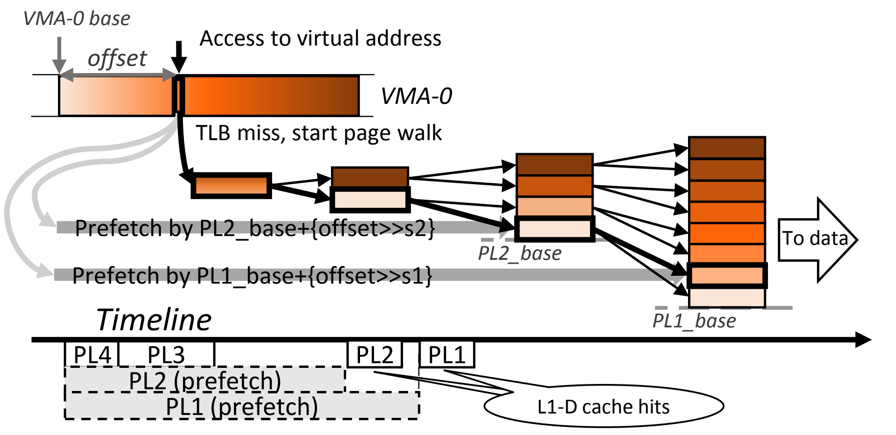
Realizing this idea requires few changes to an existing system stack. Indeed, VMAs are already explicitly maintained by modern operating systems. The key missing OS functionality is ensuring that the level(s) of the radix tree that are prefetch targets are allocated in contiguous physical memory and the entries are in sorted order. One way to achieve this is by directing the OS memory allocator to reserve, at VMA creation time, a contiguous region of physical memory for the PT entries. The required memory amounts to under 0.2% from the total memory size used by an application. On the hardware side, a set of architecturally-exposed *range registers* are needed to encode the boundaries of prefetchable VMAs. Any virtual address that misses in the TLB is checked against the range registers; on a hit, the target physical address is computed using a base-plus-offset arithmetic, and a ASAP prefetch is issued for the computed address. Multiple prefetches, to different levels of the radix tree, can be launched in parallel.

Crucially, regardless of whether a prefetch is generated or not, the PT radix tree is walked as usual. The full traversal of the radix tree guarantees that only correct prefetched entries are consumed. Thus, ASAP does not enable a CPU to speculative on the value of the address being translated as no execution will be performed until prefetched entries are validated by the existing page walk mechanism. As a result, such ASAP design minimizes the risk of introducing a new speculation-based security vulnerability into the existing architecture.

While ASAP falls short of completely eliminating page walk latencies, since under virtualization it exposes typically two accesses to the memory hierarchy (other accesses can be overlapped, hence hiding their latency), it has one major advantage: its minimally-invasive nature with respect to the existing address translation machinery. Thus, with ASAP, TLB misses trigger normal PT walks, which are accelerated thanks to ASAP prefetches. Meanwhile, TLBs, hardware PT walkers, the PT itself and the nested address translation mechanism used for virtualization require absolutely no modifications in the presence of ASAP. Thus, ASAP can be seamlessly and gradually introduced into existing systems with no disruption to either the hardware or software ecosystem.



(a) Conventional page walk and its timeline.


VMA-0 base
offset
Access to virtual address
VMA-0
TLB miss, start page walk
Prefetch by PL2_base+{offset>>s2}
PL2_base
Prefetch by PL1_base+{offset>>s1}
To data
PL1_base
Timeline
PL4
PL3
PL2
PL1
PL2 (prefetch)
PL1 (prefetch)
L1-D cache hits

(b) ASAP: Page walk accelerated with prefetching PT nodes from the bottom PT levels, pre-sorted by virtual address.

Figure 4.1: Conventional and ASAP-accelerated page table walks. Darker (lighter) PT node colors are associated with higher (lower) virtual addresses of the corresponding data pages [1].

4.2 ASAP Design

Address Translation with Prefetching – a mechanism to prefetch PT nodes ahead of the page walker. We focus on the deeper levels of the PT (PL1 and PL2) as the most valuable prefetch targets, because the fourth (PL4) and third (PL3) PT levels are small and efficiently covered by the Page Walk Caches and the regular cache hierarchy. Meanwhile, the second (PL2) and the first (PL1) levels are much larger and often beyond reach for on-chip caching structures for big datasets. For instance, for a 100GB dataset, the footprint of the PT levels is 8B, 800B, 400KB and 200MB for PL4, PL3, PL2 and PL1, respectively.

To reduce the page walk latency, we propose issuing prefetches for the PL1 and PL2 levels concurrently with the page walker initiating its first access (i.e., to PL4, which is the root of the PT), as demonstrated in Figure 4.1b. Triggered on a TLB miss, the prefetcher needs to determine the physical addresses of the target PT nodes in both PL1 and PL2 levels of the PT. This is accomplished via a simple base-plus-offset computation, enabled through an ordering of memory pages occupied by the PT as discussed below. Prefetches travel like normal memory requests through the memory hierarchy and are placed into the L1-D, thus maximally repurposing the existing machinery.

Critically, with ASAP, the page walker performs the full walk as usual, consuming the prefetched entries. By executing the page walk, ASAP guarantees that only correct entries are consumed, which enables proper handling of page faults and reduces the risk of introducing a new security vulnerability into an existing architecture. Security concerns also prohibit allocating prefetches to PWCs speculatively (i.e. before validating them by the full conventional page walk) because PWCs are indexed by virtual addresses. Indeed, if prefetches were allocated to PWCs, capacity conflicts could be used as a side-channel and reveal information about the virtual-to-physical mapping.

To introduce ASAP as a natural addition to the existing system stack, we exploit the observation that a process operates on few contiguous virtual address ranges. One important example of such a range is the heap, which forms a large contiguous region in the virtual space of a process. Each allocated virtual page inside a virtual address range has a PTE at the leaf level of the PT, reached through a chain of PT nodes, one per PT level (Figure 4.1a). Thus, there exists a one-to-one mapping between a virtual memory page and a corresponding PT node at each level of the PT radix tree. However, this correspondence exists only in the virtual space, but not in the physical space, due to the buddy allocator that scatters the virtual pages, including those of the PT, across physical memory.

To enable ASAP, there needs to be a direct mapping from a virtual page to a corresponding PT node in physical memory (shown by the grey Prefetch arrows in Figure 4.1b). Our insight is that if the PT nodes for a given level of the PT in physical memory follow the same order as the virtual pages they map to, then a direct index into the PT array is possible using simple base-plus-offset computation. The solution is to have the OS induce the required ordering for the PT nodes in physical memory. As discussed below, this requires straightforward extensions in the kernel and absolutely no modifications to the actual PT structure.

In the remainder of the section, we discuss key aspects of ASAP including existing

Application	Total VMAs	VMAs for 99% footprint coverage	Contig. phys. regions	PT page count
canneal	18	4	487	2842
mcf	16	1	626	3189
pagerank	18	1	2076	38504
bfs	14	1	4285	66015
mc80	26	6	1976	45878
mc400	33	13	5376	213097
redis	7	1	3555	44171

Table 4.1: Total number of VMAs, number of VMAs that cover 99% of footprint, number of contiguous regions in physical memory, and total number of PT pages per application.

contiguity in the virtual address space, how contiguity can be induced in the radix tree-based PT, architectural support for ASAP, and virtualization extensions.

4.2.1 Virtual Address Contiguity

Virtual addresses in PTs appear as a set of contiguous virtual ranges that are defined by the way the applications create and use virtual address spaces, such as heap and stack. In Linux, the OS manages these ranges using a virtual memory area (VMA) tree that contains the information about all non-overlapping virtual address ranges (further referred to as VMAs) allocated to a process. Other OSes have data structures analogous to Linux VMA tree, e.g., Virtual Address Descriptor (VAD) tree in Windows [52].

The applications we studied allocate few VMAs that stay stable during their execution. Our results (Table 4.1) show that a small number of VMAs cover 99% of the application footprint. These few large VMAs are attributed to heap and memory-mapped regions that contain the application data structures that are the primary causes of page walks. Meanwhile, small VMA mostly represent dynamically-linked libraries and the stack, which are frequently accessed and rarely cause TLB misses due to high temporal reuse.

4.2.2 Inducing Contiguity in the Page Table

As explained in the previous section, the virtual address space enjoys high contiguity. However, when it comes to physical memory, the pages of the PT are often scattered. For the applications we studied, the number of contiguous physical memory regions

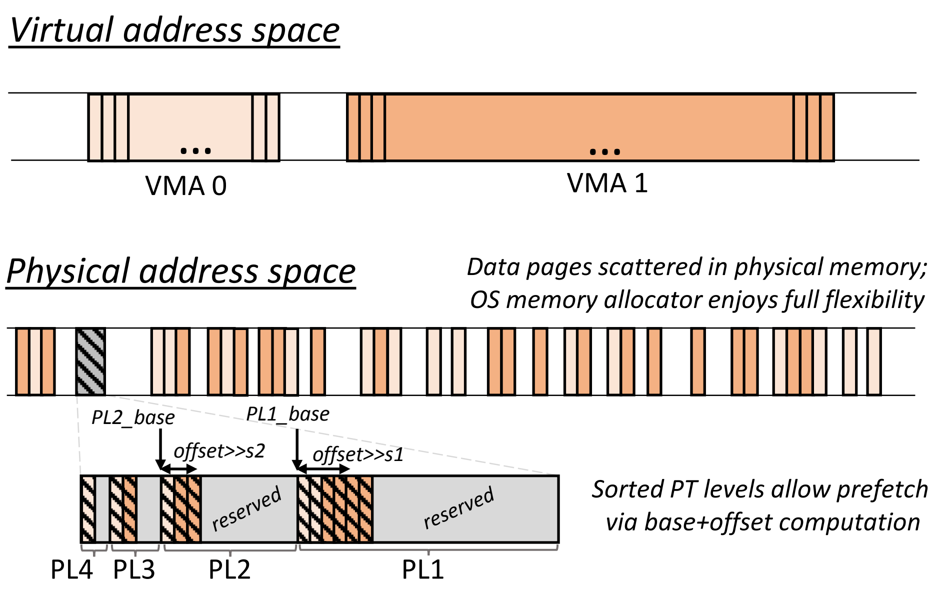

Virtual address space
VMA 0
VMA 1
Physical address space
Data pages scattered in physical memory; OS memory allocator enjoys full flexibility
PL2_base
PL1_base
offset>>s2
offset>>s1
reserved
reserved
Sorted PT levels allow prefetch via base+offset computation
PL4
PL3
PL2
PL1

Figure 4.2: Virtual and physical memory layout with ASAP. The pages that contain the PT are color-coded according to their corresponding VMAs [1].

that store the PT nodes can reach into thousands (Table 4.1). The reason for such lack of locality in the PT is that PT nodes, just as any other data in Linux, are lazily allocated in pages whose position in physical memory is determined by the Linux buddy allocator. The buddy allocator optimizes for allocation speed, allocating pages on demand in first available slots in physical memory. The result is a complete lack of correspondence between the order of virtual pages within a VMA and the physical pages containing PT nodes.

To enable ASAP, the OS needs to guarantee that PT nodes within each PT level are located in contiguous physical memory according to their corresponding virtual page numbers within a VMA of the process, conceptually shown in Figure 4.1b. There are two ways to achieve such placement of PT nodes in physical memory: first is to deploy a custom allocator to enforce page ordering and contiguity in physical memory for the PT, and second is to sort the already allocated PT nodes in the background. While both approaches are plausible, we focus on the former as a concrete case study.

In Linux, a VMA tree contains all ranges of virtual addresses (further referred to as VMAs) that the OS provides to the process per its request. According to the demand paging and lazy allocation principle, which is employed in most operating systems, including Linux, the VMAs are created immediately, e.g., upon an `mmap` system call, whereas PT nodes are created and populated only upon a first access to the corresponding virtual addresses, which cause page faults that lead to creation of the corresponding virtual-to-physical mappings in the PT. Hence, each VMA defines how many mappings will eventually appear for it in the PT, and what portion of the PT the VMA will occupy in physical memory.

Since the OS knows the beginning of each VMA in the virtual space and its size, the OS can reserve contiguous physical memory regions for PT nodes at each level of the PT ahead of the eventual demand allocation of PTEs. When these are (lazily) populated, the OS can further enforce the ordering of PT nodes within each PT level to guarantee that it matches the ordering of the virtual pages mapping to them. Doing so ensures both *contiguity* and *ordering* of PT pages in machine memory, which enables indexing into a given level of the PT.

Figure 4.2 shows the layout of virtual and physical address spaces with ASAP. The virtual space layout and the layout of data pages in physical memory remains the same as in vanilla Linux; the only change required by ASAP is the introduction of contiguous physical memory regions for pages containing PT nodes.

Cost. Unlike prior work on direct segment addressing [14], [31], [32], which requires allocating the entire dataset of an application within large contiguous physical regions (see Section 2.2.1 for a discussion of the drawbacks), ASAP requires contiguity in only a tiny portion of the physical memory thanks to the compact nature of the PT radix tree. As discussed in Section 4.2, for an application with a 100GB dataset, PL4 and PL3 footprints together fit in a single 4KB page, the PL2 footprint requires 400KB, and the leaf PL1 – necessitates around 200MB. This example shows that the physical memory footprint that must be guaranteed contiguous by the OS to hold the sorted PT nodes amounts to a mere 0.2% of an application's dataset size.

4.2.3 Architectural Support for ASAP

Figure 4.3 shows the microarchitecture of ASAP. As the figure shows, ASAP non-disruptively extends the TLB miss-handling logic. For each VMA that is a prefetch target, ASAP requires a VMA descriptor consisting of architecturally-exposed *range registers* that contain the start and end addresses of the VMA, as well as the base physical addresses of the contiguous regions containing the 1st (PL1) and 2nd (PL2) PT levels mapping the VMA. ASAP's VMA descriptors are part of the architectural state of the hardware thread and are managed by the OS in the presence of the events like a context switch or interrupt handling. According to the results from Section 4.2.1, tracking 8–16 VMAs is enough to cover 99% of the memory footprint for the studied benchmarks.

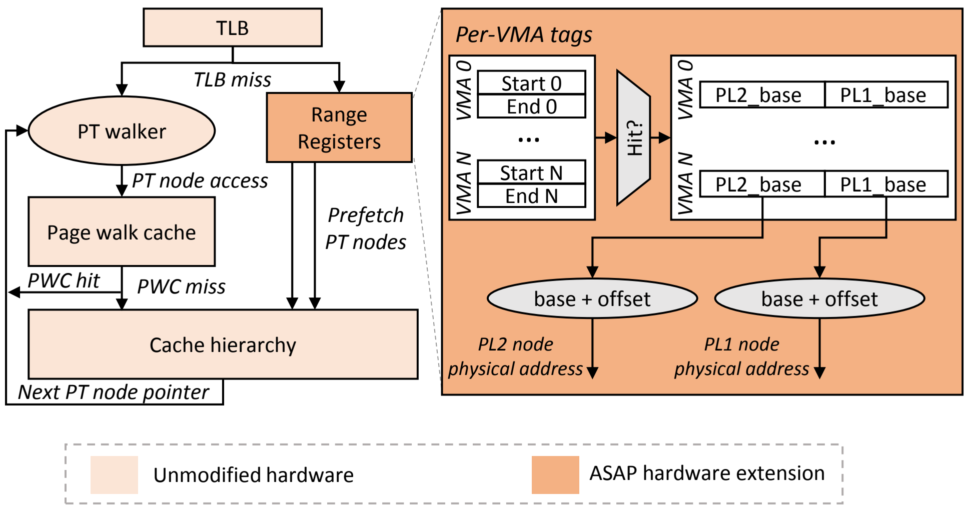


Figure 4.3: Architectural support for ASAP [1].

