RETHINKING COMPUTER ARCHITECTURE AND OPERATING SYSTEM ABSTRACTIONS FOR GOOD & EVIL

BY

DIMITRIOS SKARLATOS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Christopher W. Fletcher
Professor Nam Sung Kim
Professor Christos Kozyrakis, Stanford University
Dr. Vijayaraghavan (Ravi) Soundararajan, VMware
Professor Tianyin Xu

Abstract

Computing systems are undergoing a radical shift, propelled by stern security requirements and an unprecedented growth in data and users. This change has proven to be *abstraction breaking*. Current hardware and Operating System (OS) abstractions were built at a time when we had minimal security threats, scarce compute and memory resources, and limited numbers of users. These assumptions are not representative of today's computing landscape. On one hand, modern virtualization technologies have enabled the new cloud paradigms of serverless computing and microservices, which have in turn lead to the sharing of computing resources among hundreds of users. On the other hand, attacks such as Spectre and Meltdown have shown that current hardware is plagued by critical vulnerabilities. In this new era of computing, it is urgent that we question the existing abstractions of the OS and hardware layers and rethink their synergy from scratch.

This thesis takes the first steps toward answering this question, while following two central themes: (a) uncovering security vulnerabilities and building defenses at the boundary between hardware and OS, and (b) re-designing abstractions and interfaces between the two layers to improve performance and scalability. In the first theme, this thesis introduces *Microarchitectural Replay Attacks*, a new class of attacks that de-noise nearly arbitrary microarchitecture side-channels. In addition, it proposes *Jaimais Vu*, the first defense against microarchitectural replay attacks. *Jaimais Vu* uses either hardware only or compiler and OS-assisted techniques. The thesis also develops the *Draco* OS and hardware mechanisms for low-overhead protection of the system call interface by caching validated system calls and arguments.

In the second theme, this thesis proposes *Elastic Cuckoo Page Tables*, a radical rethink of virtual memory translation that enables high translation parallelism. Ongoing work aims to bring the benefits of Elastic Cuckoo Page tables to virtualized environments. This thesis also enhances the scalability of lightweight virtualization by developing the *BabelFish* mechanisms to share virtual memory translation resources across container processes. Finally, this thesis proposes the *PageForge* extensions to modern hypervisors for scalable page merging in virtualized environments.

To my mother and sister, for their unconditional love and support.

Acknowledgments

"The Pursuit of Happyness" would be the ideal way to describe my Ph.D. journey. Over the past six years I was fortunate enough to grasp my passion for research. This would not have been feasible without the support and guidance of countless people that transformed my life forever.

Undeniably, this thesis would not have been possible without the unwavering support and mentorship from my advisor Professor Josep Torrellas. I could not have asked for a better advisor. His unparalleled dedication to research was the driving force for me. During my studies, Josep provided ample freedom to pursue any research direction I was passionate about and enabled me to integrate multiple areas of research into my studies. He patiently listened to all my vague ideas and helped shape them into concrete research projects. This journey was not always easy, but he was always there. He taught me how to have a thick skin and keep going through fearlessly. More importantly, he taught me how to be polite and thoughtful and to listen carefully. Every time I needed advice no matter the day or time, Josep was available to guide me through any challenge. Over the years he tirelessly taught me everything I know about research. Working with Josep made me realize my love for research and for that I will be forever grateful.

Next, I would like to thank Professors Chris Fletcher and Tianyin Xu. My Ph.D. journey would not have been as productive and fruitful without them. Chris is the sole reason for my engagement with security research. His ability to motivate his collaborators is second to none. Despite the research roadblocks, Chris managed to inspire everyone and made research truly fun even when nothing worked. I will always remember our amazing brainstorming sessions and whiteboard craziness. I only had the chance to work with Tianyin during the last two years of my Ph.D. I would like to thank him for his endless support during those years and especially for our late-night conversations on slack about research, academia, and life. Tianyin turned system's research into pure joy and I truly hope that some of his unique mentoring skills and his approach to students will stick with me.

I would also like to thank my external thesis committee members Professor Christos Kozyrakis, Dr. Ravi Soundararajan, and Professor Nam Sung Kim. I always looked up to Christos since my undergraduate days and I was extremely happy when he accepted to serve on my committee. I would like to thank him for his advice and support on how to best embark on the academic journey and his motivation to pursue my goals without any hesitation. This thesis might not have been on cloud computing if it wasn't for Christo and Christina's work that inspired me to pursue this direction. I was fortunate to meet Ravi during my internship at VMware. He paid ceaseless attention to my work and constantly provided actionable feedback throughout my internship. I

learned a great deal from Ravi on how to operate in the workspace and how to provide useful and direct feedback. I started working with Nam Sung when I arrived at University of Illinois at Urbana-Champaign (UIUC). I learned several valuable lessons from him about how to efficiently present my work and how to combine complex circuit technologies with my research.

The i-acoma group was pivotal to the completion of this thesis. I would like to thank all my labmates: Adi, Bhargava, Mengjia, Tom, Tanmay, Yasser, Jiho, Raghav, Andy, Antonio, Azin, Apo, Serif, Neil, Houxiang, Namrata, Nam, Jovan, and Antonis. Together we shared priceless moments in the office, discussing research ideas, surviving submission deadlines, battling with reviewer 2, and enjoying group lunch. I would also like to thank Adi, Jiho, Azin, Apo, Serif, Neil, Riccardo, Namrata, Jovan, Umur, Qingrong, and Jianyan for all the projects we worked on together during my studies. I would especially like to thank Bhargava and Mengjia for making the office an extremely fun environment, for collaborating with me on several projects, and for tirelessly listening to me complain about everything. I learned a lot from them, and I look forward to collaborating (complaining) with them for years to come.

Without the amazing mentors during my undergraduate studies, this journey would not have started in the first place. Professor Dionisis Pnevmatikatos introduced me to the world of computer architecture research and meticulously mentored me during my years in Crete. Everything started with an internship at FORTH where, under the guidance of Dionisis and Professor Polyvios Pratikakis, my first ever workshop paper became reality. Professors Apostolos Dollas and Giannis Papaefstathiou were there every step of the way and were available to work with me on several projects that sparked my passion for research.

Next, I would like to thank all my friends for making this goal worth pursuing. First, my friends from Greece who gave me constant support during my studies despite the physical distance and time difference. Specifically, I would like to thank Panos, Zoi, Stathis, Mixalis, Elli, Panagiotis, Lena, Giouli, and Silena. Being away from home is difficult and our chats on "Skorpioxori" kept me going. I look forward to our next reunion and holidays together at Parasol. Especially, I would like to thank Panos for all the calls we had over the years, for always being there, and of course, for our roadtrip in California.

Second, a warm thank you to Filippos, Mitsos, Giorgos, and Panos. Since the day we met back in Crete you were there for me and for that I will be forever grateful. I would not have made it through undergrad without you. Filippos was a pillar of support during the last years of this journey with almost daily phone calls and need for speed and rocket league sessions. Mitsos, Giorgos, and Panos kept me on track with their friendship and by constantly reminding me what really matters in life. Work is just a part of it.

During my six years at Chambana, I was fortunate to forge life-lasting friendships. I would like to wholeheartedly thank Dimitris, Elena, and Soteris which I consider family. I would not have

made it through this journey without you. Dimitris and Elena always pushed me to become a better person through constant bickering and unconditional love. Whatever I say about Soteris will not be enough and perhaps I should write a book about our adventures. From roadtrips across the U.S., to hospital rooms, to weekend nights in Chambana, thank you for everything Soto. In addition, I would like to thank Dimitris and Nikos. I was fortunate to meet you during the last few years of this journey. I cherish our friendship, our nights at Legends with all the Greek community, and above all the one of a kind roadtrip to Iowa.

Alex was perhaps the first person I met at UIUC. Although he had to leave soon enough, my first year in Chambana would not have been half as fun without him and Tercel. 610 will always be the house of unbelievable memories with the best roommates anyone could ever ask for. I would not have survived my qual without Dimitri, Soto, and Erman. I will fondly remember our roadtrip during Christmas 2015, the first time away from home. I will also never forget the day that Apo almost burned down the house trying to make popcorn. Special thanks go to Thiago and Luis. We all started together back in 2014 and while Luis was smart enough to escape early, Thiago stayed until the very end. I would most likely have failed the third compilers project if it weren't for Thiago. More importantly, I would have failed to appreciate this journey as early as I did if it weren't for both of them.

This journey would not have been as exciting or even possible without Giusy. Thank you for being by my side, for your friendship, and for your love. Thank you for pushing me to become a better person. You tirelessly supported me and cheered me on every step of the way; from my quals, to my first paper submission, to my academic job search. Together, we've been through the freezing cold and blazing hot days of Chambana. Thank you for travelling with me across the US and Europe, for climbing the Great Smoky Mountains with me, for going camping together at the Indiana Dunes, and for being my guide in Italy. Thank you for pushing me to dance, even though I still have no idea how to. It was pure luck that led me to you and gave me the opportunity to experience all these years with you. Thank you for making Chambana a place to call Home.

Last but certainly not least, I would like to thank my mother Efi and my sister Aspa. The words thank you do not seem enough for the pure and unconditional love and support. They were there for me without question no matter the time or physical distance. Thank you for believing in me every step of the way. Thank you for showing me true support all these years and more importantly for giving me the chance and drive to pursue my dreams. To say I would not have made it without you is an understatement. I am so lucky to have you in my life.

Table of Contents

Chapter 1: Thesis Overview

Cloud computing is undergoing a major change, propelled by stern security requirements and an unprecedented growth in data and users. Modern virtualization technologies have enabled the new cloud paradigms of serverless computing and microservices, which in turn have lead to the sharing of computing resources among hundreds of users. Concurrently, attacks such as Spectre and Meltdown have shown that current hardware is plagued by critical vulnerabilities, allowing the circumvention of existing security barriers.

The vision of this thesis is to understand the fundamental interactions between the Operating System (OS) and hardware in this new era of cloud computing, break down unnecessary walls separating the OS and hardware layers, and rebuild abstractions and mechanisms to improve security, performance, and scalability. The contributions of this thesis can be classified into security, performance, and scalability, as we describe next.

1.1 SECURITY

Microarchitectural Replay Attacks. It is now well understood that modern processors can leak secrets over microarchitectural side-channels. Such channels are ubiquitous, as they include a myriad of structures such as caches, branch predictors, and various ports. Yet, most channels are very noisy, which makes exploiting them challenging.

My work shows how malicious use of existing hardware-OS abstractions enables attackers to get around this challenge. Specifically, I propose *Microarchitectural Replay Attacks* (MRA) [1], a new family of attacks that empowers an attacker to de-noise nearly any microarchitectural side-channel—even if the victim code is executed *only once*. The key observation is that pipeline squashes caused by page faults, branches, memory consistency violations, and other events make it possible for a dynamic instruction to squash and re-execute multiple times. By forcing these replays, an attacker can repeatedly measure the execution characteristics of an instruction. This observation can be leveraged by attackers to mount a large number of new privacy- and integrity-breaking attacks.

To demonstrate MRAs, I built the *MicroScope* framework, and mounted several proof-of-concept attacks in the Intel Software Guard Extensions (SGX) setting. I showed how MicroScope is capable of de-noising an extremely noisy microarchitectural side-channel—port contention caused by two divisions—in a single run of the victim instructions. MicroScope enables the attacker to cause an arbitrary number of squashes and replays of an instruction through page faults. The replays repeat until the signal-to-noise ratio is enough to leak potentially any secret. All the while, the

victim has logically run only once.

MicroScope's novel ability to provide a controllable window into speculative execution has wide applicability. For instance, it can be used for side-channel attack exploration, since it nearly guarantees the success of the attack. It can also be applied to examine the effects of speculative attacks, such as detecting which instructions are speculatively reachable from a given branch. For example, in a project we are currently pursuing, MicroScope is used to bound the number of fences that are inserted to defend against speculative execution attacks. Beyond security, the ability to capture and re-execute a set of speculative instructions is an unparalleled primitive for software development and debugging. Replaying instructions deterministically may provide a way to debug hard-to-reproduce software bugs such as data races.

MicroScope is already having an impact. It is a central work in the Intel Strategic Research Alliance (ISRA) Center on Computer Security at the University of Illinois. Further, MicroScope is the foundation of a proposal funded by Google to defend against speculative execution attacks. I made the framework available publicly on github, and it has more than two thousand hits from 28 countries since July 2019.

Defending Against MRAs. I propose *Jaimais Vu*, the first mechanism designed to thwart MRAs [2]. The mechanism works by first recording the victim instructions *V* that are squashed. Then, as each V instruction is re-inserted into the pipeline, Jaimais Vu automatically places a fence before it to prevent the attacker from squashing it again.

A highly secure defense against MRAs would keep a fine-grain record of all the victim dynamic instructions that were squashed. In reality, such a scheme is not practical due to storage requirements and the difficulty of identifying the same dynamic instruction.

To overcome this limitation, Jaimais Vu proposes three classes of schemes that discard the record of victim instructions early. The schemes differ in when and how they discard the state. They effectively provide different trade-offs between execution overhead, security, and implementation complexity. The main Jaimais Vu scheme combines hardware, operating system, and compiler support to build a robust defense with low performance overhead. This scheme leverages a program analysis pass that identifies and annotates "epochs" in the application. During execution, the hardware leverages these annotations to track "execution localities" and discard victim state when the execution moves to another locality.

Operating System Protection. Protecting the OS is a significant concern, given its capabilities and shared nature. A popular technique to protect the OS is *system call checking*. The idea is to dynamically check the system calls invoked by the program and the arguments used, and only allow the execution to proceed if they belong to a safe set. This technique is implemented by

adding code at the OS entry point of a system call. The code compares the incoming system call against a list of allowed system calls and argument set values, and either lets the execution continue or not. Linux's Seccomp is the most widely-used implementation of such a technique. Unfortunately, checking system calls and arguments incurs substantial overhead.

To solve this problem, I propose *Draco* [3], a hardware- and OS-based mechanism to protect system calls. The idea is to dynamically *cache* system calls and arguments after they have been checked and validated. Subsequent system calls first check the cache and, on a hit, are immediately considered validated. The insight behind Draco is that the patterns of system calls in real-world applications have *locality*—i.e., the same system calls are repeatedly invoked, with the same sets of argument values.

I designed both a software version of Draco in the Linux kernel and a hardware version. The software version uses a novel multilevel software cache based on cuckoo hashing. My system is substantially faster than the current state-of-art Seccomp mechanism. Still, it incurs a significant overhead compared to an insecure system that performs no checks.