With ASAP, each TLB miss triggers a lookup into the range registers, which happens in parallel with the activation of the page walker. The lookup checks the virtual address of the memory operation against the tracked VMA ranges; on a hit, target prefetch addresses in PL1 and PL2 are calculated with a base-plus-offset computation using each level's respective base physical addresses and the offset bits from the triggering virtual address. Note that the actual offset differs between PL1 and PL2, and is derived for each of these PT levels by simply shifting the incoming offset bits by a fixed amount (labeled `s1` and `s2` in Figure 4.1b). The prefetch requests to the two target PT nodes are then issued to L1-D if it has a port available. As a result, the cache lines containing PT nodes are loaded into the L1-D, from which they will be subsequently accessed by the page walker.

An important aspect of ASAP is that it requires no modifications either to the cache hierarchy or to the page walker. ASAP leverages existing machinery for buffering the outstanding prefetch requests in caches' MSHRs and allocating the data brought in by ASAP in caches themselves. If L1-D is physically indexed, data brought by prefetches placed to L1-D, whereas if L1-D is virtually indexed – to L2. Prefetches are thus best-effort (e.g., not issued if an MSHR is not available). In contrast to data prefetchers, ASAP does not noticeably increase memory bandwidth pressure since ASAP prefetches are nearly always correct (except in the special cases of ”holes” in a PT range, as discussed in Section 4.2.4.2), effectively converting the page walker's demand misses into prefetches.

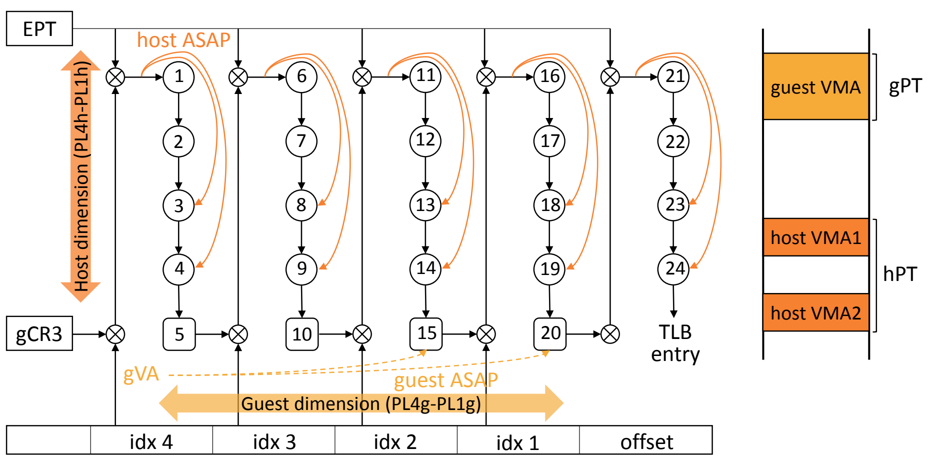


Figure 4.4: Nested prefetched address translation with ASAP. Accesses are enumerated according to their order in a 2D page walk [1].

4.2.3.1 ASAP for Large Pages and Five-Level PT

Thanks to the recursive structure of the PT radix tree, no modifications are required to support large pages of any size. Translations that correspond to large pages are stored one or two levels above from the leaf (PL1) PT level. For example, PTEs for 2MB pages are stored in the 2nd (PL2) level of the PT radix tree. Since the size of the page is unknown before the page walker inspects the ultimate PT node (e.g., the PT node at PL2 contains a dedicated bit that distinguishes a 2MB page PTE from a pointer to the PL1 node that contains a 4KB page entry), some of the prefetch requests may be redundant (e.g., a request to the PL1 node if 2MB pages are used).

With the advent of five-level PTs, ASAP can be naturally extended to issue an additional prefetch request to the added PT level.

4.2.3.2 ASAP for Nested Walks

In virtualized environment, ASAP can be applied in both guest and host dimensions, which presents a significant acceleration opportunity due to the high latency of nested page walks. Under virtualization, the radix tree levels of both guest PT (gPT) and host PT (hPT) targeted by ASAP must be contiguous and ordered in the host physical memory. Similar to the native setup, this must be ensured by the hypervisor and guest OS.

In the general case, 2D walk starts by reading the guest's value of the CR3 register that stores the location of the gPT root, followed by consecutive 1D walks in the host to access each of the gPT entries. Figure 4.4 shows the 2D walk with ASAP prefetching, assuming ASAP is configured to prefetch the 2nd (PL2) and 1st (PL1) levels of gPT and hPT. Immediately at the 2D walk start, ASAP issues prefetch requests to the gPT nodes in the PL2 and the PL1 levels to overlap accesses 15 and 20 with the previous ones. Then, just as the page walker starts the 1D walk in the host (steps 1–4), ASAP issues prefetch requests to the PL2 and PL1 levels of the hPT using the guest physical address of the gPT root. The process repeats for each 1D walk in host, namely steps 6–9, 11–14, 16–19, 21–24.

From the software perspective, to enable ASAP-accelerated 2D walks, the guest OS requires minimal modifications. Similar to the native case, the guest OS needs to ensure contiguity in the physical memory regions storing PL1 and PL2 levels of the PT. Under virtualization, the guest must make these requests to the hypervisor, and notify the hypervisor when any of these regions need to be extended. Thus, if the PT memory region requires an extension, the guest OS' system calls that change the contents of the (guest) VMA tree must execute `vmcall` instructions to trigger the transition into the hypervisor so that it can invoke the host OS's PT allocator to guarantee the region's contiguity in both host and guest physical spaces.

From the hardware perspective, accelerating address translation in the host with ASAP requires additional range registers. Crucially, we observe that in Linux/KVM virtualization, from the perspective of the host OS, an entire guest VM is a process that has a *single* virtual address region [21]. Hence, a single set of range registers is sufficient to map the guest VM (including the target PT) as a host VMA, allowing acceleration of walks in the host dimension (e.g., steps 1–4 or 21–24 in Figure 4.4). Meanwhile, the number of VMAs in the guest OS is unaffected by virtualization, requiring the same number of range registers for ASAP acceleration as in the native environment.

4.2.4 Discussion

4.2.4.1 Page Fault Handling

Since most OS'es follow the lazy allocation principle, the PT is populated with mappings on demand, i.e., the first access to a non-allocated page causes a page fault, leading to the mapping being created in the PT. Thus, both with and without ASAP, some of the PT nodes corresponding to a VMA region will stay uninitialized until the first access happens.

In this presence of ASAP, this behavior does not impact correct page fault handling.

Thus, when the page walker performs a page walk that eventually triggers a page fault, ASAP still issues prefetch requests to PT nodes in PL1 and PL2. These prefetches accelerate page fault detection by the hardware walker.

4.2.4.2 VMAs Evolution

Most VMAs in the VMA tree belong to well-defined process segments, such as heap, stack, memory-mapped files and dynamic libraries. The largest data segments – the ones that hold the application dataset, such as heap and the memory-mapped segments – can grow or shrink in a pre-determined direction as the process continues its execution. For instance, upon a `malloc` call, the allocator may grow the heap segment by invoking `brk/sbrk` system calls to extend the segment towards higher virtual addresses.

To extend the contiguous reserved PT regions in the event of a VMA extension, the OS needs to request memory from the buddy allocator next to the boundary of the existing region. Unfortunately, the buddy allocator does not optimize for contiguous region allocations, usually providing the first best fit chunk of physical memory. Furthermore, the physical memory next to the border of the region can be already allocated (e.g., for regular data pages). To avoid changing the buddy allocator mechanisms, we argue for asynchronous regions extension in the background, triggered by a system call that extends the corresponding VMA. A similar mechanism for asynchronous regions extension in the background is employed by *Transparent HugePages* [53] daemon that can compact pages per a request/hint communicated by the application via `madvise` system call (e.g., `MADV_MERGEABLE` advice value).

In the unlikely event that the OS cannot free some of the pages in the region extension area (e.g., if the pages are pinned), it can allocate some of the PT pages apart from the reserved region in the VMA. Thanks to the pointer-based structure of the PT radix tree, the page walker will be able to correctly walk the PT as usual. The only consequence of such "holes" in the reserved PT regions is that the page walks that target the PT entries located in the "holes" would not be accelerated by ASAP.

We acknowledge that PT page migration could result in locking of PTs pages and potentially slow down page fault handling. As a result, this design is effective for applications that experience VMA changes during a short initialization phase (e.g loading a dataset from disk to memory) and have a long execution phase without VMA changes. For such applications, the potential drop in performance stemming from rearranging the PT pages during the initialization phase could be amortized by faster

Parameter	Value/Description	
	x86-64 platform	AArch64 platform
Processor	Dual socket Intel® Xeon® E5-2630v4 (BDW) 2.4GHz 20 cores/socket, SMT	Dual socket Huawei Kunpeng 920-6426 2.6GHz 64 cores/socket, no SMT
Memory size	160GB/socket (768GB/socket for `mc400`)	128GB/socket (`mc400` is not studied)

Table 4.2: Parameters of hardware platforms used for memory trace generation.

page walks accelerated by ASAP during the main execution phase.

An alternative design for arranging the PT for ASAP is reserving a part of the region created by the Linux Contiguous Memory Allocator for PTs by the OS when an application starts. This approach eliminates the potential performance losses from migrating PTs but comes at the cost of slightly increased memory consumption (less than 0.5%). However, this cost can be removed in scenarios when the amount of memory that would be used by an application is known in advance.

4.3 Methodology

To evaluate ASAP, we employ a methodology similar to prior work in this space, which reports TLB-miss induced overheads and page walk latencies [10], [14], [15], [31]. Given our focus on long-running big-memory workloads, we find full-system simulations intractable for projecting end-to-end performance. Instead, we report average page walk latency obtained from a detailed memory hierarchy simulator that models processor caches, page walk caches and TLBs.

Our evaluation primarily focuses on small (4KB) pages, since fine-grain memory management delivers greater flexibility, better memory utilization and better performance predictability (Section 2.1.2.1). We explore the effect of large pages in Section 4.4.2.

Measuring page walk latency. As a primary evaluation metric for ASAP, we use page walk latency. We evaluate ASAP on x86-64 and AArch64 platforms. To that end, we functionally model the memory hierarchy of an Intel Broadwell-like or a Huawei Kunpeng-like processors, respectively, using a simulator based on DynamoRIO [49]. We extend the DynamoRIO's default memory hierarchy simulator and add support of TLBs, PWCs, and nested page walks.

Parameter	Value/Description	
	x86-64 CPU	AArch64 CPU
L1 I-TLB	64 entries, 8-way associative	48 entries, 12-way associative
L1 D-TLB	64 entries, 8-way associative	32 entries, 8-way associative
L2 S-TLB	1536 entries, 6-way associative	1024 entries, 4-way associative
Page walk caches	3-level Split PWC: 2 cycles, PL4 - 2 entries, fully assoc.; PL3 - 4 entries, fully assoc.; PL2 - 32 entries, 4-way assoc. (similar to Intel Core i7 [54]) Virtualization: one dedicated PWC for guest PT, one for host PT	
L1 I/D cache	32KB, 8-way associative, 4 cycles round trip	64KB, 4-way associative, 3 cycles round trip
L2 cache	256KB, 8-way associative, 12 cycles round trip	512KB, 8-way associative, 9 cycles round trip
L3 cache	20MB, 20-way associative, 40 cycles round trip	48MB, 12-way associative, 62 cycles round trip
Main memory access latency	191 cycles round trip	240 cycles round trip

Table 4.3: Parameters used in simulations.

As part of the simulation, we record a trace of the application's memory accesses and the content of its PT on a real system, whose parameters are listed in Table 4.2. For experiments with virtualized The parameters of simulated CPUs are shown in Table 4.3.

During simulation, using the recorded trace of memory accesses, we model updates to the memory hierarchy state, including states of caches, TLBs, and PWCs. On every TLB miss, we simulate a page walk using the application's PT dump. Thus, memory accesses triggered by a page walker also update the state of the memory hierarchy. A PT dump includes the whole kernel and user PTs of a studied process. A PT dump is captured through an in-house kernel module. On the x86-64 platform, we use a kernel module derived from the Linux kernel debug helper for dumping PTs [55]). On the AArch64 platform, we use PTEditor library [56] designed for dumping a PT. During simulation, a PT dump is used to determine physical addresses that should be read during a page walk. For each access to the memory hierarchy during a page walk, we trace and record the levels of the memory hierarchy involved in serving the access. Thus, for each page walk, we get a trace of memory hierarchy levels involved during

Name	Description
mcf	SPEC'06 benchmark (ref input)
canneal	PARSEC 3.0 benchmark (native input set)
bfs	Breadth-first search, 60GB dataset (scaled from Twitter)
pagerank	PageRank, 60GB dataset (scaled from Twitter)
mc{80,400}	Memcached, in memory key-value cache, 80GB and 400GB datasets
redis	In-memory key-value store (50GB YCSB dataset)

Table 4.4: Workloads used for evaluation.

the walk. Since a page walk is a serial pointer chasing operation, we calculate the page walk latency by adding up access latencies of all memory hierarchy levels involved in each page walk trace record.

Page walk caches configuration. We simulate PWCs configuration similar to Intel Core i7 [54] (see Table 4.3). Despite the fact that the considerable number of page walks hit to PWCs (see Figure 4.6), increasing PWCs capacity does not substantially reduce page walk latency. We observe that when running in isolation, doubling the capacity of each PWC with respect to the default configuration provides negligible average page walk latency reduction – 2% and 3% in native and virtualized scenarios, respectively. These results corroborate industry trends: PWCs capacity did not grow beyond 32 entries per level in several recent Intel processor generations from Westmere to Skylake [57].

Benchmarks. We select a set of 6 diverse applications that exhibit significant TLB pressure (6-85% L2 TLB miss ratio) from SPEC'06, PARSEC 3.0, graph analytics (atop of Galois framework [58]) and in-memory key-value stores. For the graph applications (`bfs` and `pagerank`), we used a 60GB synthetic dataset with edge distribution modeled after a (smaller) publically-available Twitter dataset. The applications and datasets are listed in Table 4.4. Unfortunately, due to an internal bug in the DynamoRIO on AArch64, we were unable to evaluate ASAPon `mc` running an AArch64 CPU.

Workload colocation. In modern datacenter and cloud environments, applications are aggressively colocated for better CPU and memory utilization [59]. Indeed, Google reports that they aggressively colocate different applications on SMT cores as a routine practice [9]. Intel offers a lot of server-grade processors supporting SMT while Huawei announced that its SMT-supporting server-grade processor, Kunpeng 930, will be released in 2021 [60].

We simulate a colocation scenario on a dual-threaded SMT core by placing a memory-intensive corunner thread alongside the studied application thread.