To overcome this limitation, I propose hardware that, at runtime, both checks system calls and arguments, and stores validated system calls in a hardware cache in the pipeline for later re-use. Specifically, I introduce the System Call Lookaside Buffer (SLB) to keep recently-validated system calls, and the System Call Target Buffer (STB) to preload the SLB in advance. Such preload during speculative execution is performed in a safe manner. The result is that the hardware-based Draco design performs system call checks with practically no overhead over an insecure processor.

Draco spurred a collaboration with IBM and RedHat in an effort to upstream the proposed software solution in the Linux Kernel and further investigate the security of containerized environments.

1.2 PERFORMANCE

Elastic Cuckoo Page Tables. The current implementation of the page tables, known as Radix page tables, organizes the translations in a multi-level, forward-mapped radix tree. Current processors use four-level trees, and a fifth level will appear in next-generation processors. Radix page tables have been highly successful for several decades, but they were not designed for the current use of terabytes and even petabytes of memory, or the emerging wide variety of memory-intensive applications. I argue that radix page tables are bottlenecked at their core by a translation process that requires searching the levels of the radix tree sequentially. Such fundamental bottleneck cannot be removed by the largely incremental improvements performed up until today: large and multi-level TLBs, huge page support, and MMU caches for page table walks.

My research introduces a radical rethink of the virtual memory address translation that enables high translation parallelism. The scheme, called *Elastic Cuckoo Page Tables* [4], introduces a parallel hashed page table design. It is enabled by Elastic Cuckoo Hashing, a novel extension of d-ary cuckoo hashing that halves the number of memory accesses required to look-up an element during gradual hash table resizing. This key algorithmic advancement transforms the sequential pointer chasing of the radix page tables into fully parallel lookups – harvesting for the first time the benefits of memory-level parallelism for address translation. Elastic Cuckoo Page Tables also introduce a hardware Cuckoo Walk Cache that effectively prunes the search space of a translation look-up.

The resulting redesign delivers substantial performance benefits. Virtual memory translation overhead is reduced, on average, by 40% over conventional systems. Effectively, Elastic Cuckoo Page Tables pave the way towards eliminating the virtual memory overhead. Furthermore, Elastic Cuckoo Page Tables can be applied to virtualized environments, and address one of the longest standing performance overheads of machine virtualization, namely nested address translation.

My proposal of Elastic Cuckoo hashing opens up new research opportunities in multiple areas. One is redesigning the TLB based on Elastic Cuckoo hashing to dramatically increase TLB associativity and enable prefetching of translations directly into the TLB. Another avenue is redesigning other hardware caching structures to enable extremely fine grain partitioning, a key requirement for resource isolation in multi-tenant environments. Finally, at the software level, Elastic Cuckoo hashing has wide applicability in hashing structures, especially in the OS and in database applications.

1.3 SCALABILITY

Extreme Container Concurrency. The emergence of lightweight virtualization technologies such as containers has caused a major change in cloud computing. A container packages an application and all of its dependencies, libraries, and configurations, and isolates it from the system it runs on. Importantly, containers share the host OS and eliminate the need for a guest OS required by virtual machines. Such lightweight virtualization enables high user application concurrency, through microservices and serverless computing.

My research has identified that current oversubscribed environments perform poorly due to limitations in state-of-the-art OS and hardware designs. Mechanisms such as virtual memory and process management have hit a scalability wall. They suffer from redundant kernel work during page table and process management, TLB thrashing, and various other overheads.

To remedy these limitations, I introduce *BabelFish* [5], a novel TLB and page table design

that enables containers to share virtual memory translations. BabelFish introduces the concept of Container Context Identifiers (CCID) in the OS. Containers in the same CCID group share TLB and page table entries. TLB entry sharing eliminates redundant TLB entry flushes and acts as an effective translation prefetching mechanism. Page table entry sharing eliminates redundant page faults and enables the sharing of translation entries in the cache hierarchy. Together, these mechanisms significantly reduce the cost of process management when thousands of containers share hardware resources in oversubscribed settings. They also significantly improve container bring-up time, a critical requirement of serverless computing, by leveraging the warmed-up state of previous containers.

BabelFish paves the way for container-aware hardware and OS structures. The vision is of an environment where containers are a first-class entity across the stack.

Content Deduplication for Virtualization. In virtualized environments that require a guest OS, content replication is a major scalability bottleneck. Hypervisors of lightweight micro Virtual Machines (VMs) and contemporary VMs aim to reduce content replication by performing content-aware page merging or de-duplication. This is a costly procedure. It requires the hypervisor to search the physical memory and identify pages with identical contents and merge them. To reduce the overhead of this procedure, numerous optimizations have been adopted—e.g., use a portion of the contents of a page to create a hash key that will prune the number of other pages that this page needs to be compared to. Still, in modern systems with hundreds of GBs of main memory, the overhead of this "datacenter tax" is substantial.

To eliminate this problem, I introduce *PageForge* [6], a near-memory hardware module for scalable content-aware page merging in virtualized environments. PageForge resides in the memory controller, and automatically compares and merges pages without polluting caches or consuming core cycles. PageForge is exposed to the hypervisor, which guides the search in batches.

A key contribution of *PageForge* is the novel use of memory Error Correcting Codes (ECC) to characterize the contents of a page. Specifically, PageForge uses ECC to generate highly-accurate per-page hash keys inexpensively. Overall, PageForge enables the deployment of twice as many VMs as state-of-the-art systems for the same physical memory consumption—without the performance overhead of software-only solutions.

Chapter 2: Microarchitectural Replay Attacks

2.1 INTRODUCTION

The past several years have seen a surge of interest in hardware-based Trusted Execution Environments (TEEs) and, in particular, the notion of *enclave programming* [7, 8, 9]. In enclave programming, embodied commercially in Intel's Software Guard Extensions (SGX) [8, 9, 10, 11], outsourced software is guaranteed virtual-memory isolation from supervisor software—i.e., the OS, hypervisor, and firmware. This support reduces the trusted computing base to the processor and the sensitive outsourced application. Since SGX's announcement over five years ago, there have been major efforts in the community to map programs to enclaves, and to SGX in particular (e.g., [12, 13, 14, 15, 16, 17, 18, 19, 20, 21]).

Despite its promise to improve security in malicious environments, however, SGX has recently been under a barrage of microarchitectural side channel attacks. Such attacks allow co-resident software-based attackers to learn a victim process' secrets by monitoring how that victim uses system and hardware resources—e.g., the cache [22, 23, 24, 25, 26] or branch predictor [27, 28], among other structures [29, 30, 31, 32, 33]. Some recent work has shown how SGX's design actually exacerbates these attacks. In particular, since the supervisor-level SGX adversary controls victim scheduling and demand paging, it can exert precise control on the victim and its environment [34, 35, 36, 37, 38].

Yet, not all hope is lost. There is scant literature on how much secret information the adversary can exfiltrate if the victim application only runs *once*, or for that matter, if the instructions forming the side channel only execute once (i.e., not in a loop). Even in the SGX setting, many modern, fine-grain side channels—e.g., 4K aliasing [31], cache banking [22], and execution unit usage [29, 32]—introduce significant noise, forcing the adversary to run the victim many (potentially hundreds of) times to reliably exfiltrate secrets. Even for less noisy channels, such as the cache, SGX adversaries still often need more than one trace to reliably extract secrets [37]. This is good news for defenders. It is reasonable to expect that many outsourced applications, e.g., filing tax returns or performing tasks in personalized medicine, will only be run once per input. Further, since SGX can defend against conventional replay attacks using a combination of secure channels, attestation, and non-volatile counters [39], users have assurance that applications meant to run once will only run once.

2.1.1 This Chapter

Despite the assurances made in the previous paragraph, this chapter introduces *Microarchitectural Replay Attacks*, which enable the SGX adversary to denoise (nearly) any microarchitectural side channel inside of an SGX enclave, even if the victim application is *only run once*. The key observation is that a fundamental aspect to SGX's design enables an adversary to replay (nearly) arbitrary victim code, without needing to restart the victim after each replay, thereby bypassing SGX's replay defense mechanisms.

At a high level, the attack works as follows. In SGX, the adversary manages demand paging. We refer to a load that will result in a page fault as a *replay handle*—e.g., one whose data page has the Present bit cleared. In the time between when the victim issues a replay handle and the page fault is triggered, i.e., after the page table walk concludes, the processor will have issued instructions that are younger than the replay handle in program order. Once the page fault is signaled, the adversary can opt to *keep the present bit cleared*. In that case, due to precise exception handling and in-order commit, the victim will resume execution at the replay handle and the process will repeat a potentially unbounded number of times.

The adversary can use this sequence of actions to denoise microarchitectural side channels by searching for replay handles that occur before sensitive instructions or sensitive sequences of instructions. Importantly, the SGX threat model gives the adversary sufficient control to carry out these tasks. For example, the adversary can arrange for a load to cause a page fault if it knows the load address, and can even control the page walk time by priming the cache with select page table entries. Each replay provides the adversary with a noisy sample. By replaying an appropriate number of times, the adversary can disambiguate the secret from the noise.

We design and implement *MicroScope*, a framework for conducting microarchitectural replay attacks, and demonstrate our attacks on real hardware.[1] Our main result is that MicroScope can be used to reliably reveal execution unit port contention, i.e., similar to the PortSmash covert channel [32], even if the victim is only run once. In particular, with SMT enabled, our attack can detect the presence or absence of *as few as two divide instructions* in the victim. With further tuning, we believe we will be able to reliably detect one divide instruction. Such an attack could be used to detect subnormal input to *individual floating-point* instructions [29], or infer branch directions in an enclave despite countermeasures to flush the branch predictor at the enclave boundary [40]. Beyond port contention, we also show how our attack can be used to single-step and perform zero-noise cache-based side channels in AES, allowing an adversary to construct a denoised trace given a single run of that application.

[1] The name MicroScope comes from the attack's ability to peer inside nearly any microarchitectural side channel.

Contributions. This chapter makes the following contributions.

- We introduce microarchitectural replay attacks, whereby an SGX adversary can denoise nearly arbitrary microarchitectual side channels by causing the victim to replay on a page-faulting instruction.
- We design and implement a kernel module called *MicroScope*, which can be used to perform microarchitectural replay attacks in an automated fashion, given attacker-specified replay handles.
- We demonstrate that MicroScope is able denoise notoriously noisy side channels. In particular, our attack is able to detect the presence or absence of two divide instructions. For completeness, we also show single-stepping and denoising cache-based attacks on AES.
- We discuss the broader implications of microarchitectural replay attacks, and discuss different attack vectors beyond denoising microarchitectural side channels with page faults.

The source code for the MicroScope framework is available at
https://github.com/dskarlatos/MicroScope.

2.2 BACKGROUND

2.2.1 Virtual Memory Management in x86

A conventional TLB organization is shown in Figure 2.1. Each entry contains a Valid bit, the Virtual Page Number (VPN), the Physical Page Number (PPN), a set of flags, and the Process Context ID (PCID). The latter is unique to each process. The flags stored in a TLB entry usually include the Read/Write permission bit, the User bit that defines the privilege level required to use the entry, and other bits. The TLB is indexed using a subset of the virtual address bits. A hit is declared when the VPN and the PCID match the values stored in a TLB entry. Intel processors often deploy separate instruction and data L1 TLBs and a unified L2 TLB.

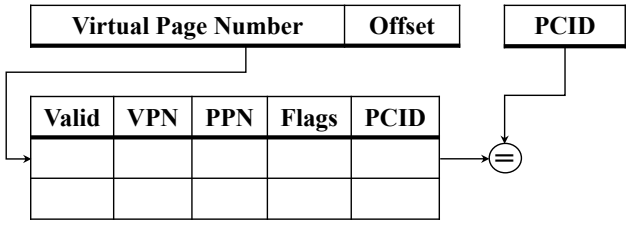


Figure 2.1: Conventional TLB organization.

If an access misses on both L1 and L2 TLBs, a page table walk is initiated to locate the missing translation. The hardware Memory Management Unit (MMU) performs this process. Figure 2.2 shows the page table walk for address *A*. The hardware first reads a physical address from the CR3 control register. This address corresponds to the process-private Page Global Directory (PGD). The page walker hardware adds the 40-bit CR3 register to bits 47-39 of the requested virtual address. The result is the physical address of the relevant pgd_t entry. Then, the page walker issues a request to the memory hierarchy to obtain the pgd_t. This memory request either hits in the data caches or is sent to main memory. The contents of pgd_t is the address of the next page table level, called Page Upper Directory (PUD). The same process is repeated for all the page table levels. Eventually, the page walker fetches the leaf pte_t entry that provides the PPN and flags. The hardware stores such informantion in the TLB.

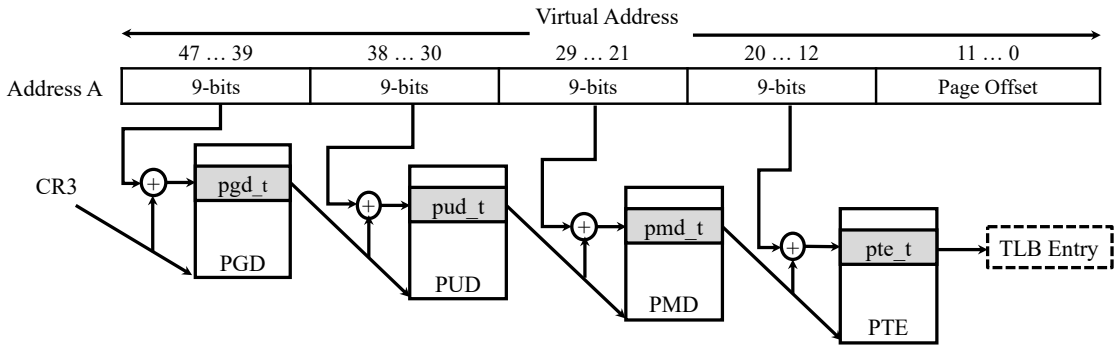


Figure 2.2: Page table walk.

Modern MMUs have a translation cache called the Page Walk Cache (PWC) that stores recent page table entries of the three upper levels. This can potentially reduce the number of memory accesses required to fetch a translation.

A pte_t entry includes the present bit. If the bit is cleared, then the translation process fails and a page fault exception is raised. The OS is then invoked to handle it. There are two types of page faults: major and minor. A major one occurs when the page for one of these physical addresses requested during the walk is not in memory. In this case, the OS fetches the page from disk into memory and resumes the translation. A minor page fault occurs when the page is in memory, but the corresponding entry in the tables says that the page is not present in memory. In this case, the OS simply marks the entry as present, and resumes the translation. This happens, for example, when multiple processes share the same physical page. Even though the physical page is present in memory, a new process incurs a minor page fault on its first access to the page.