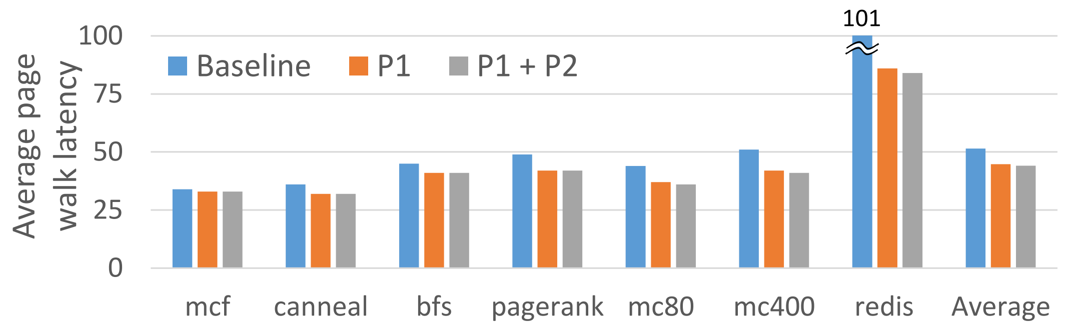
To show the full potential of ASAP, we use a synthetic corunner that issues one request to a random address for each memory access by the application thread. On both x86-64 and AArch64 platforms, colocation pressures the cache hierarchy, which is used to cache PTEs (from both intermediate and leaf nodes of the PT), hence increasing the average walk duration. By design, our microbenchmark considerably hampers the performance of the caches. To validate our methodology and show that such hampering effect is realistic and representative, we also evaluate ASAP in a colocation setup when a corunner is `mcf`. We find that, in colocation with `mcf`, ASAP's ability to accelerate page walks under virtualization is approximately the same as in colocation with our microbenchmark: 40% and 45% page walk latency reduction on average, respectively (see Section 4.4.2.1 for more detail).

We do not model contention in the TLBs and PWCs stemming from SMT colocation. This is because there is only limited and/or contradicting information regarding what partitioning schema is used for partitioning TLBs and PWCs in modern processors. Indeed, the Intel optimization manual [61] states that Intel Broadwell processors employ *fixed* partitioning of data TLBs. However, the term *fixed* is not defined in the document and no further details are given. On the other hand, the recent security research [62] reveals that data TLBs on a Broadwell processor are *competitively shared*. We argue that contention in TLBs and PWCs would result in a larger number of page walks and/or make them longer, thus increasing the opportunity for ASAP. Thus, our speed-up estimates for ASAP under colocation are conservative.

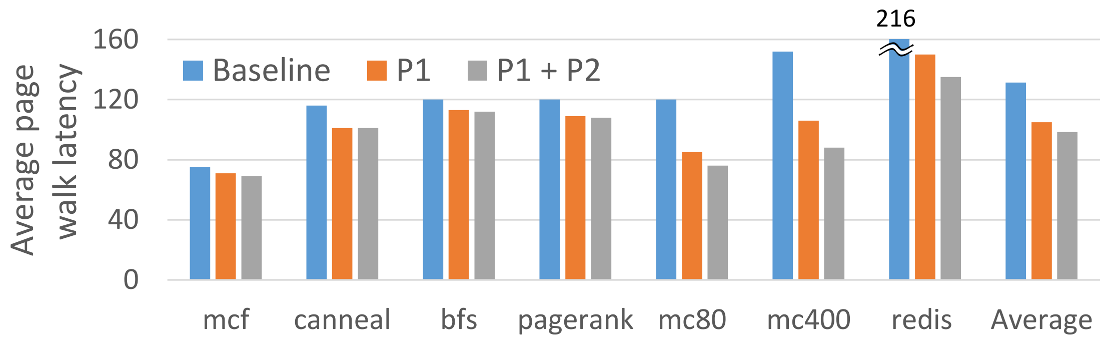
Virtualization. To assess ASAP in a virtualized environment, we record the guest PT contents using an in-house kernel module as described above. On the host side, we model the layout of the PT in a system without ASAP by mimicking the Linux buddy allocator's behavior by randomly scattering the PT pages across the host physical memory. To model ASAP, we maintain PL1 and PL2 pages in contiguous regions in the host.

4.4 Evaluation

In this section, we quantify the efficacy of ASAP in native and virtualized settings on x86-64 and in virtualized settings on AArch64, all with and without colocation. We also compare ASAP to state-of-the-art hardware and software techniques aimed



(a) In isolation.



(b) Under SMT colocation.

Figure 4.5: Average page walk latency in native execution (a) in isolation, (b) under SMT colocation [1]. Lower is better.

at reducing the cost of address translation, and show that ASAP is complementary to them.

4.4.1 ASAP in Native Environment

We first evaluate ASAP under native execution on x86-64, first without then with colocation. We evaluate several configurations. The first is the baseline, which does not employ ASAP and corresponds to a design representative of existing processors. We study two ASAP configurations: the first of these (referred to as *P1*) prefetches only from PL1 level of the PT; the second (referred to as *P1+P2*) prefetches from both PL1 and PL2 levels.

4.4.1.1 ASAP in Isolation

Figure 4.5a shows the average page walk latency for the baseline and both ASAP configurations when the application executes without a corunner. In the baseline, page walk latency varies from 34 to 101 cycles, with an average of 51 cycles. The largest latency is experienced by `redis`.

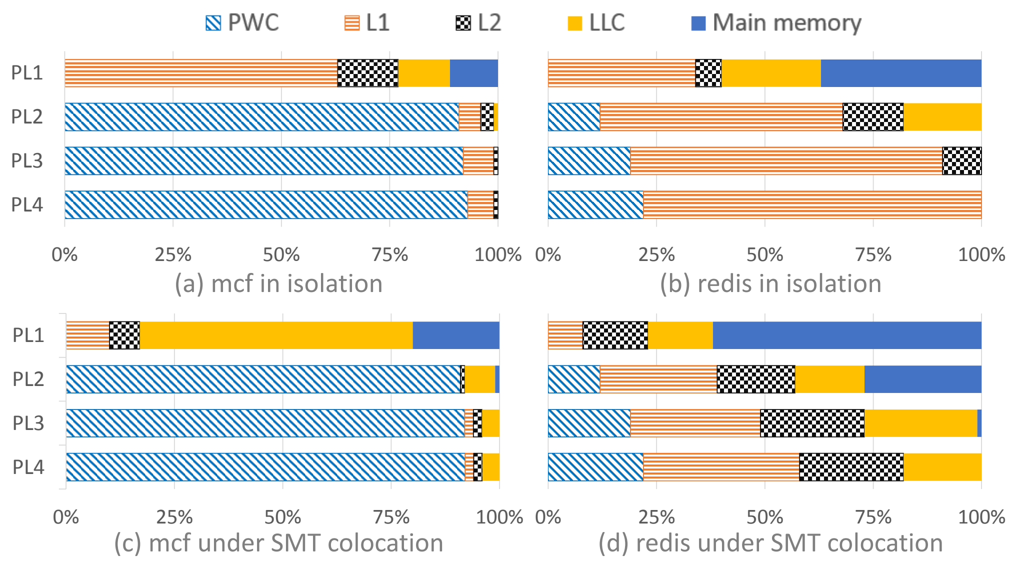


Figure 4.6: Fraction of page walk requests served by each level of the memory hierarchy for a given PT level [1].

Prefetching only PL1 reduces average page walk latency by 12% over the baseline (to 45 cycles). In absolute terms, the largest observed latency reduction is on `redis`, whose page walk latency drops by 15 cycles (or 20% compared to the baseline). In contrast, `mcf` experiences only 1 cycle reduction in average page walk latency. The difference in efficacy of ASAP for these applications can be explained by understanding from which level of the memory hierarchy page walk requests are served.

Figure 4.6 shows the fraction of requests satisfied by a given level of the memory hierarchy for each level of the PT traversed in a walk. In the case of `mcf` running in isolation (Figure 4.6a), requests to all levels except PL1 mostly hit in PWC and are satisfied within a few cycles; meanwhile, requests to PL1 take considerably longer, with nearly a third of requests hitting in L2, LLC or main memory. Because the page walker traverses PL1 through PL3 so quickly thanks to PWC hit, ASAP has little opportunity to hide latency on this workload. Meanwhile, `redis` hits in PWC much less frequently as shown in Figure 4.6b; in particular, a significant fraction of page walker's requests to PL2 reaches the L2 or LLC, which provides ASAP with an opportunity to overlap its prefetch to PL1 with the page walk to previous levels.

Despite the fact that a considerable number of page walks can hit in PWC, increasing PWC capacity does not substantially reduce page walk latency. We observe that when running in isolation, doubling the capacity of each PWC with respect to the default configuration provides a negligible page walk latency reduction – 2% and 3% in native and virtualized scenarios, respectively. These results corroborate industry trends: PWC capacity has not grown beyond 32 entries per level in several recent Intel processor generations from Westmere to Skylake [57].

Prefetching PL2, in addition to PL1, reduces the average page walk latency by 14% over the baseline – a small improvement over prefetching just PL1. The reason for such limited benefit of prefetching from an additional level of the PT can be understood by examining Figures 4.6a and 4.6c). Because the vast majority of requests to PL3 and PL4 hit in PWC or the L1-D, there is little opportunity for ASAP to overlap these accesses with prefetches to PL2.

4.4.1.2 ASAP under Colocation

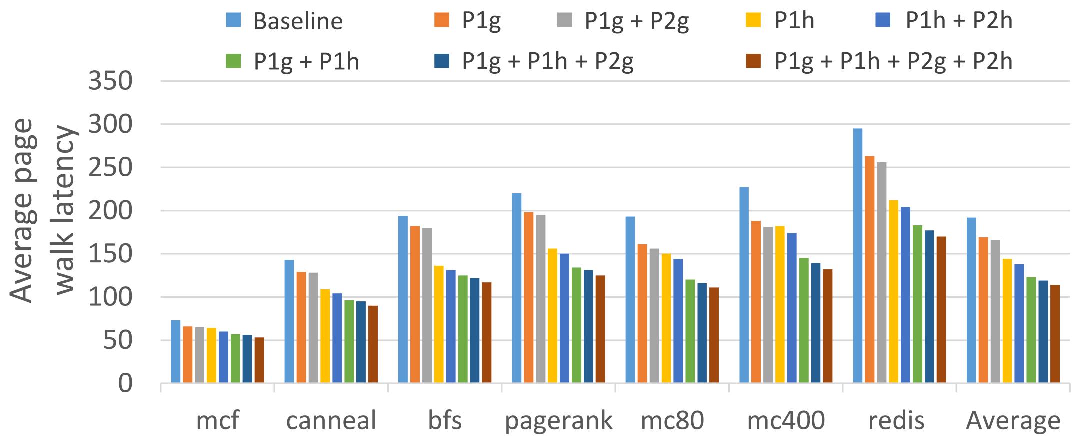
Figure 4.5b shows page walk latency for native execution under colocation. When the memory hierarchy experiences additional pressure due to the presence of a memory intensive corunner, page walk latency increases as compared to execution in isolation as PT nodes are more likely to be evicted from the caches. Comparing Figure 4.6b to Figure 4.6d, one can see that under colocation, there are considerably fewer page walk requests served by the L1-D cache than when running in isolation. As a result, the average page walk latency on the baseline with colocation ranges from 74 to 216 cycles, with an average of 131 cycles. This corresponds to an increase of 2.1-3.2× (average of 2.6×) over the execution in isolation.

With prefetching only to PL1, ASAP achieves a page walk latency reduction of 20%, on average, and 31% in the best case (on `redis`, whose average page walk latency drops by 66 cycles). When page walks frequently contain more than one long-latency request – such as when requests to PL1 and PL2 are both served by the main memory, as in Figure 4.6d – ASAP's ability to shorten the page walks latency significantly improves by overlapping the latency of these requests.

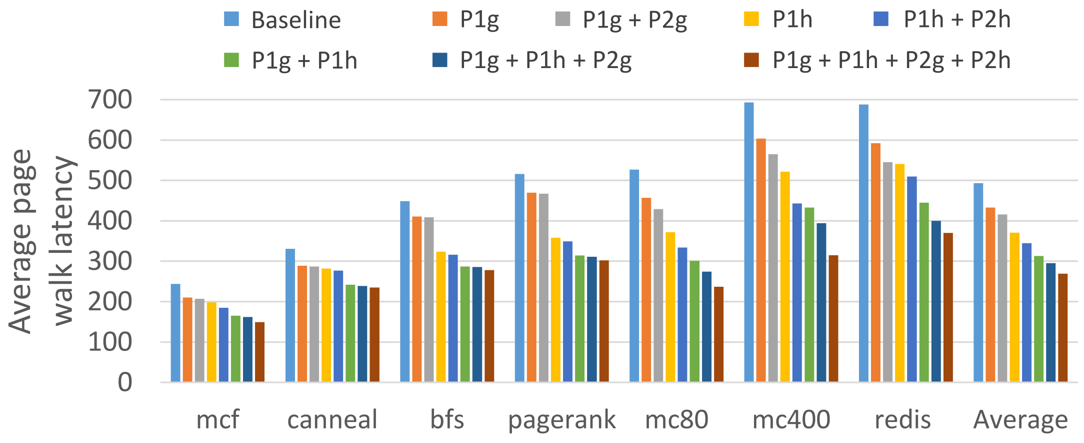
Prefetching PL2, in addition to PL1, is also more beneficial in the presence of a corunner. Prefetching both levels reduces the page walk latency by 25% on average and up to 42% (on `mc` with 400GB dataset), over the baseline.

4.4.2 ASAP under Virtualization

To understand ASAP's efficacy in a virtualized setting, we study several ASAP configurations that prefetch from PL1 only or from both PL1 and PL2 in the guest and/or host. In this scenario, we evaluate ASAP x86-64- and AArch64-based systems. The baseline corresponds to a system without ASAP.



(a) In isolation.



(b) Under SMT colocation.

Figure 4.7: Average page walk latency on the x86-64 CPU with virtualization (a) in isolation, (b) under SMT colocation [1]. Lower is better. Note the different scaling of y-axis between the subfigures.

4.4.2.1 On x86-64

Results for execution in isolation on x86-64 are shown in Figure 4.7a. Under virtualization, the baseline page walk latency ranges from 83 to 320 cycles, with an average of 227, a 4.4× increase in comparison to native execution due to the high cost of 2D walks. We observe that prefetching from PL1 of only the guest (*P1g* in the figure) reduces the average page walk latency by 13% on average.

Prefetching from both PL1 and PL2 (*P1g+P2g*) in the guest shortens the page walk latency by another 2%, totaling a 15% average reduction over the baseline. Such modest results can be explained by the fact that the nested page walk spends most of its time traversing the hPT (Section 4.2.3.2), which is not accelerated by ASAP that

mcf	canneal	bfs	pagerank	mc80	redis	Average
26%	38%	39%	37%	46%	42%	40%

Table 4.5: Average page walk latency reduction by the ASAP configuration prefetching PL1 and PL2 in both guest and host PTs in colocation with `mcf` on x86-64.

prefetches only from the gPT.

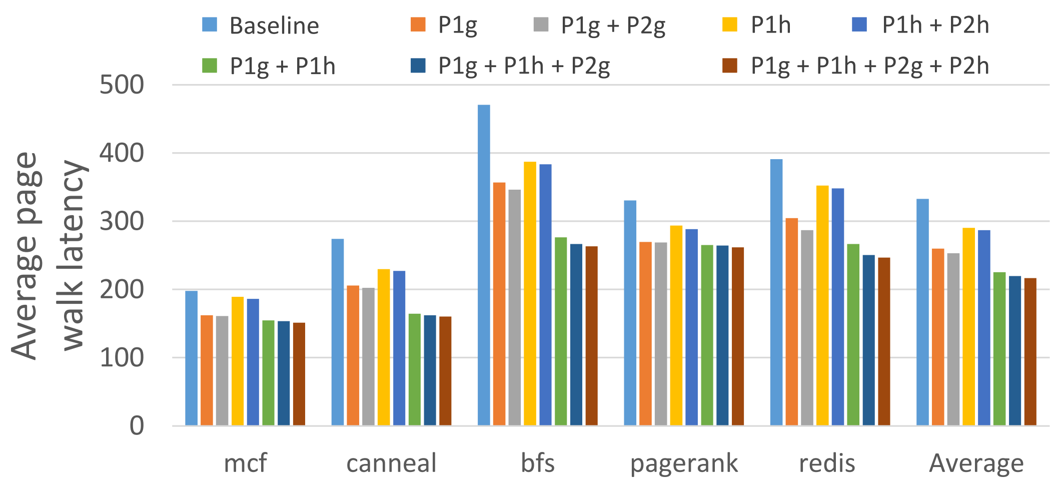
When ASAP prefetches from PL1 of the guest *and* from PL1 of the host together (*P1g+P1h*), walk latency decreases by 35% on average. Not surprisingly, the highest performance is attained if both PL1 and PL2 are prefetched in both guest and host (*P1g+P1h+P2g+P2h*). In that case, page walk latency decreases by 39% on average, and 43% (on `pagerank`) in the best case. In absolute terms, this configuration reduces page walk cycles by 88 on average and 137 max (on `pagerank`).

In a virtualized setting with workload colocation, there exists a larger opportunity for ASAP to capitalize on. Figure 4.7b shows that the baseline page walk latency under colocation increases considerably (on average, 493 cycles with colocation versus 227 cycles without), which indicates that there are more long memory accesses which can be overlapped with ASAP. Prefetching from PL1 in both guest and host reduces average page walk latency by 37% under colocation. Meanwhile, prefetching from PL1 together with PL2 in both guest and host under workload colocation reduces page walk latency by an average of 45%. The best-case improvement of 55% is registered on `mc` with 400GB dataset, whose average page walk latency drops by 378 cycles.

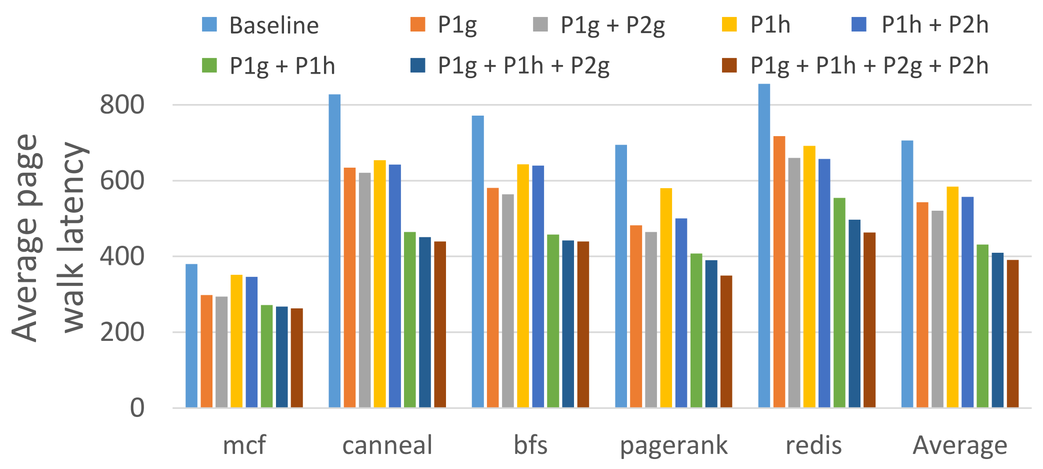
Colocation with `mcf`. To validate our evaluation methodology and show that our microbenchmark, which is used as a corunner, causes a realistic hampering effect on the performance of caches, we evaluate ASAP in a colocation setup when a corunner is `mcf`. The results of this study for the ASAP configuration prefetching PL1 and PL2 in both guest and host PTs are shown in Table 4.5. We find that in colocation with `mcf`, ASAP is capable to deliver a 40% page walk latency reduction on average. This result is close to ASAP's page walk latency reduction in colocation with our microbenchmark where on average, ASAP shortens page walks by 45%. As a result, we conclude that our studies with the microbenchmark produce representative results.

4.4.2.2 On AArch64

Figure 4.8 shows the results of ASAP's evaluation under virtualization on the modeled AArch64 CPU. Despite having larger caches than the modeled x86-64 CPU, the AArch64



(a) In isolation.



(b) Under SMT colocation.

Figure 4.8: Average page walk latency on the AArch64 CPU with virtualization (a) in isolation, (b) under SMT colocation. Lower is better. Note the different scaling of y-axis between the subfigures.

CPU experiences a higher overhead of page walks: on AArch64, the average page walk latency is 1.7× higher in isolation and 1.4× under SMT colocation compared to that on x86-64. Such a result can be explained by the fact that the AArch64 CPU has longer round trips to LLC and the main memory, considerably increasing page walk latency.

An important difference of ASAP's results on the AArch64 CPU from that on the x86-64 CPU is a higher page walk latency reduction delivered by prefetching PT nodes of the guest OS than of the host OS. For example, under SMT colocation, prefetching from PL1 in the guest OS reduces page walk latency by 30% on AArch64 versus just 14% on x86-64. The high effectiveness of prefetching PL1 in the guest on AArch64 can be explained by the fact that prefetching guest PT nodes not only hides the latency

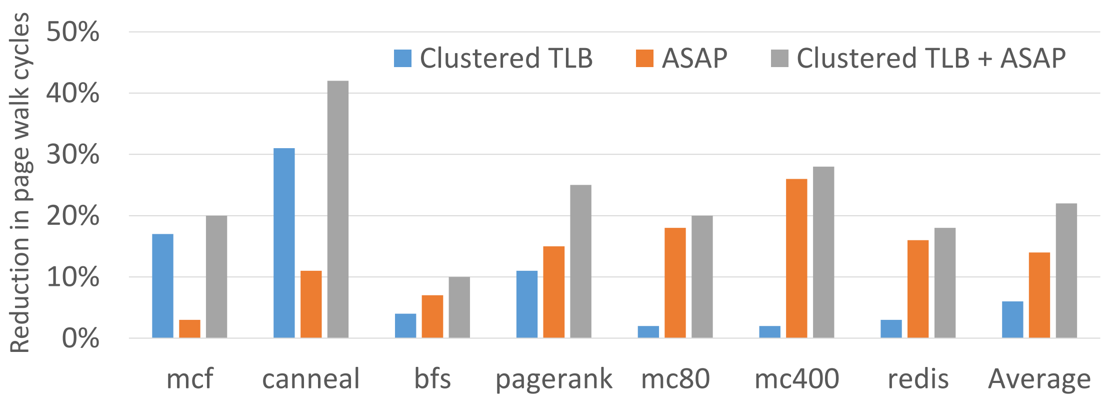


Figure 4.9: Reduction in the number of CPU cycles spent in page walks for Clustered TLB, ASAP, and the two together [1]. Native execution in isolation (higher is better).

of the corresponding memory access but also allows to start translating guest PT node's value in the host OS earlier. As a result, prefetching PL1 allows to hide the latency of accesses to a guest PT node and following accesses to the host PT. Indeed, the access to the guest PTE and the following accesses to the host PT (accesses with numbers 20-24 as shown on Figure 4.4) can be completely overlapped with translating values of guest PT nodes through the host PT (accesses with numbers 1-19). Long round trip latencies of LLC and the main memory on the AArch64 result in increasing the weight of translating values of guest PT nodes through the host PT in a page walk, increasing the opportunity for overlapping memory requests.

Similarly to results on x86-64, on AArch64, the highest performance is delivered by the ASAP configuration prefetching from PL1 and PL2 in both guest and host ($P1g$+$P1h$+$P2g$+$P2h$). On average, this ASAP configuration reduces page walk latency by 35% (44% max on `bfs`) in isolation and by 45% (50% max on `pagerank`) in colocation.

4.4.3 Comparison to Existing Techniques

In this section, on x64-64, we compare ASAP with state-of-the-art microarchitectural and software techniques and demonstrate their synergy with ASAP.

4.4.3.1 TLB Coalescing

TLB coalescing techniques [43], [44] detect and exploit available contiguity in virtual-to-physical mappings by coalescing TLB entries for adjacent pages into a single entry. Doing so increases effective TLB capacity and reduces TLB MPKI.

mcf	canneal	bfs	pagerank	mc80	mc400	redis	Average
58%	48%	10%	16%	4%	9%	12%	15%

Table 4.6: Reduction in TLB MPKI with clustered TLB. The data is normalized to native execution in isolation.

We evaluate Clustered TLB [44], a state-of-the-art TLB coalescing technique that coalesces up to 8 PTEs into 1 TLB entry. Table 4.6 shows the TLB MPKI reduction thanks to Clustered TLB. We find that Clustered TLB is highly effective for applications with smaller datasets, specifically `mcf` and `canneal`, reducing TLB MPKI by 58% and 48%, respectively. However, on the rest of the applications, which have much larger datasets (see Table 4.4), Clustered TLB is less effective, and TLB MPKI reduction varies from just 4% to 16%.

Figure 4.9 shows the reduction in page walk cycles with Clustered TLB, ASAP, and the two combined. Results are normalized to a baseline without either Clustered TLB or ASAP. On average, Clustered TLB reduces cycles spent in page walks by 5%, with largest improvement coming from workloads with small datasets. The reduction in the number of page walk cycles is smaller than reduction in TLB MPKI because the PT nodes accessed by page walks that are eliminated by Clustered TLB are the ones highly likely to be in higher-level caches due to spatio-temporal locality. Thus, clustered TLB eliminates mostly short page walks, leaving uncovered long page walks that access LLC and memory.

In contrast, ASAP is particularly effective in accelerating long page walks, particularly when both PL1 and PL2 nodes miss in higher-level caches. As a result, ASAP and clustered TLB naturally compliment each other and, when combined, can deliver additive performance gains. As shown in Figure 4.9, ASAP alone decreases the number of cycles spent in page walks by 14%, on average. Combining clustered TLB with ASAP increases TLB reach *and* reduces the walk latency, eliminating 22% of page walk cycles, on average, and 41% in the best case (on `canneal`).

4.4.3.2 ASAP with Large Pages

Under native execution, large pages can effectively tackle the high overhead of address translation as large pages ① reduce TLB miss ratio and ② shorten page walks (from 4 to 3 memory accesses). Thus, in native execution, while ASAP can be enabled for an application that uses large pages, ASAP is not expected to deliver a considerable per-

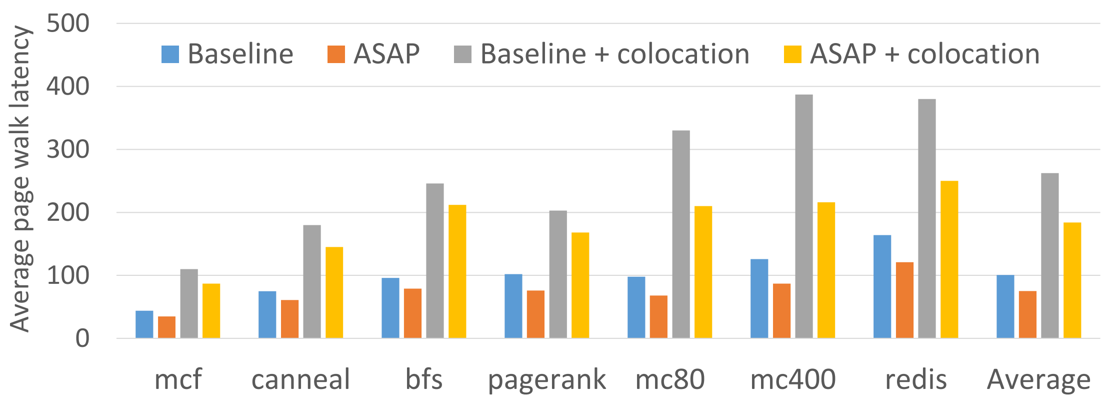


Figure 4.10: Average page walk latency with virtualization when hypervisor uses 2MB pages (lower is better). Baseline corresponds to execution in isolation.

formance improvement. Thus, we do not recommend enabling ASAP for applications that are exclusively backed by larger pages. With that being said, we note that there are numerous scenarios when large pages can not be used (see Section 2.1.2.1) and ASAP is effective under native execution.

There is a special scenario of using large pages under virtualization that deserves a separate discussion. A common optimization employed by modern hypervisors under low to moderate memory pressure is allocating guest physical memory as a collection of large pages [21]. Doing so eliminates up to five long-latency accesses to the memory hierarchy on each walk (i.e., accesses 4, 9, 14, 19, 24 in Figure 4.4).

We evaluate ASAP with 2MB host pages, with prefetching from both PL1 and PL2 in the guest and PL2-only in the host. Figure 4.10 depicts the results for this study. The baseline corresponds to execution in isolation with host using 2MB pages. ASAP reduces page walk latency by 25%, on average, over the baseline, and by up to 31% in the best case (on `mc` with 400GB dataset).

Under colocation, the average page walk latency increases by 2.6× as compared to execution in isolation. In this scenario, ASAP reduces page walk latency by 30%, on average, and by 44% in the best case on `mc` with 400GB dataset, whose average page walk latency reduces by 171 cycles). Overall, we conclude that even with shortened page walks enabled by 2MB pages, ASAP delivers a considerable reduction in page walk latency.

	mcf	canneal	bfs	pagerank	redis	Average
Fraction of cycles spent in page walks on the critical path	31%	24%	68%	50%	18%	34%
ASAP's reduction in average page walk latency	25%	32%	41%	43%	33%	39%
ASAP's minimum performance improvement	8%	8%	28%	22%	6%	12%

Table 4.7: Conservative projection of ASAP's performance improvement.

4.4.4 Estimation of Performance Improvement

In this section, we estimate performance improvement of ASAP which prefetches PL1 and PL2 in both guest and host when running in isolation under virtualization. We ① quantify the fraction of cycles spent in page walks *on the critical path*, and ② obtain a conservative estimate of ASAP's performance improvement by multiplying this fraction with ASAP's average reduction in page walk latency (see Figure 4.7a).

To quantify the fraction of cycles in page walks on the critical path, we use the methodology similar to the one used in Section 3.4.1. We measure execution time in the absence of TLB misses (hence, no page walks) and compare that to normal execution with TLB misses. To eliminate TLB misses, we run the applications using a small (3GB or smaller) dataset while forcing an application to use large pages with *libhugetlbfs* [50]. With such a small dataset and large pages enabled, the capacity of L2 S-TLB (1536 entries) is enough to capture the whole PT. As a result, we achieve a significant (~100×) reduction in the number of page walks. The reduction in execution time due to page walks elimination corresponds to page walk cycles on the critical path. Note that using large pages can significantly reduce the number of page walks *only* for datasets smaller than 3GB (reach of the TLB). This and other limitations of large pages (see Section 2.1.2.1) make their use in a datacenter problematic.