After the OS services the page fault and updates the pte_t entry, control is yielded back to the process. Then, the memory request that caused the page fault is re-issued by the core. Once again,

the request will miss in the TLB and initiate a page walk. At the end of the page walk, the updated pte_t will be stored in the TLB.

The OS is responsible for maintaining TLB coherence. This is done by flushing potentially-stale entries from the TLB. The INVLPG instruction [41] allows the OS to selectively flush a single TLB entry. When the OS needs to update a page table entry, it locates the leaf page table entry by performing a page walk following the same steps as the hardware page walker. Updating the page table causes the corresponding TLB entry to become stale. Consequently, the OS also invalidates the TLB entry before yielding control back to the process.

2.2.2 Out-of-Order Execution

Dynamically-scheduled processors execute instructions in parallel and out of program order to improve performance [42]. Instructions are fetched and enter the scheduling system in program order. However, they perform their operations and produce their results possibly out of program order. Finally, they retire—i.e., make their operation externally visible by irrevocably modifying the architected system state—in program order. In-order retirement is implemented by queueing instructions in program order in a reorder buffer (ROB) [43], and removing a completed instruction from the ROB only once it reaches the ROB head, i.e., after all prior instructions have retired.

Relevant to this chapter, out-of-order machines continue execution during a TLB miss and page walk. When a TLB miss occurs, the access causing the miss queues a hardware page walk. The processor continues fetching and executing younger instructions, potentially filling up the ROB to capacity. If a page fault is detected, before it can be serviced, the page-faulting instruction has to reach the head of the ROB. Then, all the instructions younger than it are squashed. After the page fault is serviced, the program restarts at the page-faulting instruction.

2.2.3 Shielded Execution via Enclaves

Secure enclaves [7], such as Intel's Software Guard Extensions (SGX) [8, 9, 11], are reverse sandboxes that allow sensitive user-level code to run securely on a platform alongside an untrusted supervisor (i.e., an OS and/or hypervisor).

Relative to earlier TEEs such as Intel's TPM+TXT [44] and ARM TrustZone [45], a major appeal in enclave-based TEEs is that they are compatible with mostly unmodified legacy user-space software, and expose a similar process-OS interface to the supervisor as a normal user-level process. To run code in enclaves, the user writes enclave code and declares entry and exit points into that code, which may have arguments and return values. User-level code can jump into the enclave at one of the pre-defined entry points. This is similar to context switching into a new

Spatial	Coarse Grain	Fine Grain	
Temporal	—	Low Resolution	Medium /High Resolution
No Noise	Controlled channel [36] Leaky Cauldron [35]		MicroScope (this work)
With Noise	TLBleed [33] TLB contention [48] DRAMA [30]	SGX Prime+Probe [46] Cache Bleed [22] PortSmash [32] MemJam [31] FPU subnormal [29] Exec. unit contention [51, 52] BTB collision [53] Grand Exposure [34] Leaky Cauldron [35] BTB contention [27, 54]	Cache Games [47] CacheZoom [37] Hahnel et al. [49] SGX-Step [50]

Table 2.1: Characterization of side channel attacks on Intel SGX.

hardware context from the OS point of view. While the enclave code is running, the OS performs demand paging on behalf of the enclave context as if it were a normal process.

Enclave security is broken into attestation at bootup and privacy/integrity guarantees at runtime [7]. The runtime protections give enclave code access to a dedicated region of virtual memory which cannot be read or written except by that enclave code. Intel SGX implements these memory protections using virtual memory isolation for on-chip data and cryptographic mechanisms for off-chip data [8, 10]. For ease of use and information passing, SGX's design also allows enclave code to access user-level memory, owned by the host process, outside of the private enclave memory region.

For MicroScope to attack an enclave-based TEE, the only requirement is that the OS handles page faults during enclave execution, when trying to access either private enclave pages or insecure user-level pages. Intel SGX uses the OS for both of these cases. When a page fault occurs during enclave execution in SGX, the enclave signals an AEX (asynchronous exit), and the OS receives the VPN of the faulting page. To service the fault, the OS has complete control over the translation pages (PGD, PUD, etc.). If the faulting page is in the enclave's private memory region, additional checks are performed when the OS loads the page, e.g., to make sure it corresponds to the correct VPN [11]. MicroScope does not rely on the OS changing page mappings maliciously, and thus is not impacted by these defenses. If loading a new page requires displacing another page, the OS is responsible for TLB invalidations.

2.2.4 Side Channel Attacks

While enclave-based TEEs provide strong memory isolation mechanisms, they do not explicitly mitigate microarchitectural side channel attacks. Here, we review known side channel attacks that can apply to enclaves in Intel SGX. These attacks differ in their spatial granularity, temporal resolution, and noise level. We classify these attacks according to their capabilities in Table 2.1.

We classify an attack as providing fine-grain spatial granularity if the attack can be used to monitor victim access patterns at the granularity of cache lines or finer. We classify an attack as providing coarse-grain spatial granularity if it can only observe victim access patterns at coarser granularity, such as pages.

Coarse spatial granularity. Xu et al. [36] proposed controlled side channels to observe a victim's page-level access patterns by monitoring its page faults. Further, Wang et al. [35] proposed several new attack vectors, called Sneaky Page Monitoring (SPM). Instead of leveraging page faults to monitor the accesses that trigger many AEXs, SPM monitors the Access and Dirty bits in the page tables. Both attacks target page tables, and can only achieve page-level granularity, i.e., 4KB. In terms of noise, these attacks can construct noiseless channels, since the OS can manipulate the status of pages and can observe every page access.

Gras et al. [33] and Hund et al. [48] proposed side channel attacks targeting TLB states. They create contention on the L1 DTLB and L2 TLB, which are shared across logical cores in an SMT core, to recover secret keys in cryptography algorithms and defeat ASLR. Similar to page table attacks, they can only achieve page-level granularity. Moreover, these two attacks suffer medium noise due to the races between attacker and victim TLB accesses. DRAMA [30] is another coarse-grain side channel attack that exploits DRAM row buffer reuse and contention. It can provide a granularity equal to the row buffer size (e.g., 2KB or 4KB).

Fine spatial granularity. There have been a number of works that exploit SGX to create fine spatial granularity side channel attacks that target the cache states or execution units (see Table 2.1). However, they all have sources of noise. Therefore, the victim must be run multiple times to obtain multiple traces, and intelligent post-processing techniques are required to minimize attack errors.

We further classify fine spatial granularity attacks according to the level of temporal resolution that they can achieve. We consider an attack to have high temporal resolution if it is able to monitor the execution of every single instruction. These attacks almost always require the attacker to have the ability to single-step the victim program. We define an attack to have low temporal resolution if it is only able to monitor the aggregated effects of multiple instructions.

Low temporal resolution. Several cache attacks on SGX [34, 46] use the Prime+Probe attack strategy and the PMU (performance monitoring unit) to observe a victim's access patterns at the cache line level. Leaky Cauldron [35] proposed combining cache attacks and DRAMA attacks

to achieve fine-grain spatial granularity. These attacks cannot attain high resolution, since the attacker does not have a reliable way to synchronize with the victim, and the prime and probe steps generally take multiple hundreds of cycles. Moreover, these attacks suffer from high noise, due to cache pollution and coarse-grain PMU statistics. Generally, they require hundreds of traces to get modestly reliable results—e.g., 300 traces in the SGX Software Grand Exposure attack [34].

CacheBleed [22] and MemJam [31] can distinguish a victim's access patterns at even finer spatial granularity, i.e., sub-cache line granularity. Specifically, CacheBleed exploits L1 cache bank contention, while MemJam exploits false aliasing between load and store addresses from two threads in two different SMT contexts. However, in these attacks, the attacker analyzes the bank contention or load-store forwarding effects by measuring the total execution time of the victim. Thus, these attacks have low temporal resolution, as such information can only be used to analyze the accumulated effects of many data accesses. Moreover, such attacks are high noise, and require thousands of traces or thousands of events per trace.

There are several attacks that exploit contention on execution units [32, 51, 52], including through subnormal floating-point numbers [29], and collisions and contention on the BTB (branch target buffer) [27, 53, 54]. As they exploit contention in the system, they have similar challenges as CacheBleed. Even though these attacks can achieve fine spatial granularity, they have low temporal resolution and suffer from high noise.

Medium/high temporal resolution. Very few attacks can achieve both fine spatial granularity and high temporal resolution. Cache Games [47] exploits a vulnerability in the Completely Fair Scheduler (CFS) of Linux to slow victim execution, and achieve high temporal resolution. CacheZoom [37] and Hahnel et al. [49] and SGX-Step [50] use high-resolution timer interrupts to frequently stop the victim process, at the granularity of a few memory accesses, and collect L1 access information using Prime+Probe. Although these techniques encounter relatively low noise, they still require multiple runs of the application to denoise the exfiltrated information.

In summary, none of the prior works can simultaneously achieve fine spatial granularity, high temporal resolution, and no noise. We propose MicroScope to boost the effectiveness of almost all of the above attacks by de-noising them while, importantly, requiring only one run of the victim application. MicroScope is sufficiently general to be applicable to both cache attacks and contention-based attacks on various hardware components, such as execution units [29], cache banks [22], and load-store units [31].

2.3 THREAT MODEL

We adopt a standard threat model used when evaluating Intel SGX [17, 18, 34, 35, 37, 55, 56, 57, 58], namely, a victim program running within an SGX enclave alongside malicious supervisor software (i.e., the OS or a hypervisor). This gives the adversary complete control over the platform, except for the ability to directly introspect or tamper enclave private memory as described in Section 2.2.3. The adversary's goal is to break privacy, and learn as much about the secret enclave data as possible. For this purpose, the adversary may monitor any microarchitectural side channel (e.g., those in Section 2.2.4) while the enclave runs.

We restrict the adversary to run victim enclave code only one time per sensitive input. This follows the intent of many applications, such as tax filings and personalized health care. The victim can defend against the adversary replaying the entire enclave code by using a combination of secure channels and SGX attestation mechanisms, or through more advanced techniques [39].

Integrity of computation and physical side channels. Our main presentation is focused on breaking privacy over microarchitectural (digital) side channels. While we do not focus on program integrity, or physical attacks such as power/EM [59, 60], we discuss how microarchitectural replay attacks can be extended to these threat models in Section 2.7.

2.4 THE MICROSCOPE ATTACK

MicroScope is based on the key observation that modern hardware allows recently executed, but not retired, instructions to be rolled back and replayed if certain conditions are met. This behavior can be easily exploited by an untrusted OS to denoise side channels.

2.4.1 Overview

A MicroScope attack has three actors: Replayer, Victim, and Monitor. The *Replayer* is a malicious OS or hypervisor that is responsible for page table management. The *Victim* is an application process that executes some secret code that we wish to exfiltrate. The *Monitor* is a process that performs auxiliary operations, such as causing contention and monitoring shared resources.

Attack Setup: The Replay Handle

MicroScope is enabled by what we call a *Replay Handle*. A replay handle can be any memory access instruction that occurs shortly before a sensitive instruction in program order, and that

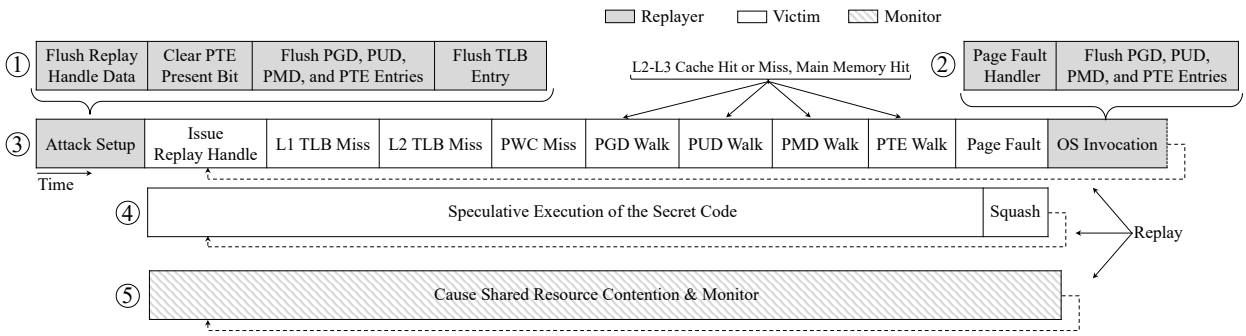


Figure 2.3: Timeline of a MicroScope attack. The *Replayer* is an untrusted OS or hypervisor process that forces the *Victim* code to replay, enabling the *Monitor* to denoise and extract the secret information.

satisfies two conditions. First, it accesses data from a different page than the sensitive instruction. Second, the sensitive instruction is not data dependent on the replay handle. Programs have many potential replay handles, including accesses to the program stack or heap, or memory access instructions that are unrelated to the sensitive instruction.

In MicroScope, the OS forces the replay handle to perform a page walk and incur a minor page fault. In the meantime, instructions that are younger than the replay handle, such as the sensitive instruction, can execute. More precisely, they can be inserted in the ROB and execute until the page fault is identified and the replay handle is at the head of the ROB, or until the ROB is full. Of course, instructions that are dependent on the replay handle do not execute.

Figure 2.3 shows the timeline of the interleaved execution of the Replayer, Victim, and Monitor. To initiate an attack, the adversary first identifies a replay handle close to the sensitive instruction. The adversary then needs to know the approximate time at which the replay handle will be executed, e.g., by single-stepping the Victim at page-fault [36] or close-to-instruction [37] granularity. The Replayer then pauses the Victim program before this point, and sets up the attack that triggers a page fault on the replay handle.

The Replayer sets up the attack by locating the page table entries required for virtual-to-physical translation of the replay handle—i.e., its pgd_t, pud_t, pmd_t, pte_t in Figure 2.2. The Replayer can easily do so by using the replay handle's virtual address. Then, the Replayer performs the following steps, shown in the timeline ① of Figure 2.3. First, it flushes from the caches the data to be accessed by the replay handle. This can be done by priming the caches. Second, it clears the present bit in the leaf page table entry (pte_t). Next, it flushes from the cache subsystem the four page table entries in the PDG, PUD, PMD, and PTE required for translation. Finally, it flushes the TLB entry that stores the {VPN, PPN} translation for the replay handle access. Together, these steps will cause the replay handle to miss in the TLB, and induce a hardware page walk to locate the translation, which will miss in the Page Walk Cache (PWC) and eventually result in a minor

page fault.

Sometimes, it is also possible for the Replayer to use an instruction with a naturally occurring page fault as the replay handle.