We study all the applications except `memcached`, which is unaffected by *libhugetlbfs*. The results of the study are shown in Table 4.7. With page walks eliminated, the largest reduction in total execution time compared to a configuration where page walks are present is observed on graph workloads – 68% on `bfs` and 50% on `pagerank`. Projecting these results on ASAP, which in isolation under virtualization reduces average page walk latency by 41% on `bfs` (43% `pagerank`), ASAP improves performance by 28% (22% on `pagerank`). On average, on the x86-64 platform, ASAP is estimated to improve performance by 12%.

4.5 Related Work

Virtual memory and address translation have been a hot research domain, with prior work explore the following ideas.

Improving TLB reach. Prior art suggests a number of mechanisms to boost TLB's effective capacity by coalescing adjacent PT entries [33], [43], [44] or by sharing TLB capacity among CPU cores [63], [64] including the die-stacked L3 TLB design [19]. ASAP's advantage over the die-stacked L3 TLB is its microarchitectural simplicity and ability to work well under colocation. ASAP requires just a set of registers and simple comparison logic, whereas L3 TLB requires more than 16MB of die-stacked DRAM. Moreover, by heavily relying on the cache hierarchy, under colocation, L3 TLB is likely to suffer from thrashing and can experience increased miss rates. Thus, even L3 TLB would benefit from ASAP. Ultimately, TLB enhancements are constrained by a combination of area, power and latency. Given the continuing growth in dataset sizes, it is imperative to accelerate the latency of TLB misses, which is precisely the target of ASAP. As shown in Section 4.4.3.1, ASAP is complementary to techniques that coalesce adjacent PT entries.

TLB entries prefetching. Prior work explores a number of prefetch techniques to decrease the number of TLB misses. Kandiraju et al. study stride and markov TLB prefetchers that rely on available spatial and temporal locality of consecutive TLB misses [65]. Lustig et al. exploit inter-core prefetching that is efficient for the workloads that exhibit sufficient dataset sharing [64]. While these techniques mitigate the translation overheads for the workloads with regular memory access patterns, ASAP is oblivious to the TLB misses origin, decreasing the penalty of all the TLB misses including those induced by the irregular access patterns that are beyond the reach of TLB prefetchers.

Speculative address translation. SpecTLB [66] interpolates on existing TLB entries to predict translations when a reservation-based memory manager is used, as in FreeBSD [67], [68]. Thus, SpecTLB allows speculative execution of memory operations before their correctness is verified. This approach may pose security threats inherent to speculative execution of memory operations, as demonstrated by recent attacks such as Spectre [69], Meltdown [70], and Foreshadow [71]. In contrast, ASAP never consumes prefetched entries unless validated by a full page walk.

Translation-triggered prefetch. Bhattacharjee observes that if a page walker accesses main memory when servicing a TLB miss, the corresponding data is also likely to be memory-resident [72]. Hence, the author suggests enabling the memory controller to complete the translation in-place, so as to immediately prefetch the data for which the address translation is being carried out. This optimization can be seamlessly combined with ASAP, whose prefetches would reduce the latency of both the page walk and the data access.

Virtualization and nested page walks. Nested PTs introduce a significant performance overhead due to the elevated number of memory accesses in a page walk. Some researchers seek to limit the number of accesses by flattening the host PT [73], while the others use a unified PT structure, called shadow PT, managed by hypervisor [74]. Finally, Gandhi et al. combines nested and shadow PTs with a mechanism that dynamically switches between the two [75]. All of these techniques would benefit from ASAP, which would further reduce page walk latencies.

4.6 Conclusion

Existing techniques for lowering the latency of address translation without disrupting the established virtual memory abstraction all rely on caching – in TLBs, page walk caches and in the processor's memory hierarchy. Problematically, the trend toward larger application datasets, bigger machine memory capacities and workload consolidation means that these caching structures will be increasingly pressured by the need to keep an ever-larger number of translations. Thus, high page walk latencies due to frequent memory accesses are bound to become a *"feature"* of big-memory workloads.

This work takes a step toward lowering page walk latencies by prefetching PTEs in advance of demand accesses by the page walker, effectively uncovering memory level parallelism within a single page walk. This idea, which we call Address Translation with Prefetching (ASAP), is powered by an insight that the inherently serial radix tree traversal performed on a page walk can be accelerated through direct indexing into a given level of the PT. Such indexing can be achieved through a simple ordering of

PTEs by the OS without modifications to the underlying PT structure. While ASAP does expose the latency of at least two accesses to the memory hierarchy under virtualization, it is nonetheless highly effective, especially for virtualized and co-located workloads, reducing page walk latency by up to 55%. A strength of ASAP lies in the fact that it is a plug-and-play solution that works with the existing virtual memory abstraction and the full ensemble of today's address translation machinery.

Chapter 5

Improving Caching Efficiency of Page Table under Virtualization and Colocation

5.1 Introduction

Public cloud customers tend to execute their tasks in virtual machines, aggressively colocating the tasks to increase virtual machines' resource utilization. In Section 3.3, we find that the combination of virtualization and colocation result in the difference in average latencies of accesses to the leaf nodes of the guest and host PTs. In this chapter, we perform a root cause analysis of the difference. We demonstrate that the combination of virtualization and application colocation causes fragmentation of the guest physical memory space, which diminishes the efficiency of caching of host PTEs and leads to elevated page walk latencies. Based on this observation, we propose PTEMagnet, a new software-only technique that prohibits memory fragmentation within small regions of the guest physical memory, improves locality of accesses to host PTEs, and reduces page walk latency under virtualization and application colocation. As a result, PTEMagnet is especially beneficial for applications that ① frequently miss in TLBs and/or PWCs (e.g. have a large dataset size) and ② exhibit locality in memory access patterns on the granularity of pages (for example, they are likely to access a page $A+1$ if a page A was accessed). We evaluate PTEMagnet and show that that PTEMagnet is free of performance over-heads and on x86-64 and AArch64 platforms, it can improve performance by up to 9% and 10% (4% and 6% on average), respectively.

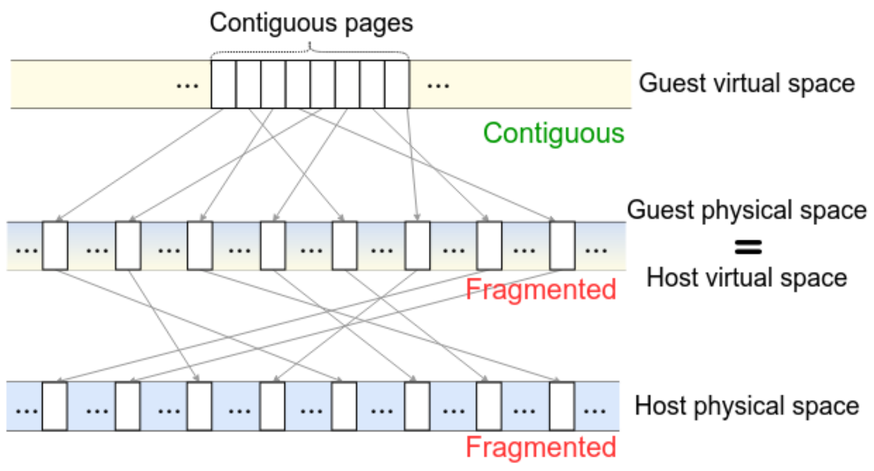


Figure 5.1: Contiguity (or lack of it) in virtual and physical address spaces under virtualization [2].

5.2 Challenges for Short Page Walk Latency under Virtualization and Colocation

Virtualization blurs the clear separation between virtual and physical address spaces. Modern virtualization solutions, e.g., Linux KVM hypervisor, are integrated with the host OS kernel allowing the host OS to reuse the bulk of existing kernel functionality, including memory allocation, for virtual machines. Hence, a virtual machine appears as a mere process for the host OS that treats the virtual machine's (guest) physical memory as a single contiguous virtual memory region [21]. As a result, one can think of the guest physical memory as the virtual memory for the host. Just like physical memory for any other virtual memory regions in the host OS, the physical memory for the guest OS is allocated on-demand and page-by-page when the guest actually accesses its physical pages.

As described in Section 2.1.3, while virtual address space features high contiguity, physical address space is highly fragmented, especially when under aggressive workload colocation. As a result, with virtualization and workload colocation, host virtual address space, being similar to guest physical address space, is highly fragmented too. In other words, virtualization does not propagate contiguity existing in the guest virtual space to the host virtual address space. Thus, the combination of virtualization and workload colocation breaks contiguity in the host virtual address space while keeping the guest virtual address space contiguous. Figure 5.1 represents contiguity (or lack of it) in virtual and physical address spaces under virtualization.

Lack of contiguity in the host virtual address space impairs spatial locality in the host PT. To understand this effect, consider a scenario when several applications run in the same virtual machine. Further, assume that each application allocates a region of guest-virtual memory that spans eight or more pages. Although pages in each of these regions are adjacent in guest-virtual memory, they are scattered across guest-physical memory. This is attributable to the fact that, after allocation of the regions, the applications start to access them concurrently, resulting in the interleaving of page faults to these regions. As a result, the Linux memory allocator in the guest OS fails to preserve guest-physical memory contiguity, assigning arbitrary guest-physical addresses to each of these pages so that these pages are distant from each other in the guest physical address space. As explained above, if the guest physical address space is fragmented, the host virtual address space is fragmented too.

Let's consider page walks performing address translation for a memory region allocated by an application in the scenario described in the previous paragraph. Page walks involve accessing guest and host PTEs (termed *gPTE* and *hPTE*, respectively). As discussed in Section 2.1.5, a single cache block with PTEs contains PTEs of eight pages neighbouring in a virtual address space. Due to spatial locality of the application's access patterns, the pages accessed by the application are likely to be close to each other in the guest virtual address space. Thanks to spatial locality, within a short period of time, a cache block holding gPTEs is likely to be accessed by multiple page walks that perform address translation for neighboring pages. In contrast, due to fragmentation in the host virtual address space, the hPTEs corresponding to pages neighbouring in the guest virtual address space are not located close to each other but scattered over different cache blocks. As a result, while accesses to gPTEs can benefit from spatial locality of application's access patterns, accesses to hPTEs cannot since fragmentation in the host virtual address space prohibits propagation of the spatial locality from guest virtual to host virtual address space.

The difference in the ability of accesses to gPTEs and hPTEs of exploiting spatial locality results in two consequences. Firstly, during a page walk, hPTEs are more likely to be fetched from the main memory than gPTEs. Secondly, such a difference makes the footprint of hPTEs larger than the footprint of gPTEs. In the extreme case, page walks of a group of eight pages would touch one cache block with gPTEs and eight cache blocks with hPTEs. Hereafter, we take the average number of hPTEs that correspond to a single cache block with tightly packed gPTEs as a metric of the host PT fragmentation.

5.2.1 Quantifying Effects of Fragmentation in the Host Page Table

Fragmentation of the host PT significantly increases its footprint in the CPU cache hierarchy (i.e., the number of cache blocks containing PTEs). A large PT cache footprint is obviously undesirable, since it presents a capacity challenge that is further amplified by cache contention (by application's code and data, as well as by corunning applications). Misses for PTEs in the CPU cache hierarchy necessarily go to memory, thus increasing page walk latency and hurting performance.

To showcase the effect of fragmentation in the host PT on performance, we construct an experiment where we run a representative `pagerank` benchmark from the GPOP graph workload suite [76] inside a virtual machine in isolation and in colocation with a memory-intensive corunner. As a corunner, we use `stress-ng` [77], configured to run 12 threads which continuously allocate and deallocate physical memory. As a consequence of colocation, host virtual memory space gets fragmented, which results in fragmentation in the host PT, thus increasing page walk latency. As a metric of performance, we measure execution time. Using *perf*, we also measure different hardware metrics to validate the fact that the change in performance stems from fragmentation in the host PT (see Section 5.4 for details of the complete setup). We analyze the source code of `pagerank` to identify a moment of execution by which `pagerank` finishes allocation of physical memory (namely, when it completes initializing all allocated data structures). Before collecting measurements, we stop the corunner after `pagerank` finishes allocation of physical memory, because by that moment the corunner already caused fragmentation in the host PT, which is the intended effect. As a result, when measuring `pagerank`'s performance, there is no contention for shared resources, such as LLC capacity, between `pagerank` and the corunner.

Table 5.1 represents changes in values of the measured metrics caused by fragmentation in the host PT. We observe that fragmentation in the host PT, caused by colocation with the memory-intensive corunner, increases execution time by 11%. We find that while not affecting the number of cache and TLB misses, fragmentation in the host PT increases the number of page walk cycles by 61%. Therefore, we conclude that the performance degradation is attributed to the change in the overhead of address translation.

We observe that colocation affects 63% of pagerank's contiguous memory regions, scattering their hPTEs to 8 distinct cache blocks. Overall, colocation raises the host PT fragmentation metric to 6.8, a significant increase from 2.8 observed in isolation.

Metric	Change
Execution time	+11%
Cache misses	<1%
TLB misses	<1%
Page walk cycles	+ 61%
Cycles spent traversing the host page table	+117%
Guest page table accesses served by main memory	+3%
Host page table accesses served by main memory	+283%
Host page table fragmentation (defined in Sec 5.2)	+242%

Table 5.1: Changes in metrics of `pagerank` in colocation with `stress-ng` as compared to standalone execution.

We find that fragmentation in the host PT has a nominal effect on the number of guest PT accesses served from memory. In contrast, the number of accesses to the host PT served from memory increases by 283%. As a result, in colocation, page walk incurs misses to host page table 4.4× more frequently than to the guest PT. Since a main memory access has higher latency than a cache access, with memory fragmentation, traversing the host PT takes more time, which increases page walk latency. Indeed, we observe a 117% increase in the number of cycles spent while traversing the host PT.

Our experiment shows that memory fragmentation under virtualization and colocation inside the same virtual machine can significantly increase page walk latency and degrade application performance.

5.2.2 Virtual Private Clouds: Virtualization + Colocation

While it is common knowledge that virtualization is a foundational technology in cloud computing, an astute reader might ask how likely are multiple applications to be colocated inside a single virtual machine. This section addresses this question.

Colocation in the same virtual machine is common-place in public clouds due to prevalence of services known as virtual private cloud (VPC), inlcuding Amazon VPC [78] and Google VPC [79]. These services allow internal and external users to run their applications on a cluster of virtual machines using an orchestration framework.

In a VPC, colocation of different applications in the same virtual machine occurs as a result of a combination of three factors. Firstly, a VPC typically includes a virtual machine that has a large number of virtual CPUs (vCPUs) and thus is capable of colocation. Such *large virtual machines* are needed to run large-memory applications as cloud providers tend to offer virtual machines with a fixed RAM-to-CPU ratio [80]. However, that is not the only scenario when a VPC includes a large virtual machine. Another case for including a large virtual machine in a VPC is reducing the costs by constructing a VPC from lower-cost available transient virtual machine instances. Such a policy selects the cheapest configuration of a virtual machine that can happen to be a large virtual machine [81]. Secondly, to increase utilization and reduce costs, cloud customers tend to run multiple different applications on a cluster at the same time [82]. Thirdly, cluster orchestration frameworks, such as Kubernetes [83]–[85], manage resources by a small unit of compute, typically one vCPU [86]. As a result of these factors, a machine with many vCPUs can receive a command to execute multiple different applications at the same time.

Colocation in the same VM is widely employed in Amazon Elastic Container Service (Amazon ECS) [87]. The bin packing task placement strategy that aims to fit Kubernetes tasks in as few EC2 instances as possible can easily cause a colocation of multiple applications in the same VM [88]. As a result, *any* applications can be colocated together in the same VM. Amazon ECS powers a growing number of popular AWS services including Amazon SageMaker, Amazon Polly, Amazon Lex, and AWS Batch, and is used by hundreds of thousands of customers including Samsung, GE, Expedia, and Duolingo [89].

Summary. Public clouds ubiquitously employ both virtualization and colocation inside the same virtual machine. Our analysis of the virtual memory subsystem in such an environment reveals that a lack of coordination between different parts of the memory subsystem – namely, the OS physical memory allocator and the address translation mechanism – leads to memory fragmentation in the host PT. This fragmentation increases the cache footprint of host PTEs, which results in elevated cache misses during page walks, thus increasing page walk latency and hurting application performance. Preventing fragmentation of the host PT in such an environment can thus improve cache locality for PTEs and increase performance.

5.3 PTEMagnet Design

Our goal is to prevent fragmentation of the host PT and reduce the latency of page walks under virtualization and colocation. We aim to achieve the latency reduction by leveraging existing CPU capabilities and without disrupting the existing software stack.

5.3.1 Design Overview

As shown in Section 5.2, the page walk overhead comes from the lengthy pointer chase through the fragmented host PT (hPT) due to its poor utilization of caches. Our key insight is that it is possible to reduce the page walk latency by increasing the efficiency of hPTE caching, namely grouping hPTEs corresponding to neighbouring application's pages in one cache block. Such placement can be achieved by propagating the contiguity that is naturally present in the guest PT (gPT) to the hPT.

To exploit the contiguity potential presented inside the guest-virtual address space for compacting hPTEs inside one cache block, one needs to guarantee that adjacent guest-virtual addresses are mapped contiguously onto adjacent host-virtual addresses or, equivalently, onto guest-physical addresses. This mapping criterion requires prohibiting fragmentation and introducing contiguity within small regions in the guest physical space. Since a CPU cache block contains eight 8-byte PTEs, to achieve the maximum locality for hPTEs, the contiguity degree in the guest physical space should be at least eight pages (see Figure 2.4). This means that, with 4KB pages, the size of the region contiguously allocated in the guest physical space should be 32KB. Meanwhile, the Linux/x86 page fault handler requests a single page from the buddy allocator on each page fault.

To eliminate fragmentation of the hPT, we introduce PTEMagnet – a new Linux memory allocator. PTEMagnet increases the current memory allocation granularity in the guest OS to the degree that maximizes cache block utility in the existing CPU hierarchy – that is, eight adjacent pages that correspond to eight adjacent hPTEs packed into a single cache block.

To support a different memory allocation granularity, we draw inspiration from the superpage (i.e., as in large page) promotion/demotion mechanism in FreeBSD [67] that relies on allocation-time physical memory reservations. Upon the first page fault to an eight-page virtual memory range, PTEMagnet reserves an eight-page long contiguous physical memory range inside the kernel so that future accesses to this page group get allocated to their corresponding pages inside the reserved range. This approach guarantees zero fragmentation for allocated hPTEs inside a cache block and minimizes the hPTEs footprint in the CPU memory cache hierarchy.

The allocation-time reservation approach adopted by PTEMagnet avoids costly memory fragmentation that is associated with using large pages because the OS keeps track of reserved pages and can reclaim them in case of high memory pressure.

In the remainder of this section, we discuss the key aspects of PTEMagnet design that includes the reservation mechanism and its key data structures, and the pages reclamation mechanism.

5.3.2 Page Group Reservation

PTEMagnet attempts to reserve physical memory for adjacent virtual page groups eagerly, upon the first page fault to any of the pages within that group. Upon such a page fault, a contiguous eight-page group is requested from the buddy allocator while only one virtual page (corresponding to the faulting page) is mapped to a physical memory page, creating a normal mapping in the guest and host PTs. The other seven physical pages inside that reservation are not mapped until the application accesses them. Although these physical pages are taken from the buddy allocator's lists, these pages are still owned by the OS and can be quickly reclaimed in case of high memory pressure.

To track the existing reservations, PTEMagnet relies on an auxiliary data structure, called Page Reservation Table (PaRT). PaRT is queried on every page fault. A look-up to PaRT succeeds if there already exists a reservation for a group of eight virtual pages that includes the faulting page. If not found in PaRT, the page fault handler takes a contiguous chunk of eight pages from the buddy allocator and stores the pointer to the base of the chunk in a newly created PaRT entry. In addition to the pointer, the entry includes an 8-bit mask that defines which pages in the group are used by the application.

Upon a page fault, the faulting virtual address is rounded to 32KB (i.e., eight 4KB pages) before performing a PaRT lookup. If the reservation exists, then a page fault can be served immediately, without a call into the buddy allocator, by creating a PTE that maps to one of the reserved pages. Thus, the extra work upon the first page fault to reserve eight pages can be largely amortized with faster page faults to the rest of the pages in a reservation. Once all the reserved pages inside a reservation are mapped, their PaRT entry can be safely deleted.

PaRT is implemented as a per-process 4-level radix tree that is indexed with a virtual address of the page fault. A leaf PaRT node corresponds to one reservation and holds a pointer to the base of a chunk of physical memory, an 8-bit mask for tracking mapped pages, and a lock. This results in a memory overhead of 17 bytes per a 32KB region which is less than 0.003%. To guarantee safety and avoid a scalability bottleneck that may appear when a large number of threads spawned by one process concurrently allocate memory, the radix tree must support fast concurrent access. Thus, to reduce lock contention and maximize inter-thread concurrency, we implement fine-grain locking with one lock per node of the PaRT radix tree.

5.3.3 Reserved Memory Reclamation

Reservations can be reclaimed in one of two ways: ① by the application, once it freed all eight pages in a reservation, by calling `free()`; or ② by the OS, when the system is under memory pressure. To avoid unnecessary complexity, the OS reclaims a reservation entirely and returns all the physical pages in the group back to the buddy allocator's free list. If an application explicitly frees all pages in the group, the last call results in the deletion of the reservation. If a PaRT entry was removed because all reserved pages had been mapped, freeing of the associated memory (if and when it happens) is performed as in the default kernel, without involving PTEMagnet.

Under memory pressure, the OS must be able to reclaim the reserved physical memory pages. Similar to the `swappiness` kernel parameter [90], we introduce a configurable threshold that, when reached, triggers a daemon process that walks through all reservations in PaRT of a randomly selected application, returning all reserved pages to the buddy allocator. The demon keeps releasing reservations until the overall memory consumption goes below the threshold. Note that when the OS has to free the reserved pages, the affected application(s) still continue to benefit from the shorter page walks to pages that have previously been allocated via PTEMagnet.

We expect no noticeable performance degradation from the PTEMagnet's reclamation mechanism, as the reclamation is a mere `free()` call to the buddy allocator. In contrast, other similar reclamation mechanisms, such as in Transparent Huge Pages or in FreeBSD, are associated with the demotion of large pages into collections of small pages. Such a demotion requires PT updates and TLB flushes, which can delay application's access to its memory and lead to performance anomalies [23], [67]. PTE-Magnet's reclamation mechanism does not change the PT content and does not lock memory pages used by the application.

5.3.4 Discussion

Fork and copy-on-write. Reservations are not copied, only individual pages. On a page fault in a child process, the reservation map of a parent can be checked to see if this page was allocated or not. If the requested page is not allocated by a parent (or other children), a page from a parent's reservation is returned to the child. This works well as the majority of pages shared between a parent and a child processes are read-only pages. Shared read-only pages don't invoke copy-on-write, stay contiguous and benefit from faster page walks, accelerated by PTEMagnet. Children processes cannot create new reservations in the parent's reservation map.

Swap and THP. If the OS chooses a reserved page for swapping or THP compaction, it triggers a reclamation of the reservation.

Security implications. PTEMagnet does not violate existing security barriers. Similarly to accesses to the guest PT, accesses to the data structure holding reservations are performed within the kernel code on behalf of the memory owner process only.

System interface for enabling PTEMagnet. It is possible to limit the set of applications for which PTEMagnet is used. By design, PTEMagnet improves the performance of big-memory applications – applications experiencing a large number of TLB misses. To conditionally enable PTEMagnet, a mechanism can be implemented through `cgroups` as follows. In a public cloud, the orchestrator (e.g., Kubernetes) usually specifies the maximum memory usage for each deployed container, by setting `memory.limit_in_bytes`. If this parameter is set, and it is above a predefined threshold, the OS can enable PTEMagnet for the target process. While evaluating PTEMagnet, we find that PTEMagnet does not cause a slow down even for applications that exhibit infrequent TLB misses and hence limited benefits of faster page walks (see Section 5.5.1). Consecutively, limiting the set of applications for which PTEMagnet is optional.

5.4 Methodology

System setup. We prototype PTEMagnet in Linux kernel v4.19 for x86-64 and AArch64 platforms. We assume public cloud deployment (as with Amazon VPC [78] or Google VPC [79]) where multiple jobs are scheduled on top of a fleet of virtual machines. We model the cloud environment by using QEMU/KVM for virtualized execution and by running multiple applications inside one virtual machine at the same time. As a metric of performance, we evaluate the execution time of an application in the presence of corunners. Table 5.2 summarizes the configuration details of our experimentation systems.

Parameter	Value/Description	
	x86-64 platform	AArch64 platform
Processor	Dual socket Intel® Xeon® E5-2630v4 (BDW) 2.4GHz 20 cores/socket, SMT	Dual socket Huawei Kunpeng 920-4826 2.6GHz 48 cores/socket, no SMT
Memory size	128GB/socket	32GB/socket
Hypervisor	QEMU 2.11.1	QEMU 5.2.50
Host OS	Ubuntu 18.04.3, Linux Kernel v4.15	Ubuntu 20.04.1, Linux Kernel v5.4
Guest OS	Ubuntu 16.04.6, Linux Kernel v4.19	
Guest configuration	20 vCPUs and 64GB RAM	32 vCPUs and 32GB RAM

Table 5.2: Parameters of x86-64 and AArch64 platforms used for the evaluation.

Our evaluation primarily focuses on small (4KB) pages, since fine-grain memory management delivers greater flexibility, better memory utilization, and better performance predictability (Section 2.1.2).

Benchmarks. We select a set of diverse applications that are representative of those run in the cloud and that exhibit significant TLB pressure. Our set of applications includes benchmarks from SPEC'17 and GPOP graph analytics framework [76]). Table 5.3 lists the benchmarks.

Corunners. We select a set of diverse applications from domains that are typically run in a public cloud such as data compression, machine learning, compilation, and others. The full list of corunners is shown in Table 5.3. The list includes benchmarks from SPEC'17, graph analytics, MLPerf and other benchmarks.

5.5 Evaluation

5.5.1 PTEMagnet's Performance Improvement

We evaluate PTEMagnet in two colocation scenarios. In the first scenario, we study a colocation with a corunner which has the highest page fault rate among all the corunners listed in Table 5.3, which is the 8-threaded `objdet` from MLPerf. We evaluate the first scenario on x86-64 and AArch64 architectures. In the second scenario, we colocate an application with a combination of all corunners listed in Table 5.3. In both

Name	Description
Benchmarks	
mcf, gcc, xz, omnetpp	SPEC'17 benchmarks (ref input)
cc, bfs, nibble, pagerank	GPOP graph analytics benchmarks [76], 16GB dataset (scaled from Twitter)
Co-runners	
objdet	MLPerf [91] object detection benchmark, SSD-MobileNet [92], COCO dataset [93]
chameleon	HTML table rendering
pyaes	Text encryption with an AES block-cipher
json_serdes	JSON serialization and de-serialization
rnn_serving	Names sequence generation (RNN, PyTorch)
gcc, xz	SPEC'17 benchmarks (ref input)

Table 5.3: Evaluated benchmarks and co-runners.

scenarios, we evaluate 8-threaded applications. To minimize performance variability in the system stemming from contention for hardware resources, in both scenarios, we pin applications' and corunners' threads to different CPU cores.

PTEMagnet in colocation with `objdet`. Figure 5.2 represents fragmentation in the host PT measured in colocation with `objdet` with and without PTEMagnet on x86-64-based system. We observe that PTEMagnet reduces fragmentation in the host PT to almost 1 for all evaluated benchmarks. The host fragmentation metric shows how many cache blocks on average hold hPTEs corresponding to gPTEs stored in one cache block. The results show that PTEMagnet prevents fragmentation in the host PT, reducing the footprint of the host PT, and enhances spatial locality across page walks, reducing their latency.

We evaluate performance improvement stemming from accelerated page walks. Figure 5.3 shows performance improvement delivered by PTEMagnet in comparison to the default Linux kernel on x86-64 and AArch64 systems. The baseline corresponds to execution in colocation with `objdet` without PTEMagnet. On x86-64, PTEMagnet increases performance by 4% on average (up to 9% in the best case on `xz`) whereas on AArch64 PTEMagnet delivers a speedup of 6% on average (10% max on `omnetpp`).

To highlight the fact that PTEMagnet is an overhead-free technique, we measure how PTEMagnet affects the performance of applications that do not experience high

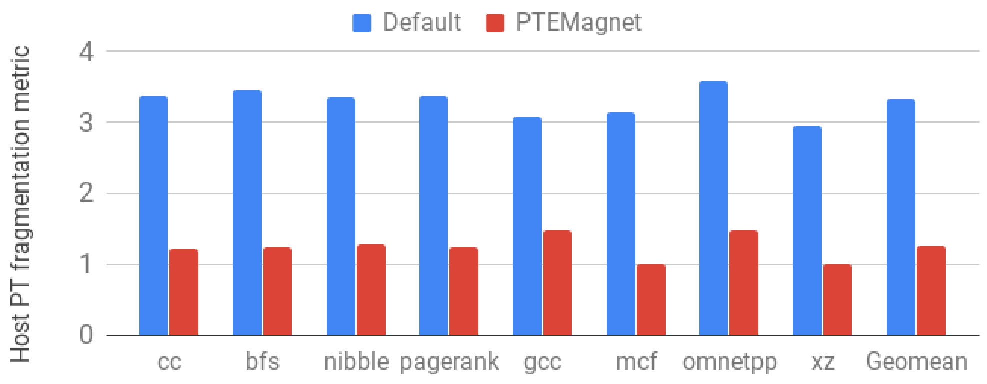


Figure 5.2: Host PT fragmentation in colocation with `objdet` on x86-64 (lower is better) [2].

TLB pressure. We evaluate PTEMagnet on all SPEC'17 Integer benchmarks. We find that on these benchmarks PTEMagnet delivers performance improvement in the range of 0-1% (not shown in Figure 5.3).