Page Walk and Speculative Execution

After the attack is set-up, the Replayer allows the Victim to resume execution and issue the replay handle. The operation is shown in timeline ③ of Figure 2.3. The replay handle access misses in the L1 TLB, L2 TLB, and PWC, and initiates a page walk. The hardware page walker fetches the necessary page table entries sequentially, starting from PGD, then PUD, PMD, and finally PTE. The Replayer can tune the duration of the page walk time to take from a few cycles to over one thousand cycles, by ensuring that the desired page table entries are either present or absent from the cache hierarchy (shown in the arrows above timeline ③ of Figure 2.3).

In the shadow of the page walk, due to speculative execution, the Victim continues executing subsequent instructions, which perform the secret code computation. Such speculative instructions execute but will not retire. They leave some state in the cache subsystem and/or create contention for hardware structures in the core. When the hardware page walker locates the leaf PTE that contains the translation, it finds that the present bit is clear. This finding eventually causes the hardware to raise a page fault exception and squash all of the speculative state in the pipeline.

The Replayer is then invoked to execute the page fault handler and handle the page fault. The Replayer could now set the present bit and allow the Victim to make forward progress. Alternatively, as shown in timeline ② of Figure 2.3, MicroScope's Replayer keeps the present bit clear and re-flushes the PGD, PUD, PMD, and PTE page table entries from the cache subsystem. As a result, as the Victim resumes and re-issues the replay handle, the whole process repeats. Timeline ④ of Figure 2.3 shows the actions of the Victim. This process can be repeated as many times as desired to denoise and extract the secret information.

Monitoring Execution

The *Monitor* is responsible for extracting the secret information of the *Victim*. Depending on the Victim application and the side channel being exploited, we distinguish two configurations. In the first one, shown in timeline ⑤ of Figure 2.3, the Monitor executes in parallel with the Victim's speculative execution. The Monitor can cause contention on shared hardware resources and monitor the behavior of the hardware. For example, an attack that monitors contention in the execution units uses this configuration.

In the second configuration, the Monitor is part of the Replayer. After the Victim has yielded control back to the Replayer, the latter inspects the result of the speculative execution, such as the state of specific cache sets. A cache-based side channel attack could use this configuration.

Summary of a MicroScope Attack

The attack consists of the following steps:

1. The Replayer identifies the replay handle and prepares the attack.
2. When the Victim executes the replay handle, it suffers a TLB miss followed by a page walk. The time taken by this step can be over one thousand cycles. It can be tuned as per the requirements of the attack.
3. In the shadow of the page walk and until the page fault is serviced, the Victim continues executing speculatively past the replay handle into the sensitive region, potentially until the ROB is full.
4. The Monitor can cause and measure contention on shared hardware resources during the Victim's speculative execution, or inspect the hardware state at the end of the Victim's speculative execution.
5. When the Replayer gains control after the replay handle causes a page fault, it can optionally leave the present bit cleared in the PTE entry. This will induce another replay cycle that the Monitor can leverage to collect more information. Before the replay, the adversary may also prime the processor state for the next measurement. For example, if it uses a Prime+Probe cache-based attack, it can re-prime the cache.
6. When sufficient traces have been gathered, the Replayer sets the present bit in the PTE entry. This enables the Victim to make forward progress.

With these steps, MicroScope can generate a large number of execution traces for one "logical" execution trace. It can denoise a side channel formed by, potentially, any instruction(s)—even ones that expose a secret only once in straight-line code.

2.4.2 Simple Attack Examples

Figure 2.4 shows several examples of codes that present opportunities for MicroScope attacks. Each example showcases a different use case.

```
1 //public address
2 handle(pub_addr);
3 ...
4 transmit(secret);
5 ...
```

(a) Single secret.

```
1 for i in ...
2     handle(pub_addrA);
3     ...
4     transmit(secret[i]);
5     ...
6     pivot(pub_addrB);
7     ...
```

(b) Loop secret.

```
1 handle(pub_addrA);
2 if (secret)
3     transmit(pub_addrB)
4 else
5     transmit(pub_addrC)
```

(c) Control flow secret.

Figure 2.4: Simple examples of codes that present opportunities for microarchitectural replay attacks.

Single-Secret Attack

Figure 2.4a shows a simple code that has a single secret. Line 2 accesses a public address (i.e., known to the OS). This access is the replay handle. After a few other instructions, sensitive code at Line 4 processes some secret data. We call this computation the *transmit* computation of the Victim, using terminology from [61]. The transmit computation may leave some state in the cache or may use specific functional units that create observable contention. The goal of the adversary is to extract the secret information. The adversary can obtain it by using MicroScope to repeatedly perform steps (2)–(5) from Section 2.4.1.

To gain more insight, consider a more detailed example of the single-secret code (Figure 2.5). The figure shows function `getSecret` in C source code (Figure 2.5a) and in assembly (Figure 2.5b). In Figure 2.5a, we see that the function increments `count` and returns `secrets[id]/key`.

With MicroScope, the adversary can leverage the read to `count` as a replay handle. In Figure 2.5b, the replay handle is the `mov` instruction at Line 6. Then, MicroScope can be used to monitor the port contention in the floating-point division functional unit that executes `secrets[id]/key`. In Figure 2.5b, the division instruction is at Line 12. This is the transmit instruction. With this support, the adversary can determine whether `secrets[id]/key` is a subnormal floating-point operation, which has a different latency.

```
1 static uint64_t count;
2 static float secrets[512];
3
4 float getSecret(int id,
5           float key){
6   //replay handle
7   count++;
8   //measurement access
9   return secrets[id]/key;
10 }
```

(a) Single-secret source.

```
1  _getSecret:
2  push   %rbp
3  mov    %rsp,%rbp
4  mov    %edi,-0x4(%rbp)
5  movss  %xmm0,-0x8(%rbp)
6  mov    0x200b27(%rip),%rax
7  add    $0x1,%rax
8  mov    %rax,0x200b1c(%rip)
9  mov    -0x4(%rbp),%eax
10 cltq
11 movss  0x601080(,%rax,4),%xmm0
12 divss  -0x8(%rbp),%xmm0
13 pop    %rbp
14 retq
```

(b) Single-secret assembly.

Figure 2.5: Single-secret detailed code.

Alternatively, MicroScope can be used to monitor the cache access made by `secrets[id]`. In Figure 2.5b, the access `secrets[id]` is at Line 11. With MicroScope, the adversary can extract the cache line address of `secrets[id]`.

Loop-Secret Attack

We now consider the scenario where we want to monitor a given instruction in different iterations of a loop. We call this case `Loop Secret`, and show an example in Figure 2.4b. In the code, the loop body has a replay handle and a transmit operation. In each iteration, the transmit operation accesses a different secret. The adversary wants to obtain the secrets of all the iterations. The challenging case is when the address of the replay handle maps to the same physical data page in all the iterations.

This scenario highlights a common problem in side channel attacks: `secret[i]` and `secret[i+1]` may induce similar effects, making it hard to disambiguate between the two. For example, both secrets may co-locate in the same cache line, or induce similar pressure on the execution units. This fact severely impedes the ability to distinguish the two accesses.

MicroScope addresses this challenge through two capabilities. The first one, discussed in Section 2.4.1, is that the Replayer can dynamically tune the duration of the speculative execution, by

controlling the page walk duration. In particular, the speculative execution window can be tuned to be short enough to allow the execution of only a single secret transmission per replay. This allows the Replayer to extract `secret[i]` without any noise.

The second capability is that the Replayer can use a *second* memory instruction to move between the replay handles in different iterations. This second instruction is located after the transmit instruction in program order, and we call it the *Pivot* instruction. For example, in Figure 2.4b, the instruction at Line 6 can act as the pivot. The only condition that the pivot has to satisfy is that its address should map to a different physical page than the replay handle.

MicroScope uses the pivot as follows. After the adversary infers `secret[i]` and is ready to proceed to extract `secret[i+1]`, the adversary performs one additional action during step 6 in Section 2.4.1. Specifically, after setting the present bit in the PTE entry for the replay handle, it clears the present bit in the PTE entry for the pivot, and resumes the Victim's execution. As a result, all the Victim instructions before the pivot are retired, and a new page fault is incurred for the pivot.

When the Replayer is invoked to handle the pivot's page fault, it sets the present bit for the pivot and clears the present bit for the replay handle. This is possible because we choose the pivot from a different page than the replay handle. When the Victim resumes execution, it retires all the instructions of the current iteration and proceeds to the next iteration, suffering a page fault in the replay handle. Steps 2- 5 repeat again, enabling the monitoring of `secret[i+1]`. The process is repeated for all the iterations.

As a special case of this attack scenario, when the transmit instruction (Line 4) is itself a memory instruction, MicroScope can simply use the transmit instruction as the pivot. This eliminates the need for a third instruction to act as pivot.

Control Flow Secret Attack

A final scenario that is commonly exploited using side channels is a secret-dependent branch condition. We call this case `Control Flow Secret`, and show an example in Figure 2.4c. In the code, the direction of the branch is determined by a secret, which the adversary wants to extract.

As shown in the figure, the adversary uses a replay handle before the branch, and a transmit operation in both paths out of the branch. The adversary can extract the direction taken by the branch using at least two different types of side channels. First, if Lines 3 and 5 in Figure 2.4c access different cache lines, then the Monitor can perform a cache based side-channel attack to identify the cache line accessed, and deduce the branch direction.

A second case is when the two paths out of the branch access the same addresses but perform different computations—e.g., one path performs a multiplication and the other performs a division.

In this case, the transmit instructions are instructions that use the functional units. The Monitor can apply pressure on the functional units and, by monitoring the contention observed, deduce the operation that the code performs.

Prediction. The above situation is slightly more complicated in the presence of control-flow prediction such as branch prediction. With a branch predictor, the branch direction will initially be a function of the predictor state, *not* the secret. If the secret does not match the prediction, execution will squash. In this case both sides of the branch will execute, complicating the adversary's measurement.

MicroScope deals with this situation using the following insight: If the branch predictor state is *public*, whether there is a misprediction (re-execution) leaks the secret value, i.e., reveals secret==predictor state. The adversary can measure whether there is a misprediction by monitoring side channels for both sides of the branch in different replays.

The branch predictor state will likely be public in our setting. For example, the adversary can prime the predictor to a known state as in [62]. Likewise, if the predictor is flushed at enclave entry [40] the very act of flushing it puts it into a known state.

2.4.3 Exploiting Port Contention

To show the capabilities of MicroScope, we implement two popular attacks: in this section, we perform a port contention attack similar to PortSmash [32] without noise; in the next section, we use a cache-based side channel to attack AES.

In a port contention attack, the attacker tries to infer a few arithmetic operations performed by the Victim. Typically, the Monitor executes different types of instructions on the same core as the Victim, to create contention on the functional units, and observes the resulting contention. These attacks can have very high resolution [32], since they can potentially leak secrets at instruction granularity—even if the Victim code is fully contained in a single instruction and data cache line. However, they suffer from high noise due to the difficulty of perfectly aligning the timing of the execution of Victim and Monitor instructions.

We build the attack using the `Control Flow Secret` code of Figure 2.4c. One side of the branch performs two integer multiplications, while the other side performs two floating-point divisions. Importantly, there is *no loop* in the code; each side of the branch simply performs the two operations. The assembly code for the two sides of the branch is shown in Figure 2.6a (multiplication) and 2.6b (division). For clarity, each code snippet also includes the replay handle instruction in Line 1. Such instruction is executed before the branch. We can see that, in Lines 13 and 15, one code performs two integer multiplications and the other two floating-point divisions.

```
1 addq   $0x1,0x20(%rbp)
2 ...
3 __victim_mul
4 mov    0x2014b1(%rip),%rax
5 mov    %rax,0x20(%rsp)
6 mov    0x201498(%rip),%rax
7 mov    %rax,0x28(%rsp)
8 mov    0x20(%rsp),%rsi
9 mov    0x28(%rsp),%rdi
10 mov   (%rsi),%rbx
11 mov   (%rdi),%rcx
12 mov   %rcx,%rax
13 mul   %rbx
14 mov   %rcx,%rax
15 mul   %rbx
```

(a) Multiplication side.

```
1 addq   $0x1,0x20 (%rbp)
2 ...
3 __victim_div
4 mov    0x201548(%rip),%rax
5 mov    %rax,0x10(%rsp)
6 mov    0x20153f(%rip),%rax
7 mov    %rax,0x18(%rsp)
8 mov    0x10(%rsp),%rax
9 mov    0x18(%rsp),%rbx
10 movsd (%rax),%xmm0
11 movsd (%rbx),%xmm1
12 movsd %xmm1,%xmm2
13 divsd %xmm0,%xmm2
14 movsd %xmm1,%xmm3
15 divsd %xmm0,%xmm3
```

(b) Division side.

Figure 2.6: Victim code executing two multiplications (a) or two divisions (b). Note that code is not in a loop.

The goal of the adversary is to extract the secret that decides the direction of the branch.[2] To do so, the Monitor executes the simple port contention monitor code of Figure 2.7a. The code is a loop where each iteration repeatedly invokes `unit_div_contention()`, which performs a single floating-point division operation. The code measures the time taken by these operations and stores the time in an array. Figure 2.7b shows the assembly code of `unit_div_contention()`. Line 11 performs the single division, which creates port contention in the division functional unit.

The attack begins with the Replayer causing a page fault at Line 1 of Figure 2.6. MicroScope forces the victim to keep replaying the code that follows, which is either 2.6a or 2.6b, depending on the value of the secret. On a different SMT context of the same physical core, the Monitor concurrently executes the code in Figure 2.7a. The Monitor's `divsd` instruction at Line 11 of Figure 2.7b may or may not experience contention depending on the execution path of the Victim. If the Victim takes the path with `mul` (Figure 2.6a), the Monitor does not experience any contention and executes fast. If it takes the path with `divsd` (Figure 2.6b), the Monitor experiences contention and executes slower. Based on the execution time of the Monitor code, MicroScope reliably identifies the operation executed by the Victim, and thus the secret, after a few replay iterations.

[2]We note that prior work demonstrated how to infer branch directions in SGX. Their approach, however, is no longer effective with today's countermeasures that flush branch predictor state at the enclave boundary [40].

```
for (j = 0; j < buff; j++){
  t1 = read_timer();
  for (i = 0; i < cont; i++){
  // cause contention
    unit_div_contention();
  }
  t2 = read_timer();
  buffer[j] = t2 - t1;
}
```

(a) Monitor source code.

```
__unit_div_contention
mov    0x2012f1(%rip),%rax
mov    %rax,-0xc0(%rbp)
mov    0x2012eb(%rip),%rax
mov    %rax,-0xb8(%rbp)
mov    -0xc0(%rbp),%rax
mov    -0xb8(%rbp),%rbx
movsd  (%rax),%xmm0
movsd  (%rbx),%xmm1
movsd  %xmm1,%xmm2
divsd  %xmm0,%xmm2
```

(b) Assembly code for the division operation.