Crucially, we find that none of the applications experience any performance degradation, underscoring that PTEMagnet can be widely deployed without concern for specifics of the application or the colocation setup.

PTEMagnet in colocation with a combination of different corunners. Figure 5.4 represents improvement delivered by PTEMagnet in comparison to the default Linux kernel. On average, PTEMagnet improves performance by 3%, with a maximum gain of 5% achieved with `mcf`. A large number of corunners increases contention for the capacity of shared caches. Due to increased contention, the application's cache blocks with hPTEs have higher chances to be evicted and reduced opportunity to experience locality. Our results show that even under high cache contention, PTEMagnet is capable to speed up execution, losing just 1% of performance improvement on average, in comparison to colocation with lower cache pressure – with `objdet` only.

5.5.2 Page Walk Cycles with PTEMagnet

In this section, we evaluate a reduction in page walk cycles caused by PTEMagnet by collecting hardware performance counters data with *perf*. We also measure other hardware metrics previously studied in Section 5.2.1. We measure the metrics for `pagerank` application running in colocation with `objdet` with and without PTEMagnet. Note that in contrast to the study in Section 5.2.1, in this study, the corunner was

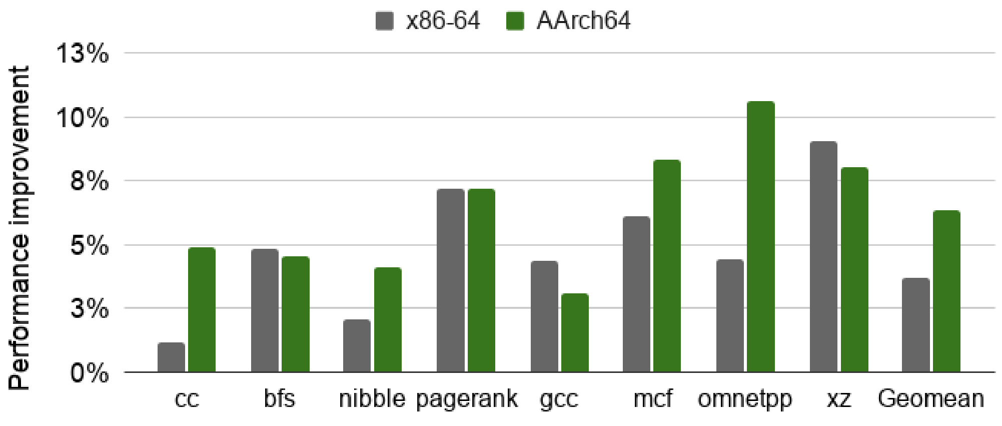


Figure 5.3: Performance improvement delivered by PTEMagnet under colocation with `objdet`.

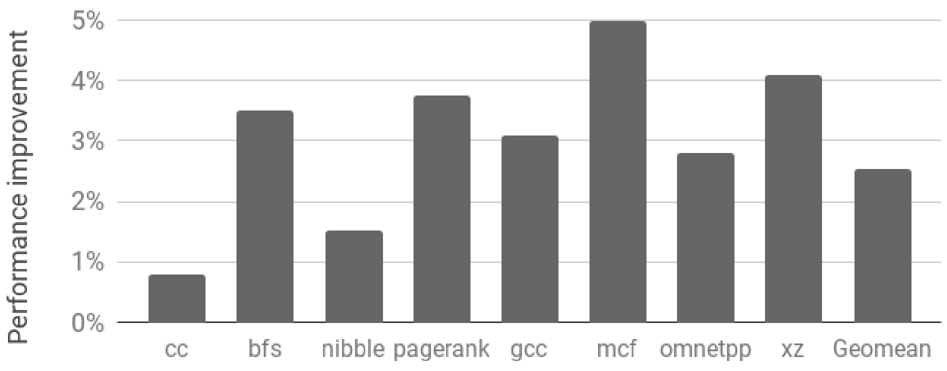


Figure 5.4: Performance improvement delivered by PTEMagnet under colocation with a combination of co-runners on x86-64 [2].

present during the entire execution of `pagerank` in both scenarios: with and without PTEMagnet. Table 5.4 lists the evaluated metrics and changes in their value delivered by PTEMagnet.

We find that PTEMagnet reduces the host page table fragmentation from 3.4 to 1.2, shortening execution time by 7%. Performance counters report that with PTEMagnet the CPU spends by 26% fewer cycles traversing the host page table, resulting in by 17% fewer cycles in page walks in total. This result is confirmed by a 13% reduction in the number of host page table accesses served by the main memory.

Metric	Change
Host page table fragmentation (defined in Sec 5.2)	-66%
Execution time	-7%
Page walk cycles	-17%
Cycles spent traversing the host page table	-26%
Guest page table accesses served by main memory	-1%
Host page table accesses served by main memory	-13%

Table 5.4: Changes in metrics of `pagerank` in colocation with `objdet` with PTEMagnet compared to the default kernel.

5.5.3 PTEMagnet's Effect on Memory Allocation Latency

In this section, we show that the reservation mechanism itself employed by PTEMagnet is overhead-free. As explained in Section 5.3, on a first page fault to a 32KB region, PTEMagnet requests 32KB from the buddy allocator, replacing subsequent page faults to the reservation group by quick accesses to PaRT. To show that the reservation-based mechanism does not cause performance degradation, we study if PTEMagnet slows down physical memory allocation.

We construct a microbenchmark that allocates a 60GB array and accesses each of its pages once to invoke the physical memory allocator. We measure the execution time of the microbenchmark with and without PTEMagnet. We observe that PTEMagnet negligibly – by 0.5% – reduces the execution time of the microbenchmark. This result can be explained by the fact that PTEMagnet makes fewer calls to the buddy allocator, replacing 7 out of 8 calls to it with quick accesses to PaRT. As a result, we conclude that PTEMagnet does not increase memory allocation latency.

5.6 Related work

Disruptive vs incremental proposals on accelerating address translation. Prior attempts at accelerating address translation take one of two directions. One calls for a disruptive overhaul of the existing radix-tree based mechanisms in both hardware and software [10], [11], [14], [15]. The other direction focuses on incremental changes to existing mechanisms [43], [48], [53]. While disruptive proposals are potentially more attractive from a performance perspective than incremental ones, disruptive approaches entail a radical re-engineering of the whole virtual memory subsystem, which presents an onerous path to adoption. In contrast, incremental techniques, requiring fewer efforts to be incorporated into existing systems, are favoured by practitioners from OS and hardware communities. Requiring only small modifications in the Linux kernel of the guest OS, PTEMagnet falls within the incremental technique category as it can be easily added to the existing systems. Moreover, in the cloud computing platforms, e.g. at AWS or Google Cloud Platform, PTEMagnet can be enabled just by cloud customers, without the involvement of cloud providers.

Incremental techniques inducing contiguity by software means. Other researches have studied incremental techniques on enforcing contiguous mappings for reducing the overhead of address translation [1], [27], [28]. Contiguity-aware (CA) paging [28] introduces a change to the OS memory allocator to promote contiguity in the physical address space. CA paging leverages contiguity to improve the performance of any hardware technique that relies on contiguous mappings, including a speculative address translation mechanism introduced by themselves. PTEMagnet is different from CA paging in two important dimensions. Firstly, CA paging, being a no pre-allocation technique, is a best-effort approach to achieve contiguity: it does not guarantee contiguity since contiguous mapping can be impossible due to allocations of other applications running on the machine. As a result, improvements of CA paging can be significantly reduced under aggressive colocation – when there are multiple memory consumers running on the same system. In contrast, PTEMagnet guarantees contiguity by eager reservation and it is insensitive to colocation. Secondly, to deliver performance improvement, CA paging requires an advanced TLB design currently not employed by modern processors, whereas PTEMagnet reduces the overhead of address translation without a need to change hardware.

Translation Ranger [27] is another incremental technique enforcing translation contiguity by software means. Translation Ranger is designed as a software helper to increase the benefits of emerging TLB designs coalescing multiple TLB entries into a single TLB entry [43], [44]. Translation Ranger creates contiguity by employing a THP-like daemon which places pages together by copying them to a contiguous region. As a consequence, Translation Ranger has disadvantages, such as high tail latency and various performance anomalies, inherent to THP-based large-page construction methods (see Section 2.1.2 for more details). In contrast to Translation Ranger, PTEMagnet creates contiguity at the moment of memory allocation and doesn't involve page copying, harmful for performance predictability. Moreover, in comparison to Translation Ranger, PTEMagnet can be straightforwardly incorporated into existing systems:

Translation Ranger relies on hardware which is non-existed in the current systems, whereas PTEMagnet does not have such a requirement.

ASAP [1] presented in Chapter 4 is another incremental proposal aiming at reducing the overhead of address translation by enforcing contiguity through software. ASAP [1] is different from CA paging and Translation Ranger in that it calls only for contiguity in the page table rather than contiguity in the whole memory. With ASAP, contiguity in the page table enables page table node prefetching which reduces page walk latency, accelerating address translation. Compared to PTEMagnet, ASAP [1] has a disadvantage of requiring a change in hardware, namely the addition of the prefetching mechanism, whereas PTEMagnet can be straightforwardly used on existing machines.

Other prior incremental techniques. There is a large amount of work addressing the overhead of address translation by reducing the number of TLB misses, including such techniques as increasing TLB reach [19], [43], [63], [64], TLB prefetch [72], and speculative translation [66]. PTEMagnet is complimentary to these techniques and can be used to reduce page walk latency on a TLB miss under virtualization.

5.7 Conclusion

In this chapter, we identified a new address translation bottleneck specific to cloud environments, where the combination of virtualization and workload colocation results in heavy memory fragmentation in the guest OS. This fragmentation increases the effective cache footprint of the host PT relative to that of the guest PT. The bloated footprint of the host PT leads to frequent caches misses during nested page walks, increasing page walk latency. We introduced PTEMagnet, which addresses this problem with a legacy-preserving software-only technique to reduce page walk latency in the cloud environments. PTEMagnet is powered by an insight that grouping hPTEs of nearby application's pages in one cache block can reduce the footprint of the host PT. Such grouping can be achieved by enhancing contiguity in guest physical memory space via a light-weight memory reservation approach. PTEMagnet requires minimal changes in the Linux memory allocator and no modifications to either user code or virtual address translation mechanisms. PTEMagnet enables a significant reduction in page walk latency, improving application performance by up to 10%.

Chapter 6

Conclusions and Future Work

As the end of Moore's law is approaching, the TLB reach fails to keep up with the dataset size growth, making TLB misses, and hence page walks, the common case and one of the main performance bottlenecks. The ubiquitous use of virtualization and application colocation increases page walk latency, amplifying the performance tax of virtual memory. In pursuit of a virtual memory solution that would address the problem of the high overhead of virtual memory under virtualization, we find that accelerating page walks should be a prime target.

We aim to reduce page walk latency by upgrading the address translation mechanism. The address translation mechanism is deeply rooted in the memory subsystem which is operated by dedicated hardware and complex software that has a lot of interdependencies with other system components. As a result, a disruptive change in the address translation mechanism would require a lot of effort for updating numerous components in a system in both software and hardware. In contrast, improving the address translation mechanism by incremental changes in a fully backward-compatible manner is a low-effort approach. Indeed, the wide adoption of the fully backward-compatible 2D page walk mechanism – even at the cost of tens of architecturally exposed memory accesses – is a great example of the vital need for full compatibility with the existing virtual memory mechanisms, and specifically with the radix-tree-based organization of the PT. Therefore, in this thesis, we aim to reduce page walk latency under virtualization without disruptive changes to the memory subsystem.

The rest of this section briefly summarizes our proposals for reducing page walk latency under virtualization and lists directions for future research in this area.

6.1 Contributions

6.1.1 Characterizing Page Walks

In Chapter 3, we characterize page walks. We find that virtualization considerably increases page walk latency, causing a significant performance overhead. We analyze sources of the high page walk latency under virtualization. We find that accesses to the deeper levels of the guest and host PTs are responsible for most of the page walk latency. Moreover, we discover that under application colocation, accesses to the host PTEs on average take much longer than accesses to the guest PTEs.

6.1.2 ASAP: Accelerating Page Walks by Prefetching Page Table Nodes

In Chapter 4 we introduce ASAP, a new approach for mitigating the high latency of address translation under virtualization. At the heart of ASAP is a lightweight technique for directly indexing individual levels of the page table radix tree. Direct indexing enables ASAP to fetch nodes from deeper levels of the page table without first accessing the preceding levels, thus lowering the page walk latency in virtualized deployments. ASAP is non-speculative and fully legacy-preserving, requiring no modifications to the existing radix-tree-based page table and TLBs.

6.1.3 PTEMagnet: Improving Caching of Page Table Nodes

In Chapter 5, we study why under application colocation, accesses to the host PTEs on average take much longer than accesses to the guest PTEs. By analyzing latencies of individual accesses during a nested page walk under virtualization and application colocation, we find that cache misses to the host PTEs are more frequent than to the guest PTEs under virtualization and application colocation. We discover that such a difference is created by the guest OS memory allocator which under application colocation causes memory fragmentation and reduces the locality of cache blocks with the host PT. We propose PTEMagnet, a new approach for reducing the high latency of address translation in cloud environments. PTEMagnet prevents memory fragmentation by using fine-grained reservation-based allocation in the guest OS. PTEMagnet is a software-only, overhead-free, and lightweight technique, that can be easily incorporated into the existing systems.

6.2 Future Work

In this section, we highlight the limitations of our proposals and describe potential future directions for the research in accelerating address translation under virtualization.

6.2.1 Tailoring the Page Table Page Allocation Policy for Shortening Page Walks

As explained in Chapter 4, ASAP reduces page walk latency under virtualization by prefetching intermediate PT nodes during a nested page walk. The key idea behind ASAP is a custom PT page allocation policy that enables direct intermediate PT node indexing that is required for prefetching. The great advantage of ASAP is that ASAP is fully compatible with the radix-tree-based PT organization. However, even though the ASAP-enabling changes to the memory subsystem are small and isolated, the PT nodes prefetching mechanism requires additional hardware that can only be added to processors of future generations. Thus, the requirement for additional hardware does not allow to use ASAP on a large fleet of already manufactured processors.

To reduce page walk latency on already manufactured processors, the ASAP's concept can be restricted to leverage hardware feature already available on the manufactured processors. For example, a partial effect of ASAP can be achieved by reducing the number of memory accesses during a nested page walk by allocating the guest PT in large pages (and notifying the host OS about it). In comparison to the default Linux policy of allocating PT pages as small pages only, allocating guest PT in large pages would result in skipping the accesses to host PTEs when translating addresses of the guest PT. A potential future work can study an ASAP-inspired policy of allocating guest PT pages in large pages and measure its effect on page walk latency and the overall performance.

6.2.2 Enhancing Locality in the Host Page Table by a Custom Physical Memory Allocation Policy

As explained in Chapter 5, PTEMagnet is designed to restore the locality of accesses to the host PTEs destroyed by the Linux memory allocator under application colocation. Without application colocation, the locality of accesses to the host PTEs is the same as the locality of accesses to the guest PTEs. PTEMagnet guarantees that the locality of accesses to the host PTEs is the same as the locality of accesses to the guest PTEs, regardless of application colocation. The locality of accesses to the guest PTEs emerges from the natural locality of the application's memory accesses: locality of accesses to the guest PTEs is high if an application accesses data from pages located nearby in the virtual address space within a short period of time. As a result, the PTEMagnet's ability to improve locality in the host PT is limited by the natural locality of the application's memory accesses, and only applications with high locality of accesses to the guest PTEs can benefit from PTEMagnet.