Figure 2.7: Monitor code that creates and measures port contention in the division functional unit.

This attack can also be used to discover fine-grain properties about an instruction's execution. As indicated before, one example is whether an individual floating-point operation receives a subnormal input. Prior attacks in this space are very coarse-grained, and can only measure whole-program timing [29].

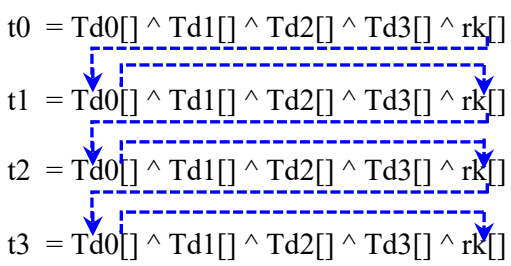
2.4.4 Attacking AES

This section shows how MicroScope is used to attack AES decryption. We consider the AES decryption implementation from OpenSSL 0.9.8. [63]. For key sizes equal to 128, 192, and 256 bits, the algorithm performs 10, 12, and 14 rounds of computation, respectively. During each round, a set of pre-computed tables are used to generate the substitution and permutation values. Figure 2.8a shows the upper part of a computation round. For simplicity, we only focus on the upper part of a computation round; the lower part is similar. In the code shown in Figure 2.8a, each of the tables `Td0`, `Td1`, `Td2`, and `Td3` stores 256 unsigned integer values, and `rk` is an array of 60 unsigned integer elements. Our goal in this attack is to extract which entries of the `Td0-Td3` tables are read in each assignment in Figure 2.8a.

MicroScope attacks the AES decryption function using two main observations. First, the `Td0-Td3` tables and the `rk` array are stored in different physical pages. Therefore, MicroScope uses an access to `rk` as a replay handle, and an access to one of the `Td` tables as a pivot. This approach was described in Section 2.4.2. Second, the Replayer can fine-tune the page walk duration so that

```
1 for (;;) {
2   t0 = Td0[(s0 >> 24)] ^ Td1[(s3 >> 16) & 0xff] ^
3        Td2[(s2 >> 8) & 0xff] ^ Td3[(s1)&0xff] ^ rk[4];
4   t1 = Td0[(s1 >> 24)] ^ Td1[(s0 >> 16) & 0xff] ^
5        Td2[(s3 >> 8) & 0xff] ^ Td3[(s2)&0xff] ^ rk[5];
6   t2 = Td0[(s2 >> 24)] ^ Td1[(s1 >> 16) & 0xff] ^
7        Td2[(s0 >> 8) & 0xff] ^ Td3[(s3)&0xff] ^ rk[6];
8   t3 = Td0[(s3 >> 24)] ^ Td1[(s2 >> 16) & 0xff] ^
9        Td2[(s1 >> 8) & 0xff] ^ Td3[(s0)&0xff] ^ rk[7];
10
11  rk += 8;
12  if (--r == 0) {
13    break;
14  }
15  ...
16  ...
17 }
```

(a) AES decryption code from OpenSSL.



(b) MicroScope's replay handle and pivot path.

Figure 2.8: Using MicroScope to attack AES decryption.

a replay covers only a small number of instructions. Hence, with such a small replay window, MicroScope can extract the desired information without noise. Overall, with these two strategies, we mount an attack where the adversary single steps the decryption function, extracting all the information without noise.

Specifically, the Replayer starts by utilizing the access to `rk[4]` in Line 3 of Figure 2.8a as the replay handle, and tunes the page walk duration so that the replay covers the instructions from Line 4 to Line 9. After each page fault is triggered, the Replayer acts as the Monitor, and accesses all the cache lines of all the `Td` tables. Based on the access times, after several replays, the Replayer

can reliably deduce the lines accessed speculatively by the Victim. However, it does not know if a given line was accessed in the assignment to `t1`, `t2`, or `t3`.

After extracting this information, the Replayer sets the present bit for the `rk[4]` page, and clears the present bit for the `Td0` page. As explained in the `Loop Secret` attack of Section 2.4.2, `Td0` in Line 4 is a pivot. When the Victim resumes execution, the `rk[4]` access in Line 3 commits, and a page fault is triggered at the `Td0` access in Line 4. Now, the Replayer sets the present bit for the `Td0` page, and clears the present bit for the `rk[5]` page. As execution resumes, the Replayer now measures the accesses in Lines 6 to 9 of this iteration, and in Lines 2 to 3 of the next iteration. Hence, it can disambiguate any overlapping accesses from the instructions in Lines 4 to 5, since these instructions are no longer replayed.

The process repeats for the next lines and across loop iterations. Figure 2.8b shows a graphical representation of the path followed by the replay handle and pivot in one iteration. Note that, as described, this algorithm misses the accesses to tables `Td0-Td3` in Lines 2 and 3 for the first iteration. However, such values are obtained by using a replay handle before the loop.

Overall, with this approach, MicroScope reliably extracts all the cache accesses performed during the decryption. Importantly, MicroScope does it with only a single execution of AES decryption.

2.5 MICROSCOPE IMPLEMENTATION

In this section, we present the MicroScope framework that we implemented in the Linux kernel.

2.5.1 Attack Execution Path

Figure 2.9 shows a high-level view of the execution path of a MicroScope attack. The figure shows the user space, the kernel space, and the MicroScope module that we implemented in the kernel.

Applications issue memory accesses using virtual addresses, which are later translated to physical ones (①). When the MMU identifies a page fault, it raises an exception and yields control to the OS to handle it (②). The page fault handler in the OS identifies what type of page fault it is, and calls the appropriate routine to service it (③). If it is a fault due to the present bit being clear, our modified page fault handler compares the faulting PTE entry to the ones currently marked as under attack. On a match, trampoline code redirects the page fault handler to the MicroScope module that we implemented (④). The MicroScope module may change the present bits of the PTE entries under attack (⑤), and prevents the OS page handler from interfering with them. After

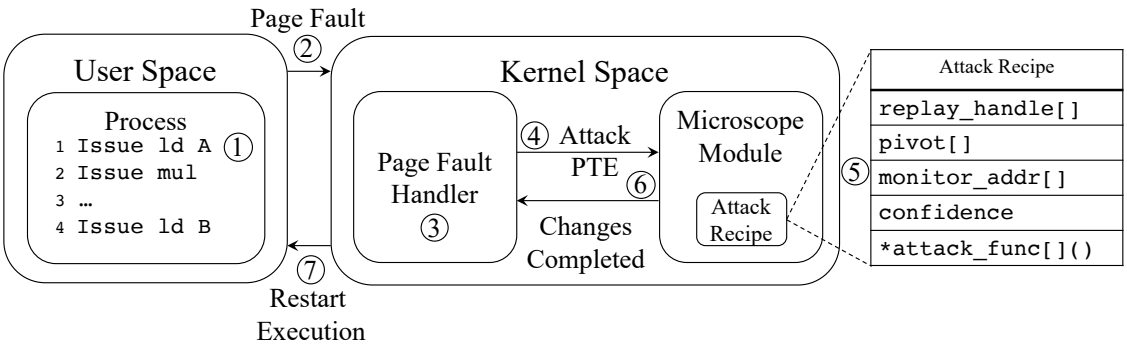


Figure 2.9: Execution path of a MicroScope attack.

the MicroScope module completes its operations, the page fault handler is allowed to continue (⑥). Finally, control returns to the application (⑦).

2.5.2 MicroScope Module

The MicroScope module in the kernel uses a set of structures described below.

Attack Recipes

The `Attack Recipes` is a structure in the MicroScope module that stores all the required information for specific microarchitectural replay attacks. For a given attack, it stores the replay handle, the pivot, and addresses to monitor for cache based side-channel attacks. It also includes a confidence threshold that is used to decide when the noise is low enough to stop the replays. Finally, each recipe defines a set of attack functions that are used to perform the attack.

This modular design allows an attacker to use a variety of approaches to perform an attack, and to dynamically change the attack recipe depending on the victim behavior. For example, if a side-channel attack is unsuccessful for a number of replays, the attacker can switch from a long page walk to a short one.

Attack Operations

MicroScope performs a number of operations to successfully carry-out microarchitectural replay attacks. The MicroScope module contains the code needed to execute such operations. The operations are as follows. First, MicroScope can identify the page table entries required for a virtual memory translation. This is achieved by performing a software page walk through the page table entries. Second, MicroScope can flush specific page table entries from the PWC and from

Function	Operands	Semantics
provide_replay_handle	addr	Provide a replay handle
provide_pivot	addr	Provide a pivot
provide_monitor_addr	addr	Provide address to monitor
initiate_page_walk	addr, length	Initiate a walk of *length*
initiate_page_fault	addr	Initiate a page fault

Table 2.2: API used by a user process to access MicroScope.

the cache hierarchy. Third, it can invalidate TLB entries. Fourth, it can communicate through shared memory or signals with the Monitor that runs concurrently with the Victim; it sends stop and start signals to the Monitor when the Victim pauses and resumes, respectively, as well as other information based on the attack recipe. Finally, in cache based side-channel attacks, MicroScope can prime the cache system.

Interface to the User for Attack Exploration

To enable microarchitectural replay attack exploration, MicroScope provides an interface for the user to pass information to the MicroScope module. This interface enables the operations in Table 2.2. Some operations allow a user to provide a replay handle, a pivot, and addresses to monitor for cache based side-channel attacks. In addition, the user can force a specific address to initiate a page walk of *length* page-table levels, where length can vary from 1 to 4. Finally, the user can force a specific address to suffer a page fault.

2.6 EVALUATION

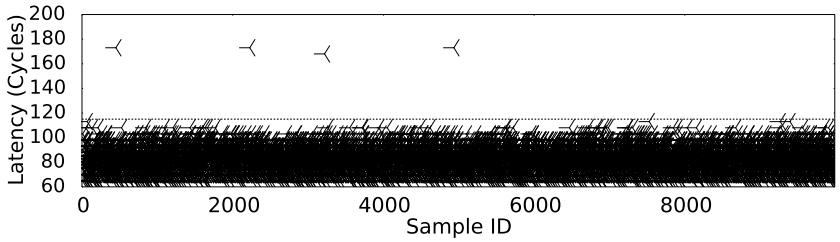
We evaluate MicroScope on a Dell Precision Tower 5810 with an Intel Xeon E5-1630 v3 processor running Ubuntu with the 4.4 Linux kernel. We note that while our current prototype is not on an SGX-equipped machine, our attack framework uses an abstraction equivalent to the one defined by the SGX enclaves, as discussed in Section 2.3. Related work makes similar assumptions [36]. In this section, we evaluate two attacks: the port contention attack of Section 2.4.3, and the AES attack of Section 2.4.4.

2.6.1 Port Contention Attack

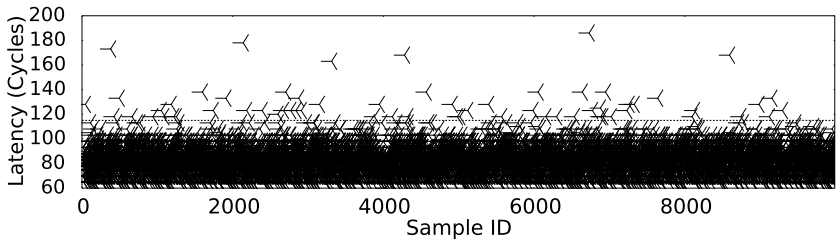
In this attack, the Monitor performs the computation shown in Figure 2.7a. Concurrently, MicroScope forces the Victim to replay either the code in Figure 2.6a or the one in Figure 2.6b. The

Monitor performs 10,000 measurements. They measure a single logical run of the Victim, as the Victim code snippet is replayed many times.

Figure 2.10 shows the latency in cycles of each of the 10,000 Monitor measurements while the Victim runs the code with the two multiplications (Figure 2.6a), or the one with the two divisions (Figure 2.6b). When the victim executes the code with the two multiplications, the latency measurements in Figure 2.10a show that all but 4 of the samples take less than 120 cycles. Hence, we set the contention threshold to slightly less than 120 cycles, as shown by the horizontal line.



(a) Victim executes two multiply operations as shown in Figure 2.6a.



(b) Victim executes two division operations as shown in Figure 2.6b.

Figure 2.10: Latencies measured by performing a port contention attack.

When the victim executes the code with the two divisions, the latency measurements in Figure 2.10b show that 64 measurements are above the threshold of 120 cycles. To understand this result, note that most Monitor samples are taken while the page fault handling code is running, rather than when the Victim code is running. The reason is that the page fault handling code executes for considerably longer than the Victim code in each replay iteration, and we use a simple free-running Monitor. For this reason, many measurements are below the threshold for both figures.

However, there is substantial difference between Figure 2.10b and Figure 2.10a. The former has 16x more samples over the threshold. This makes the two cases clearly distinguishable.

Overall, MicroScope is able to detect the presence or absence of two divide instructions, without any loop. It does so by denoising a notoriously noisy side channel through replay.

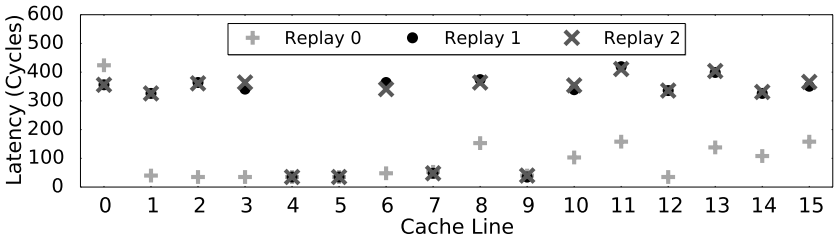


Figure 2.11: Latency of the accesses to the `Td1` table after each of three replays of one iteration of AES.

2.6.2 AES Attack

We use MicroScope to perform the cache side-channel attack on AES described in Section 2.4.4. We focus on one iteration of the loop in Figure 2.8a, and replay three times to obtain the addresses of the cache lines accessed by the `Td0-Td3` tables. Each of these tables uses 16 cache lines.

Before the first replay (`Replay 0`), the Replayer does not prime the cache hierarchy. Hence, when the Replayer probes the cache after the replay, it finds the cache lines of the tables in different levels of the cache hierarchy. Before each of the next two replays (`Replay 1` and `Replay 2`), the Replayer primes the cache hierarchy, evicting all the lines of the tables to main memory. Therefore, when the Replayer probes the cache after each replay, it finds the lines of the tables accessed by the Victim in the L1 cache, and the rest of the lines in main memory. As a result, the Replayer is able to identify the lines accessed by the victim.

Figure 2.11 shows the latency in cycles (Y axis) observed by the Replayer as it accesses each of the 16 lines of table `Td1` (X axis) after each replay. We see that, after `Replay 0`, some lines have a latency lower than 60 cycles, others between 100 and 200 cycles, and one over 300. They correspond to hits in the L1, hits in L2/L3, and misses in L3, respectively. After `Replay 1` and `Replay 2`, however, the picture is very clear and consistent. Only lines 4, 5, 7, and 9 hit in the L1, and all the other lines miss in L3. This experiment shows that MicroScope is able to extract the lines accessed in the AES tables without noise in a single logical run.

2.7 GENERALIZING MICROARCHITECTURAL REPLAY ATTACKS

While this chapter focused on a specific family of attacks (denoising microarchitectural side channels using page fault-inducing loads), the notion of replaying snippets in a program's execution is even more general and can be used to mount other privacy- or integrity-based attacks.

Figure 2.12 gives a framework illustrating the different components in a microarchitectural replay attack. In our attack, the *replay handle* is a page fault-inducing load, the *replayed code*

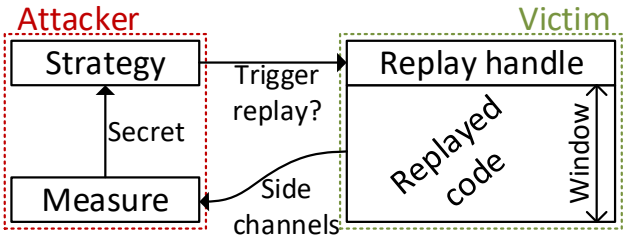


Figure 2.12: Generalized microarchitectural replay attacks.

contains instructions that leak privacy over microarchitectural side channels, and the attacker's *strategy* is to unconditionally clear the page table present bit until it has high confidence that it has extracted the secret. We now discuss how to create different attacks by changing each of these components.

2.7.1 Attacks on Program Integrity

Our original goal with microarchitectural replay attacks was to bias non-deterministic instructions such as the Intel true random number generator `RDRAND`. Suppose the replayed code contains a `RDRAND` instruction. If the attacker learns the `RDRAND` return value over a side channel, its strategy is to *selectively* replay the Victim depending on the returned value (e.g., if it is odd, or satisfies some other requirement). This is within the attacker's power: to selectively replay the Victim, the OS can access the last level page table (the PTE) directly and set/clear the present bit before the hardware page walker reaches it. The result is that the attacker can prevent the victim from obtaining certain values, effectively biasing the `RDRAND` from the Victim's perspective.

We managed to get all the components of such an attack to work correctly. However, it turns out that the current implementation of `RDRAND` on Intel platforms includes a form of fence. This fence prevents speculation after `RDRAND`, and the attack does not go through. In discussions with Intel, it appears that the reason for including this fence was not related to security. The lesson is that there should be such a fence, for security reasons.

More generally, the above discussion on integrity applies when there is any instruction in the replayed code that is non-deterministic. For example: `RDRAND`, `RDSEED`, `RDTSC`, or memory accesses to shared, writeable variables.

2.7.2 Attacks Using Different Replay Handles

While this chapter uses page fault-inducing loads as replay handles, there are other instructions which can similarly cause a subsequent dynamic instruction to execute multiple times. For ex-

ample, entering a transaction using transactional memory may cause code within the transaction to replay if the transaction aborts (e.g., if there are write set conflicts). Intel's Transactional Synchronization Extensions (TSX) in particular will abort a transaction if dirty data is evicted from the private cache, which can be easily controlled by an attacker. (We note that prior work has proposed using TSX in conjunction with SGX for defensive purposes [64].) One benefit of using TSX enter as a replay handle is that the window of replayed code is large, i.e., potentially the number of instructions in the transaction as opposed to the ROB size. This changes mitigations strategies. For example, fencing `RDRAND` (see above) will no longer be effective.

Other instructions can cause a limited number of replays. For example, any instruction which can squash speculative execution [65], e.g., a branch that mispredicts, can cause some subsequent code to be replayed. Since a branch will not mispredict an infinite number of times, the application will eventually make forward progress. However, the number of replays may still be large, e.g., if there are multiple branches in-flight, all of which mispredict. To maximize replays, the adversary can perform setup before the victim runs. For example, it can prime the branch predictor (similar to [62]) to mispredict if there are not already mechanisms to flush the predictors on context switches [40].

2.7.3 Amplifying Physical Side Channels

While our focus was to amplify microarchitecture side channels, microarchitectural replay attacks may also be an effective tool to amplify physical channels such as power and EM [59, 60]. For example, in the limit, the replayed code may be as little as a single instruction in the Victim, plus the attacker instructions needed to setup the next replay. Unsurprisingly, reducing Victim execution to fewer irrelevant instructions can improve physical attack effectiveness by denoising attack traces [66].

2.8 POSSIBLE COUNTERMEASURES

We now overview possible defense solutions and discuss their performance and security implications.

The root cause of microarchitectural replay attacks is that each dynamic instruction may execute more than one time. Based on the discussion in Section 2.7, this can be for a variety of reasons (e.g., a page fault, transaction abort, squash during speculative execution). Thus, it is clear that new, general security properties are required to comprehensively address these vulnerabilities. While we are working to design a comprehensive solution, we review some point mitigation strategies below that can be helpful to prevent specific attacks.

Fences on Pipeline Flushes. The obvious defense against attack variants, whose replayed code is contained within the ROB (see Section 2.7), is for the hardware or the OS to insert a fence after each pipeline flush. However, there are many corner cases that need to be considered. For example, it is possible that multiple instructions in a row induce a pipeline flush. This can be due to different causes, such as multiple page faults and/or branch mispredictions in close proximity. In these cases, even if a fence is introduced after every pipeline flush, the adversary can extract information from the resulting multiple replays.

Speculative Execution Defenses. MicroScope relies on speculative execution to replay Victim instructions. Therefore, a defense solution that holistically blocks side effects caused by speculative execution can effectively block MicroScope. However, existing defense solutions either have limited defense coverage or introduce substantial performance overhead. For example, using fences [41] or mechanisms such as InvisiSpec [67] or SafeSpec [68] only block specific covert channels such as the cache, and apply protections to all loads, which incurs large overhead. One idea to adapt these works to our attack is to only enable defenses while page faults are outstanding. Even with such an idea, however, these protections do not address side channels on the other shared processor resources, such as port contention [32]. Further, there may be a large gap in time between when an instruction executes and an older load misses in the TLB.

Page Fault Protection Schemes. As MicroScope relies on page faults to trigger replays, we consider whether page fault oriented defense mechanisms could be effective to defeat MicroScope. In particular, T-SGX [64] uses Intel's Transactional Synchronization Extensions (TSX) to capture page faults within an SGX application, and redirect their handling to a user-level code instead of the OS. The goal of T-SGX is to mitigate a controlled side-channel attack that leaks information through the sequence of page faults. However, T-SGX does not mitigate other types of side channels such as cache- or contention-based side channels.

The T-SGX authors are unable to distinguish between page faults and regular interrupts as the cause of transaction aborts. Hence, they use a threshold $N = 10$ of failed transactions to terminate the application. This design decision still provides $N - 1$ replays to MicroScope. Such number can be sufficient in many attacks.

Déjà Vu [69] is a technique that finds out whether a program is compromised by measuring with a clock if it takes an abnormal amount of time to execute. Déjà Vu uses TSX to protect the reference-clock thread. However, Déjà Vu presents two challenges. First, since the time to service an actual page fault is much longer than the time to perform a MicroScope replay, replays can be masked by ordinary application page faults. Second, to update state in Déjà Vu, the clock instructions need to retire. Thus, the attacker can potentially replay indefinitely on a replay handle, while concurrently preventing the clock instructions from retiring until the secret is extracted.

Both of the above defenses rely on Intel TSX. As discussed in Section 2.7, TSX itself creates a new mechanism with which to create replays, through transaction aborts. Thus, we believe further research is needed before applying either of the above defenses to any variant of microarchitectural replay attack.

Finally, Shinde et. al. [70] proposed a mitigation scheme to obfuscate page-granularity access patterns by providing page-fault obliviousness (or PF-obliviousness). The main idea is to change the program to have the same page access patterns for different input values, by inserting redundant memory accesses. Interestingly, this mechanism makes it easier for MicroScope to perform an attack, as the added memory accesses provide more replay handles.

2.9 RELATED WORK

We discuss several related works on exploiting speculative execution and improving side-channel attack accuracy.

Transient Execution Attacks. Starting with Meltdown [71] and Spectre [62], out-of-order speculative execution has created a new attack class known as transient execution attacks. These attacks rely on a victim executing code that it would not have executed at program level—e.g., instructions after a faulting load [71] or mispredicted branch [62]. The Foreshadow [72] attack is a Meltdown-style attack on SGX.

MicroScope is fundamentally different from transient execution attacks as it uses out-of-order speculative execution to create a replay engine. Replayed instructions may or may not be transient—e.g., instructions after a page faulting load may retire once the page fault is satisfied. Further, while Foreshadow exploits an implementation defect (L1TF), MicroScope exploits SGX's design, namely how the attacker is allowed to control demand paging.

Improving Side-Channel Attack Accuracy. As side channel attacks are generally noisy, there are several works that try to improve the temporal resolution and decrease the noise of cache attacks. Cache games [47] exploits the OS scheduler to slow down the victim execution and achieve higher attack resolution. CacheZoom [37] inserts frequent interrupts in the victim SGX application to stop the program every several data access instructions. SGX Grand Exposure [34] tries to minimize the noise during an attack by disabling interrupts, and uses performance monitoring counters to detect cache evictions. We provide more background on like attacks in Section 2.2.4.

All of the mechanisms mentioned can only improve the attack resolution in a limited manner. Also, they are helpful only for cache attacks. Compared to these approaches, MicroScope attains the highest temporal resolution with the minimum noise, since it replays the Victim execution in a fine-grained manner many times. In addition, MicroScope is the first framework that is gen-

eral enough to be applicable to both cache attacks and other contention-based attacks on various hardware components [22, 31, 32].

2.10 CONCLUSION

Side-channel attacks are popular approaches to attack applications. However, many modern fine-grained side channels introduce too much noise to reliably leak secrets, even when the victim is run hundreds of times.

In this chapter, we introduced Microarchitectural Replay Attacks targeting hardware-based Trusted Execution Environments such as Intel's SGX. We presented a framework, called MicroScope, which can denoise nearly arbitrary microarchitectural side channels in a *single run*, by causing the victim to replay on a page faulting instruction. We used MicroScope to denoise notoriously noisy side channels. In particular, our attack was able to detect the presence or absence of *two divide* instructions in a single run. Finally, we showed that MicroScope is able to single-step and denoise a cache-based attack on the AES implementation of OpenSSL.

Chapter 3: Thwarting Microarchitectural Replay Attacks

3.1 INTRODUCTION

The microarchitecture of modern computer systems creates many side channels that allow an attacker running on a different process to exfiltrate execution information from a victim. Indeed, hardware resources such as caches [22, 23, 24, 26, 73, 74], TLBs [33], branch predictors [28, 75, 76], load-store units [31], execution ports [32, 77, 78], functional units [29, 32], and DRAM [30] have been shown to leak information.

Luckily, a limitation of these microarchitectural side channels is that they are often very noisy. To extract information, the execution of the attacker and the victim processes has to be carefully orchestrated [23, 26, 73], and often does not work as planned. Hence, an attacker needs to rely on many executions of the victim code section to obtain valuable information. Further, secrets in code sections that are executed only once or only a few times are hard to exfiltrate.

Unfortunately, as indicated in the previous chapter, Microarchitectural Replay Attacks (MRAs) are able to eliminate the measurement variation in (i.e., to denoise) most microarchitecural side channels. This is the case even if the victim is run only once. Such capability makes the plethora of existing side-channel attacks look formidable and suggests the potential for a new wave of powerful side channel attacks.

MRAs use the fact that, in out-of-order cores, pipeline squashes due to events such as exceptions, branch mispredictions, or memory consistency model violations can force a dynamic instruction to re-execute. Hence, in an MRA, the attacker actuates on one or multiple instructions to force the squash and re-execution of a victim instruction V multiple times. This capability enables the attacker to cleanly observe the side-effects of V.

MRAs are powerful because they exploit a central mechanism in modern processors: out-of-order execution with in-order retirement. Moreover, MRAs are not limited to speculative execution attacks: the instruction V that is replayed can be a correct instruction that will eventually retire. Finally, MRAs come in many forms. While the MRA in the previous chapter exposed the side effects of V by repeatedly causing a page fault on an older instruction, the same result can be attained with other events that trigger pipeline flushes.

To thwart MRAs, one has to eliminate instruction replay or at least bound the number of replays that a victim instruction V may suffer. The goal is to deny the attacker the opportunity to see many executions of V.

This chapter presents the first mechanism to thwart MRAs. We call it *Jamais Vu*. The simple idea is to first record the victim instructions that are squashed. Then, as each victim instruction

is re-inserted into the Reorder Buffer (ROB), *Jamais Vu* automatically places a fence before it to prevent the attacker from squashing it again. In reality, pipeline refill after a squash may not bring in the same instructions that were squashed, or not in the same order. Consequently, *Jamais Vu* has to provide a more elaborate design.

At a high level, there are two main design questions to answer: how to record the squashed instructions and for how long to keep the record. *Jamais Vu* presents several designs of the mechanism that give different answers to these questions, effectively providing different trade-offs between execution overhead, security, and implementation complexity.

Architectural simulations using SPEC17 applications show the effectiveness of the *Jamais Vu* designs. Our preferred design, called *Epoch-Rem*, can effectively mitigate MRAs, and has an average execution time overhead of 22.7% in benign executions. Further, it only needs simple hardware counting Bloom filters associated with the ROB.

Contributions. This chapter makes the following contributions.

- *Jamais Vu*, the first defense mechanism to thwart MRAs. It performs selective fencing of instructions to prevent replays.
- Several designs of *Jamais Vu* that provide different tradeoffs between execution overhead, security, and complexity.
- An evaluation of these designs using simulations.

3.2 BRIEF BACKGROUND

3.2.1 Microarchitectural Side Channel Attacks

Microarchitectural side channels allow an attacker to exploit timing variations in accessing a shared resource to learn information about a victim process' execution. Side channel attacks have been demonstrated that rely on a variety of microarchitectural resources including the CPU caches [23, 24, 26, 47, 73, 74, 79, 80, 81, 82, 83], the TLB [33], execution ports [32, 77, 78], functional units [29, 84, 85], cache banks [22], the branch predictor [28, 75, 76] and DRAM row buffers [30]. A common challenge for these attacks is the need to account for noise [86]. To solve this challenge, existing attacks run in the absence of background noise (e.g., [26, 73, 87]) or rely on many victim executions (e.g., [22, 23, 24, 47, 76, 79, 80]).