However, it is possible to extend PTEMagnet's approach and improve the locality of accesses to the host PTEs beyond the level of the application's locality of accesses to the guest PTEs. Such an extension of PTEMagnet is possible under the assumption that an application experiences repeatable access patterns during which it accesses distant pages in the virtual address space. In such a case, since the pages are distant in the virtual address space, their PTEs are located in different cache blocks, and hence there is no locality of accesses to PTEs (of both guest and host PTs) during the pattern execution. We assume that by knowing repeatable access patterns to distant pages, it is possible to improve the locality of accesses to host PTEs during a pattern execution. To achieve the improvement, a potential technique would require ① finding repeatable access patterns to distant pages in the virtual address space and ② placing these pages nearby in the guest physical space either by a custom memory allocator or by a page migration kernel daemon. Similar to the effect of PTEMagnet, the improvement in the locality of accesses to the host PT would reduce page walk latency and improve the application's performance under virtualization. A potential future work can evaluate the opportunity of extending PTEMagnet and analyze if modern applications experience repeatable memory access patterns to distant pages in the virtual address space.

Bibliography

[1] A. Margaritov, D. Ustiugov, E. Bugnion and B. Grot, “Prefetched address translation.,” in *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*, 2019, pp. 1023–1036.

[2] A. Margaritov, D. Ustiugov, A. Shahab and B. Grot, “PTEMagnet: Fine-grained physical memory reservation for faster page walks in public clouds,” in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 211–223.

[3] Intel, “5-level paging and 5-level EPT,” Intel, White Paper 335252-002, 2017.

[4] Linley Group, “3D XPoint fetches data in a flash,” *Microprocessor Report*, 2015.

[5] A. Bhattacharjee, “Preserving virtual memory by mitigating the address translation wall,” *IEEE Micro*, vol. 37, no. 5, pp. 6–10, 2017.

[6] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale and J. Wilkes, “CPI^2: CPU performance isolation for shared compute clusters,” in *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, 2013, pp. 379–391.

[7] I. Paul, S. Yalamanchili and L. K. John, “Performance impact of virtual machine placement in a datacenter,” in *Proceedings of the 31st International Performance Computing and Communications Conference (IPCCC)*, 2012, pp. 424–431.

[8] MarketsandMarkets. “Cloud computing market report.” (2020), [Online]. Available: *www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html.*

[9] S. Kanev, J. P. Darago, K. Hazelwood *et al.*, “Profiling a warehouse-scale computer,” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[10] I. Yaniv and D. Tsafrir, “Hash, don’t cache (the page table),” *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 337–350, 2016.

[11] D. Skarlatos, A. Kokolis, T. Xu and J. Torrellas, “Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism,” in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1093–1108.

[12] J. Shi, W. Ji, J. Zhang, Z. Gao, Y. Wang and F. Shi, “KernelGraph: Understanding the kernel in a graph,” *Information Visualization*, vol. 18, no. 3, pp. 283–296, 2019.

[13] J. Huang, M. K. Qureshi and K. Schwan, "An evolutionary study of Linux memory management for fun and profit," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[14] A. Basu, J. Gandhi, J. Chang, M. D. Hill and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 237–248, 2013.

[15] H. Alam, T. Zhang, M. Erez and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th International Symposium on Computer Architecture (MICRO)*, 2017, pp. 457–468.

[16] A. Margaritov, D. Ustiugov, E. Bugnion and B. Grot, "Virtual address translation via learned page table indexes," in *Proceedings of the Workshop on Machine Learning for Systems at NeurIPS*, 2018.

[17] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla and B. Grot, "Stretch: Balancing QoS and throughput for colocated server workloads on SMT cores," in *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 15–27.

[18] A. Shahab, M. Zhu, A. Margaritov and B. Grot, "Farewell my shared LLC! A case for private die-stacked DRAM caches for servers," in *Proceedeings of the 51st International Symposium on Microarchitecture (MICRO)*, 2018, pp. 559–572.

[19] J. H. Ryoo, N. Gulur, S. Song and L. K. John, "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, ACM New York, NY, USA, 2017, pp. 469–480.

[20] T. W. Barr, A. L. Cox and S. Rixner, "Translation caching: Skip, don't walk (the page table).," in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, 2010, pp. 48–59.

[21] E. Bugnion, J. Nieh and D. Tsafrir, *Hardware and Software Support for Virtualization*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[22] J. Araujo, R. Matos, P. Maciel, R. Matias and I. Beicker, "Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure," in *Proceedings of the Middleware Industry Track Workshop*, 2011, pp. 1–7.

[23] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach and E. Witchel, "Coordinated and efficient huge page management with Ingens," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 705–721.

[24] B. Pham, J. Veselỳ, G. H. Loh and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 1–12.

[25] J. Hu, X. Bai, S. Sha, Y. Luo, X. Wang and Z. Wang, "HUB: Hugepage ballooning in kernel-based virtual machines," in *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*, 2018.

[26] K. Kirkconnell. "Often Overlooked Linux OS Tweaks." (2016), [Online]. Available: *http://blog.couchbase.com/often-overlooked-linux-os-tweaks*.

[27] Z. Yan, D. Lustig, D. Nellans and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 698–710.

[28] C. Alverti, S. Psomadakis, V. Karakostas *et al.*, "Enhancing and exploiting contiguity for fast memory virtualization," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 515–528.

[29] D. Ustiugov, A. Daglis, J. Picorel *et al.*, "Design guidelines for high-performance SCM hierarchies," in *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*, 2018, pp. 3–16.

[30] K. Keeton, "Memory-driven computing," in *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[31] V. Karakostas, J. Gandhi, F. Ayar *et al.*, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 66–78.

[32] J. Gandhi, A. Basu, M. D. Hill and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 178–189.

[33] C. H. Park, T. Heo, J. Jeong and J. Huh, "Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations," in *In Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 444–456.

[34] D. Tang, P. Carruthers, Z. Totari and M. W. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *Proceedings of the International Conference on Dependable Systems and Networks (DNS)*, 2006, pp. 365–370.

[35] *Bad page offlining*, 2009. [Online]. Available: *www.mcelog.org/badpageofflining.html*.

[36] J. Meza, Q. Wu, S. Kumar and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 415–426.

[37] L. Zhang, B. Neely, D. Franklin, D. B. Strukov, Y. Xie and F. T. Chong, "Mellow writes: Extending lifetime in resistive memories through selective slow write backs," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 519–531.

[38] M. Zhang, L. Zhang, L. Jiang, Z. Liu and F. T. Chong, "Balancing performance and lifetime of MLC PCM by using a region retention monitor," in *Proceedings of the 23rd Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 385–396.

[39] Y. Du, M. Zhou, B. R. Childers, D. Mossé and R. G. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st Symposium on High-Performance Computer Architecture*, 2015, pp. 223–234.

[40] Intel, *Intel Itanium© architecture software developer's manual, Volume 2*, 2010. [Online]. Available: *www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-software-developer-rev-2-3-vol-2-manual.html.*

[41] IBM, *Power ISA version 2.07 B*, 2018. [Online]. Available: *http://openpowerfoundation.org/?resource_lib=ibm-power-isa-version-2-07-b.*

[42] Sun Microsystems, *UltraSPARC T2 supplement to the UltraSPARC architecture*, 2007. [Online]. Available: *http://www.oracle.com/technetwork/systems/opensparc/t2-13-ust2-uasuppl-draft-p-ext-1537760.html.*

[43] B. Pham, V. Vaidyanathan, A. Jaleel and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 258–269.

[44] B. Pham, A. Bhattacharjee, Y. Eckert and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA)*, 2014, pp. 558–567.

[45] I. Cutress. "The AMD Zen and Ryzen 7 Review: A deep dive on 1800X, 1700X and 1700." (2017), [Online]. Available: *www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700.*

[46] M.-M. Papadopoulou, X. Tong, A. Seznec and A. Moshovos, "Prediction-based superpage-friendly TLB designs.," in *Proceedings of the 21st International Symposium on High-Performance Computer Architecture*, 2015, pp. 210–222.

[47] A. Seznec, "Concurrent support of multiple page sizes on a skewed associative TLB," *IEEE Trans. Computers*, vol. 53, no. 7, pp. 924–927, 2004.

[48] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes.," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 435–448.

[49] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, 2004.

[50] "libhugetlbfs(7) - Linux man page." (2006), [Online]. Available: *http://linux.die.net/man/7/libhugetlbfs.*

[51] Khang T Nguyen. "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family." (2016), [Online]. Available: *software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.*

[52] R. Kath. “Managing virtual memory.” (1993), [Online]. Available: *http://msdn.microsoft.com/en-us/library/ms810627.aspx.*

[53] A. Arcangeli, “Transparent hugepage support,” *KVMForum*, 2010.

[54] A. Bhattacharjee, “Large-reach memory management unit caches,” in *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, 2013, pp. 383–394.

[55] Arjan van de Ven, *Linux kernel debug helper for dumping kernel page tables*, 2008. [Online]. Available: *http://github.com/torvalds/linux/blob/v4.4/arch/x86/mm/dump_pagetables.c.*

[56] M. Schwarz, *PTEditor library*, 2021. [Online]. Available: *http://github.com/misc0110/PTEditor.*

[57] S. Van Schaik, K. Razavi, B. Gras, H. Bos and C. Giuffrida, “RevAnC: A framework for reverse engineering hardware page table caches,” in *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017, p. 3.

[58] *Galois*, 2018. [Online]. Available: *http://iss.oden.utexas.edu/?p=projects/galois.*

[59] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–17.

[60] D. Schor. “Huawei expands kunpeng server CPUs, plans SMT, SVE for next gen.” (2019), [Online]. Available: *http://fuse.wikichip.org/news/2274/huawei-expands-kunpeng-server-cpus-plans-smt-sve-for-next-gen/.*

[61] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2021. [Online]. Available: *http://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.*

[62] B. Gras, K. Razavi, H. Bos and C. Giuffrida, “Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks,” in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 955–972.

[63] A. Bhattacharjee, D. Lustig and M. Martonosi, “Shared last-level TLBs for chip multiprocessors,” in *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*, 2011, pp. 62–63.

[64] D. Lustig, A. Bhattacharjee and M. Martonosi, “TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, pp. 1–38, 2013.

[65] G. B. Kandiraju and A. Sivasubramaniam, “Going the distance for TLB prefetching: An application-driven study,” in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002, pp. 195–206.

[66] T. W. Barr, A. L. Cox and S. Rixner, “SpecTLB: A mechanism for speculative address translation,” in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 307–318.

[67] J. Navarro, S. Iyer, P. Druschel and A. Cox, “Practical, transparent operating system support for superpages,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[68] M. Talluri and M. D. Hill, “Surpassing the TLB performance of superpages with less operating system support,” in *Proceedings of the 6th Conference on Architectural Support of Programming Languages and Operating Systems (ASPLOS)*, 1994, pp. 171–182.

[69] P. Kocher, J. Horn, A. Fogh *et al.*, “Spectre attacks: Exploiting speculative execution,” in *Proceedings of the 40th Symposium on Security and Privacy (SP)*, 2019, pp. 19–37.

[70] M. Lipp, M. Schwarz, D. Gruss *et al.*, “Meltdown: Reading kernel memory from user space,” in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 973–990.

[71] J. V. Bulck, M. Minkin, O. Weisse *et al.*, “Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution,” in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 991–1008.

[72] A. Bhattacharjee, “Translation-triggered prefetching,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 63–76.

[73] J. Ahn, S. Jin and J. Huh, “Revisiting hardware-assisted page walks for virtualized systems,” in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 476–487.

[74] C. A. Waldspurger, “Memory resource management in VMware ESX server,” in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002, pp. 181–194.

[75] J. Gandhi, M. D. Hill and M. M. Swift, “Agile paging: Exceeding the best of nested and shadow paging,” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 707–718.

[76] K. Lakhotia, R. Kannan, S. Pati and V. Prasanna, “GPOP: A cache and memory-efficient framework for graph processing over partitions,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 393–394.

[77] C. I. King. “Stress-ng.” (2020), [Online]. Available: *http://kernel.ubuntu.com/~cking/stress-ng*.

[78] Amazon. “AWS virtual private cloud.” (2021), [Online]. Available: *http://aws.amazon.com/vp*.

[79] Google Cloud. “Virtual private cloud.” (2021), [Online]. Available: *http://cloud.google.com/vpc*.

[80] J. Barr. “Choosing the right EC2 instance type for your application.” (2013), [Online]. Available: *http://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application*.

[81] ——, “Capacity-optimized spot instance allocation in action at Mobileye and Skyscanner.” (2020), [Online]. Available: *http://aws.amazon.com/blogs/aws/capacity-optimized-spot-instance-allocation-in-action-at-mobileye-and-skyscanner*.

[82] AWS Editorial Team. “Multi-tenant design considerations for Amazon EKS clusters.” (2020), [Online]. Available: *http://aws.amazon.com/blogs/containers/multi-tenant-design-considerations-for-amazon-eks-clusters*.

[83] Kubernetes. “Turnkey cloud solutions.” (2021), [Online]. Available: *http://kubernetes.io/docs/setup/production-environment/turnkey*.

[84] AWS. “Amazon Elastic Kubernetes Service.” (2021), [Online]. Available: *http://aws.amazon.com/ek*.

[85] Google Cloud. “Google Kubernetes Engine.” (2021), [Online]. Available: *http://cloud.google.com/kubernetes-engine*.

[86] R. Pary. “Run your Kubernetes workloads on Amazon EC2 spot instances with Amazon EKS.” (2018), [Online]. Available: *http://aws.amazon.com/blogs/compute/run-your-kubernetes-workloads-on-amazon-ec2-spot-instances-with-amazon-eks*.

[87] Amazon. “Amazon Elastic Container Service features.” (2021), [Online]. Available: *http://aws.amazon.com/ecs/features*.

[88] T. Jernigan, *Amazon ECS task placement*, 2019. [Online]. Available: *http://aws.amazon.com/blogs/compute/amazon-ecs-task-placement*.

[89] D. Singh. “Amazon ECS vs Amazon EKS: Making sense of AWS container services.” (2020), [Online]. Available: *http://aws.amazon.com/blogs/containers/amazon-ecs-vs-amazon-eks-making-sense-of-aws-container-services*.

[90] R. van Riel and P. Morreale. “Linux kernel documentation for the Sysctl files.” (2018), [Online]. Available: *http://www.kernel.org/doc/Documentation/sysctl/vm.txt*.

[91] V. J. Reddi, C. Cheng, D. Kanter *et al.*, “MLPerf inference benchmark,” in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[92] A. G. Howard, M. Zhu, B. Chen *et al.*, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[93] T.-Y. Lin, M. Maire, S. Belongie *et al.*, “Microsoft COCO: Common objects in context,” in *Proceedings of the 13th European conference on computer vision (ECCV)*, 2014, pp. 740–755.