3.2.2 Out-of-Order Execution

Dynamically-scheduled processors [42] execute data-independent instructions in parallel, out of program order, and thereby exploit instruction-level parallelism [88] to improve performance. Instructions enter the ROB in program order, execute possibly out of program order, and finally retire (i.e., are removed) from the ROB in program order [43]. After retirement, writes merge with the caches in an order allowed by the memory consistency model.

3.2.3 Definition of Microarchitectural Replay Attacks

As indicated in the previous chapter, a Microarchitectural Replay Attack (MRA) consists of forcing the squash and re-execution of an instruction *I* multiple times. This capability enables the attacker to cleanly observe the side-effects of *I*. There are multiple events that cause a pipeline squash, such as various exceptions (e.g., page faults), branch mispredictions, memory consistency model violations, and interrupts.

In this chapter, we refer to the instruction that causes the squash as the *Squashing* (S) one; we refer to the younger instructions in the ROB that the Squashing one squashes as the *Victims* (V). The type of Victim instruction that the attacker wants to replay is one whose usage of and/or contention for a shared resource depends on a secret. We call such an instruction a *transmitter*. Loads are obvious transmitters, as they use the shared cache hierarchy. However, many instructions can be transmitters, including those that use functional units.

3.3 A MEMORY CONSISTENCY MODEL VIOLATION MRA

MicroScope used OS-induced page faults to cause replays. In this section, we provide evidence for the first time that memory consistency model violations can also create MRAs [1]. In these MRAs, a load issued speculatively to a location x is squashed because either the cache receives an invalidation for the line where x resides, or the line where x resides is evicted from the cache [89, 90]. A user-level attacker can force either event.

Our proof-of-concept experiment consists of a victim and an attacker running on two sibling CPU threads and sharing a cache line A. In reality, the PoC variant using cache evictions could be carried out without victim and attacker sharing memory and, instead, using other eviction techniques [91].

[1] The memory consistency MRA attack was implemented by Riccardo Paccagnella and is used in this thesis with his consent.

The proof-of-concept is shown in Figure 3.1. The victim first brings the shared cache line *A* into the L1 cache and ensures that a private cache line *B* is not in the cache.

The victim then reads line *B* and misses in the entire cache hierarchy. While *B* is being loaded from memory, the victim speculatively loads shared cache line *A*, followed by other speculative instructions. The attacker's goal is to evict *A* or write to *A* after *A* has been loaded speculatively by the victim into the cache but before the load *B* completes. It this is accomplished, the load(A) instruction in the victim will be squashed, together with the following instructions due to a violation of the memory model.

```
for i in 1..N
    LFENCE
    LOAD(A)    // Make LOAD(A) hit in the cache
    CLFLUSH(B) // Make LOAD(B) miss in the cache
    LFENCE
    LOAD(B)    // LOAD(B) misses in the cache
    LOAD(A)    // LOAD(A) hits in the cache and then
               // is evicted/invalidated by attacker
    ADD ...    // 40 unrelated add instructions
```

(a) Victim

```
while(1)
    CLFLUSH(A) or STORE(A) //Evict/Invalidate A
    .REPT 100    // Do nothing for a small interval
    NOP          // by executing 100 nops
    .ENDR
```

(b) Attacker.

Figure 3.1: Pseudocode of victim and attacker causing pipeline squashes due to memory consistency model violations.

We run our experiment on a 4.00 GHz quad-core Intel i7-6700K CPU. We configure the victim to run in a loop, and set up the attacker to evict/invalidate the shared cache line *A* periodically. If during some victim loop iterations the attacker's eviction/invalidation occurs after the victim has speculatively loaded *A* but before cache line *B* is loaded, the victim will incur a pipeline squash.

To detect if the victim incurs any pipeline squashes, we read the number of *machine clears* (Intel's terminology for pipeline squashes), micro-ops issued, and micro-ops retired from the hardware performance counters [92] (using Intel VTune [93] to monitor only the victim's function of interest).

	Number of squashes	Percentage of micro-ops issued that did not retire
Baseline (No attacker)	0	0%
Attacker evicting A	3,221,778	30%
Attacker writing to A	5,677,938	53%

Table 3.1: Results of experiment. The numbers are collected over 10 million victim loop iterations.

We compare these numbers under three scenarios: (1) there is no attacker; (2) the attacker periodically evicts line *A*; (3) the attacker periodically writes to line *A*. Table 3.1 reports the results of our experiment, with a victim configured with 10 million loop iterations. When there is no attacker, we measure zero pipeline squashes in the victim and all the issued micro-ops retire successfully. When the attacker periodically evicts line *A*, more than 3 million pipeline squashes occur in the victim, and 30% of the issued micro-ops never retire. Finally, when the attacker periodically writes to *A*, more than 5 million pipeline squashes occur in the victim and 53% of the issued micro-ops never retire.

These results confirm that memory consistency violations can be used as a source of pipeline squashes and replays.

3.4 THWARTING MRAS

3.4.1 Understanding the Types of MRAs

MRAs come in many forms. Table 3.2 shows three orthogonal characteristics that can help us understand these threats. The first one is the source of the squash. Recall that there are many sources, namely various exceptions (e.g., page faults), branch mispredictions, memory consistency model violations, and interrupts [94]. Some sources can force a single Squashing instruction to trigger squashes repeatedly, while others only squash the pipeline once. Examples of the former are attacker-controlled page faults and memory consistency model violations; examples of the latter are branch mispredictions. The former can create more leakage.

Moreover, some sources trigger the squash when the Squashing instruction is at the ROB head, while others can do it at any position in the ROB. The former include exceptions, while the latter include branch mispredictions and memory consistency violations. The former create more leakage because they typically squash and replay more Victims.

Characteristic	Why It Matters
Source of squash?	Determines: (i) the number of squashes and (ii) where in the ROB the squash occurs
Victim is transient?	If yes, it can leak a wider variety of secrets
Victim is in a loop leaking the same secret every time it executes?	If yes, it is harder to defend: (i) leaks from multiple iterations add up (ii) *multi-instance* squashes

Table 3.2: Characteristics of microarchitectural replay attacks.

Figure 3.2(a) shows an example where repeated exceptions on one or more instructions can squash and replay a transmitter many times. This is the example used in the previous chapter. Figure 3.2(b) shows an example where attacker-instigated mispredictions in multiple branches can result in the repeated squash and replay of a transmitter. Different branch structures and different orders of resolution result in different replay counts.

```
1 inst_1
2 inst_2
3 ...
4 transmit(x)
```

(a) Straight-line code.

```
1 if (cond_1) {...}
2 else {...}
3 if (cond_2) {...}
4 else {...}
5 ...
6 transmit(x)
```

(b) Transmitter that is independent of multiple branches.

```
1 if (i == expr)
2     x = secret;
3 else
4     x = 0;
5 transmit(x);
```

(c) Condition-dependent transmitter.

```
1 if (i == expr)
2     transmit(x);
```

(d) Transient transmitter.

```
1 for i in 1..N
2     if (i == expr)
3         x = secret;
4     else
5         x = 0;
6     transmit(x);
```

(e) Condition-dependent transmitter in a loop with an iteration-independent secret.

```
1 for i in 1..N
2     if (i == expr)
3         transmit(x);
```

(f) Transient transmitter in a loop with an iteration-independent secret.

```
1 for i in 1..N
2     if (i == expr)
3         transmit(a[i]);
```

(g) Transient transmitter in a loop with an iteration-dependent secret.

Figure 3.2: Code snippets where an attacker can use an MRA to denoise the address accessed by the *transmit* load.

The second characteristic in Table 3.2 is whether the Victim is transient. Transient instructions are speculatively-executed instructions that will never commit. MRAs can target both transient

and non-transient instructions. Transient Victims are more concerning: since the programmer or compiler do not expect their execution, they can leak a wider variety of secrets.

Figure 3.2(d) shows an example where an MRA can attack a transient instruction through branch misprediction. The transmitter should never execute, but the attacker trains the branch predictor so that it does. Figure 3.2(c) shows a related example. The transmitter should not execute using the secret, but the attacker trains the branch predictor so that it does.

The third characteristic in Table 3.2 is whether the Victim is in a loop leaking the same secret in every iteration. If it is, MRAs are more effective for two reasons. First, the attacker has more opportunities to force the re-execution of the transmitter and leak the secret. Second, since the loop is dynamically unrolled in the ROB during execution, the ROB may contain multiple instances of the transmitter, already leaking the secret multiple times. Only when a squash occurs will any MRA defense engage. We call a squash that squashes multiple transmitter instances leaking the same secret a *multi-instance* squash.

Figures 3.2(e) and (f) are like (c) and (d), but with the transmitter in a loop. In these cases, the attacker can create more executions of the transmitter by training the branch predictor so all of these branches mispredict *in every iteration*. In the worst case, the branch in the first iteration resolves after K loop iterations are loaded into the ROB and are executing. The resulting multi-instance squash may engage an MRA defense but, by that time, K transmitters have already leaked x.

Figure 3.2(g) is like (f) except that the transmitter leaks a different secret every iteration. In this case, it is easier to minimize the leakage.

3.4.2 Our Approach to Thwarting MRAs

To see how to thwart MRAs, consider Figure 3.3(a), where a squashing instruction S causes the squash of all the younger instructions in the ROB (Victims V_0 ... V_n). The idea is to detect this event and record all the Victim instructions. Then, as the Victim instructions are re-inserted into the ROB, precede each of them with a fence. We want to prevent the re-execution of each V_i until V_i cannot be squashed anymore. In this way, the attacker cannot observe the side effects of V_i more than once. The point when V_i cannot be squashed anymore is when V_i is at the head of the ROB or when neither any older instructions than V_i in the ROB or any other event (e.g., a memory consistency violation) can squash V_i. This point has been called the Visibility Point (VP) of V_i [67].

The type of fence needed is one that only prevents the execution of the V_i instruction, where V_i can be any type of transmitter instruction. When V_i reaches its VP, the fence is automatically disabled by the hardware, and V_i executes.

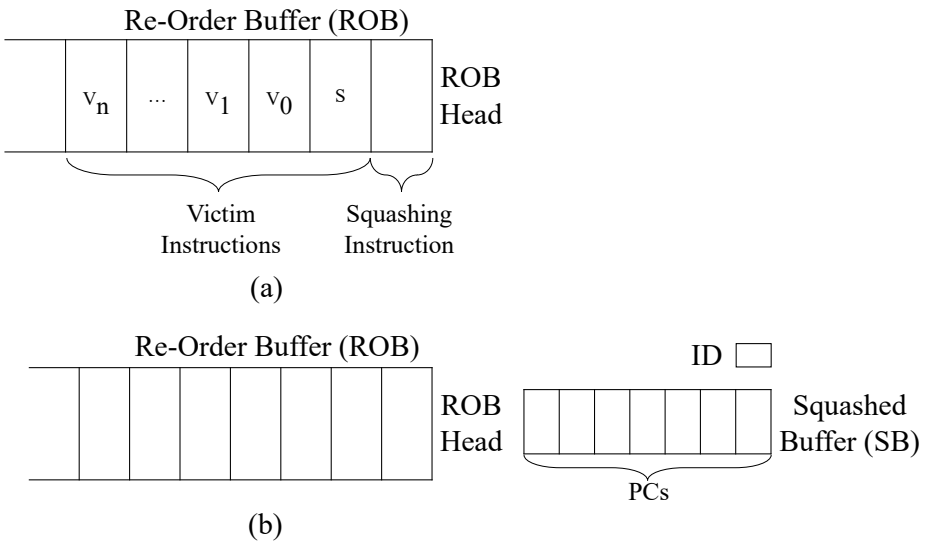


Figure 3.3: Re-Order Buffer (ROB) and Squashed Buffer (SB).

In this approach, there are two main decisions to make: (i) on a squash, how to record the Victim instructions? and (ii) for how long to keep this information? As a straw-man, consider a squash triggered by one of the *inst_i* of Figure 3.2(a). Program re-start involves re-executing the same dynamic instructions that were executed before the squash, in the exact same order. Our defense can be to record, on a squash, the Victims in a list, in the same order as they were executed; then, as each Victim is about to re-execute, precede it with a fence. When a Victim reaches its VP, we can remove it from the list. Once the list becomes empty, we can resume normal, fence-free execution.

In reality, most squashes are more complicated, especially when we have branches (Figure 3.2(b)) or execute transient instructions (Figure 3.2(d)). In these cases, program re-start may not result in the re-execution of all the saved Victims, or perhaps not in the same order as the first time. Moreover, when we have loops such as in Figure 3.2(e), the list of Victims of a squash may include multiple dynamic instances of the same static instruction – each from a different loop iteration – possibly leaking the same secret. Consequently, we will need more elaborate designs.

Finally, an attacker can potentially force a given Squashing instruction to trigger repeated back-to-back squashes. The previous chapter showed such an example, where a malicious OS repeatedly causes a page fault in the same instruction. In this case, no amount of fencing can prevent the repeated re-execution of the Squashing instruction. Hence, we suggest handling an attack on the Squashing instruction itself differently. Specifically, the hardware should not allow an instruction to trigger more than a very small number of back-to-back squashes before raising an attack alarm.

Scheme	Removal Policy	Rationale	Pros/Cons
Clear-on -Retire	When the Squashing instruction reaches its visibility point (VP)	The program makes forward progress when the Squashing instruction reaches its VP	+ Simple scheme + Most inexpensive hw - Some unfavorable security scenarios
Epoch	When an epoch completes	An epoch captures an execution locality	+ Inexpensive hardware + High security if epoch chosen well - Need compiler support
Counter	No removal, but information is compacted	Keeping the difference between squashes and retirements low minimizes leakage beyond natural program leakage	+ Conceptually simple - Intrusive hardware - May require OS changes - Some pathological cases

Table 3.3: Proposed defense schemes for microarchitectural replay attacks.

3.5 THREAT MODEL

We consider supervisor- and user-level attackers. In both cases, we assume the attacker can monitor any microarchitectural side channel (e.g., those in Section 3.2.1). This is easily realized when the attacker has supervisor-level privileges, as in the previous chapter for the SGX setting. It is also possible, subject to OS scheduler assumptions, when the attacker is unprivileged code [47]. In addition, we assume the attacker can trigger squashes in the victim program to perform MRAs. Which squashes are possible depends on the attacker. In the supervisor-level setting, the attacker can trigger fine-grain squashes due to page faults or other actions such as priming the branch predictor state. In the user-level setting, the attacker has more limited capabilities. For example, it may be able to prime predictor state [62] but cannot cause exceptions.

3.6 PROPOSED DEFENSE SCHEMES

3.6.1 Outline of the Schemes

A highly secure defense against MRAs would keep a fine-grain record of all the dynamic instructions that were squashed. When one of these instructions would later attempt to execute, the hardware would fence it and, on reaching its VP, remove it from the record. In reality, such a scheme is not practical due to storage requirements and the difficulty of identifying the same dynamic instruction. Hence, *Jamais Vu* proposes three classes of schemes that discard this state early. The schemes differ on when and how they discard the state.

The *Clear-on-Retire* scheme discards any Victim information as soon as the program makes forward progress — i.e., when the Squashing instruction reaches its VP (and hence will retire). The

Epoch discards the state when the current "execution locality" or *epoch* terminates, and execution moves to another one. Finally, the *Counter* scheme keeps the state forever, but it compresses it so that all instances of the same static instruction keep their state merged. Each of these policies to discard or compress state creates a different attack surface.

3.6.2 Clear-on-Retire Scheme

The rationale for the simple *Clear-on-Retire* scheme is that an MRA obtains information by stalling a program's forward progress and repeatedly re-executing the same set of instructions. Hence, when an MRA defense manages to force forward progress, it is appropriate to discard the record of Victim instructions. Hence, *Clear-on-Retire* clears the Victim state when the Squashing instruction reaches its VP.

Clear-on-Retire stores information about the Victim instructions in a buffer associated with the ROB called the *Squashed Buffer (SB)*. Figure 3.3(b) shows a conceptual view of the SB. It includes a *PC Buffer* with the set of program counters (PCs) of the instructions that were squashed. It also includes an identifier register (*ID*) which contains the ROB index of the Squashing instruction – i.e., the one that caused the squash. Since a squash may flush multiple iterations of a loop in the ROB, the SB may contain the same PC multiple times.

Multiple instructions in the ROB may cause squashes, in any order. For example, in Figure 3.2(b), the branch in Line 3 may cause a squash, and then the one in Line 1 may. At every squash, the Victims' PCs are added to the PC Buffer. However, ID is only updated if the Squashing instruction is *older* than the one currently in ID. This is because instruction retirement is in order.

The *Clear-on-Retire* algorithm is as follows. On a squash, the PCs of the Victims are added to the PC Buffer and the ID is updated if necessary. When trying to insert an instruction I in the ROB, if I is in the PC Buffer, a fence is placed before I. When the instruction in the ID reaches its VP, since the program is making forward progress, the SB is cleared and all the fences introduced by *Clear-on-Retire* are nullified.

The first row of Table 3.3 describes *Clear-on-Retire*. The scheme is simple and has the most inexpensive hardware. The SB can be implemented as a simple Bloom filter (Section 3.7.1).

One shortcoming of *Clear-on-Retire* is that it has some unfavorable security scenarios. Specifically, the attacker could choose to make slow forward progress toward the transmitter V, forcing every single instruction to be a Squashing one.

In practice, this scenario may be hard to setup, as the squashes have to occur in order, from older to younger predecessor of I. Indeed, if a Squashing instruction S_1 squashes V, and V is then re-inserted into the ROB with a fence, a second Squashing instruction S_2 older than S_1 cannot squash V's execution again. The reason is that V is fenced and has not executed.

3.6.3 Epoch Scheme

The rationale for the *Epoch* scheme is that an MRA operates on an "execution locality" of a program, which has a certain combination of instructions. Once execution moves to another locality, the re-execution and squash of the original Victims is not seen as dangerous. Hence, it is appropriate to discard the record of Victim instructions when moving to another locality. We refer to an execution locality as an *epoch*. Possible epochs are a loop iteration, a whole loop, or a subroutine.

Like *Clear-on-Retire*, *Epoch* uses an SB to store the PC of the Victim instructions. However, the design is a bit different. First, *Epoch* requires that the hardware finds *start-of-epoch* markers as it inserts instructions into the ROB. We envision that such markers are added by the compiler. Second, the SB needs one {ID, PC-buffer} pair for each in-progress epoch. The ID now stores a small-sized, monotonically-increasing epoch identifier; the PC buffer stores the PCs of the Victims squashed in that epoch.

The *Epoch* algorithm works as follows. As instructions are inserted into the ROB, the hardware records every start-of-epoch marker. On a squash, the Victim PCs are separated based on the epoch they belong to and added to different PC buffers. The IDs of the PC buffers are set to the corresponding epoch IDs. A given PC may be in different PC buffers and even multiple times in the same PC buffer. When trying to insert an instruction I in the ROB, if I is in the PC buffer of the current epoch, I is fenced. Finally, when the first instruction of an epoch reaches its VP, the hardware clears the {ID, PC-buffer} pairs of all the earlier epochs.

The second row of Table 3.3 describes *Epoch*. The scheme is also simple and has inexpensive hardware. It can also implement the PC buffers as Bloom filters. *Epoch* also has high security if epochs are chosen appropriately, as the Victim information remains for the duration of the epoch. A drawback of *Epoch* is that it needs compiler support.

An epoch can be long, in which case its PC buffer may contain too many PCs to operate efficiently. Hence, our preferred implementation of this scheme is a variation of *Epoch* called *Epoch-Rem* that admits PC removal. Specifically, when a Victim from an epoch reaches its VP, the hardware removes its PC from the corresponding PC buffer. This support reduces the pressure on the PC buffer. This functionality is supported by implementing the PC buffers as *counting* Bloom filters.

3.6.4 Counter Scheme

The *Counter* scheme never discards information about Victim squashes. However, to be amenable to implementation, the scheme merges the information on all the dynamic instances of the same

static instruction into a single variable. *Counter*'s goal is to keep the count of squashes and the count of retirements of any given static instruction close to each other. The rationale is that, if both counts are similar, an MRA is unlikely to exfiltrate much more information than what the program naturally leaks.

While *Counter* can be implemented similarly as the two previous schemes, a more intuitive implementation associates Victim information with each static instruction. A simple design adds a Squashed bit to each static instruction I. When I gets squashed, its Squashed bit is set. From then on, an attempt to insert I in the ROB causes a fence to be placed before I. When I reaches its VP, the bit is reset. After that, a future invocation of I is allowed to execute with no fence.

In reality, multiple dynamic instances of the same static instruction may be in the ROB at the same time and get squashed together. Hence, we propose to use a *Squashed Counter* per static instruction rather than a bit. The algorithm works as follows. Every time that instances of the instruction get squashed, the counter increases by the number of instances; every time that an instance reaches its VP, the counter is decremented by one. The counter does not go below zero. Finally, when an instruction is inserted in the ROB, if its counter is not zero, the hardware fences it.

To reduce the number of stalls, a variation of this scheme could allow a Victim to execute speculatively as long as its counter is lower than a threshold.

The third row of Table 3.3 describes *Counter*. The scheme is conceptually simple. However, it requires somewhat intrusive hardware. One possible design requires counters that are stored in memory and, for the most popular instructions, get cached into a special cache next to the I-L1 (Section 3.7.3). This counter cache or the memory needs to be updated every time a counter changes. In addition, the OS needs changes to allocate and manage pages of counters for the instructions (Section 3.7.3).

Counter has some pathological patterns. Specifically, an attacker may be able to repeatedly squash an instruction by interleaving the squashes with retirements of the same static instruction. In this case, one access leaks a secret before being squashed; the other access is benign, retires, and decreases the counter. This pattern is shown in Figure 3.2(e). In every iteration, the branch predictor incorrectly predicts the condition to be true, x is set to *secret*, and the transmitter leaks x. The execution is immediately squashed, the *else* code executes, and the transmitter retires. This process is repeated in every iteration, causing the counter to toggle between one and zero.

3.6.5 Relative Analysis of the Security of the Schemes

To assess the relative security of the schemes, we compare their worst-case leakage for each of the code snippets in Figure 3.2. We measure leakage as the number of executions of the transmitter

for a given secret. We report Transient Leakage (TL) (when transmitter is a transient instruction) and Non-Transient Leakage (NTL). The summary is shown in Table 3.4. In the table, for the *Epoch* schemes, we show the leakage without removal (*NR*) and with removal (*R*) of entries.

Case	**Non-Transient Leakage**	**Transient Leakage**					
		Clear-on-Retire	**Epoch**				**Cntr**
			Iter		**Loop**		
			NR	R	NR	R	
(a)	1	ROB-1	1				1
(b)	1	ROB-1	1				1
(e)	0	K*N	N	N	K	N	N
(f)	0	K*N	N	N	K	K	K
(g)	0	K	1	1	1	1	1

Table 3.4: Worst-case leakage count in the proposed defense schemes for some of the examples in Figure 3.2. For a loop, N is the number of iterations and, as the loop dynamically unrolls in the ROB, K is the number of iterations that fit in the ROB.

In Figure 3.2(a), since the transmitter should execute once, the NTL is one. The TL is found as follows. In *Clear-on-Retire*, the attacker could make each instruction before the transmitter a Squashing one. In the worst-case, the squashes occur in program order, and the timing is such that the transmitter is squashed as many times as the ROB size minus one. Hence TL is ROB size minus 1. While this is a large number, it is smaller than the leakage in the MicroScope attack of the previous chapter, where TL is infinite because one instruction can cause repeated squashes. In all *Epoch* designs, the transmitter is squashed only once. Hence, TL is 1. *Counter* sets the transmitter's counter to 1 on the first squash; no other speculative re-execution is allowed. Hence, TL is 1.

Figure 3.2(b) is conceptually like (a). The NTL in all schemes is 1. The TL of *Counter* and *Epoch* is 1. In *Clear-on-Retire*, in the worst-case where all the branches are mispredicted and resolve in program order, the TL is equal to the number of branches that fit in the ROB minus one slot (for the transmitter). For simplicity, we say that TL is the ROB size minus one.

Figures 3.2(c) and (d) are not interesting. NTL is 0 (since in Figure 3.2(c) x is never set to the secret in a non-speculative execution) and TL is 1 for all schemes.

In Figure 3.2(e), NTL is zero. However, the attacker ensures that the branch is mispredicted in every iteration. To assess the worst-case TL in *Clear-on-Retire*, assume that, as the N-iteration loop dynamically unrolls in the ROB, K iterations fit in the ROB. In this case, the worst-case is that each iteration (beyond the first $K-1$ ones) is squashed K times. Hence, TL in *Clear-on-Retire* is $K*N$. In *Epoch* with iteration, since each epoch allows one squash, the TL is N—with and without PC removal. In *Epoch* with loop without removal, in the worst-case, the initial K iterations are in the ROB when the squash occurs, and we have a *multi-instance* squash (Section 3.4.1). Hence, the

TL is K. In *Epoch* with loop with removal, since every retirement of the transmitter removes the transmitter PC from the SB, TL is N. Finally, in *Counter*, since every iteration triggers a squash and then a retirement, TL is N.

Figure 3.2(f) is like Figure 3.2(e), except that the transmitter will never retire. As a consequence, *Epoch* with loop with removal does not remove it from the SB, and *Counter* does not decrease the counter. Hence, their TL is K.

Finally, Figure 3.2(g) is like 3.2(f) but each iteration accesses a different secret. The NTL is zero. The TL for *Clear-on-Retire* is K because of the dynamic unrolling of iterations in the ROB. For the other schemes, it is 1 in the worst-case.

Overall, for the examples shown in Table 3.4, *Epoch* at the right granularity (i.e., loop level) without removal has the lowest leakage. With removal, the scheme is similar to *Counter*, and better than *Epoch* with iteration. *Clear-on-Retire* has the highest worse-case leakage.

A similar analysis can be done for loop-based code samples where NTL is larger than 1. The same conclusions stand.

3.7 MICROARCHITECTURAL DESIGN

3.7.1 Implementing *Clear-on-Retire*

The PC Buffer in the SB needs to support three operations. First, on a squash, the PCs of all the Victims are inserted in the PC Buffer. Second, before an instruction is inserted in the ROB, the PC Buffer is queried to see if it contains the instruction's PC. Third, when the instruction in the ID reaches its VP, the PC Buffer is cleared.

These operations are easily supported with a hardware Bloom filter [95]. Figure 3.4 shows the filter's structure. It is an array of M entries, each with a single bit. To insert an item in the filter (BF), the instruction's PC is hashed with n hash functions (H_i) and n bits get set: $BF[H_1]$, $BF[H_2]$, ... $BF[H_n]$. A high-performance implementation of this functionality uses an n-port direct mapped cache of M 1-bit entries.

A Bloom filter can have false positives but no false negatives. A false positive occurs when a PC is not in the PC Buffer but it is deemed to be there due to a conflict. This situation is safe, as it means that *Clear-on-Retire* will end up putting a fence before an instruction that does not need it.

In practice, if we size the filter appropriately, we do not see many false positives when running benign programs. Specifically, as we will see in Section 3.10.3, for a 192-entry ROB, a filter with 1232 bits and 7 hash functions has less than 3.6% false positives. An attacker can create more conflicts, but they will only cause more fencing.

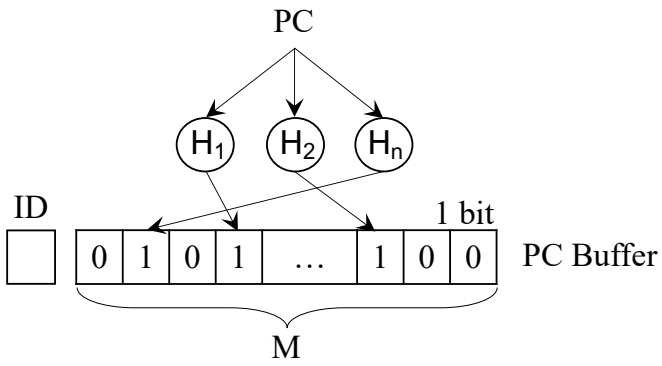


Figure 3.4: SB with a PC Buffer organized as a Bloom filter.

3.7.2 Implementing *Epoch*

The SB for *Epoch* is like the one for *Clear-on-Retire* with two differences. First, there are multiple {ID, PC-buffer} pairs – one for each in-progress epoch. Second, in *Epoch-Rem*, which supports the removal of individual PCs from a PC buffer, each PC buffer is a counting Bloom filter [96].

A counting Bloom filter is shown in Figure 3.5. It is like a plain filter except that each entry has k bits. To insert an item in the filter, the n entries selected by the hashes are incremented by one – i.e., $BF[H_1]$++, $BF[H_2]$++, ... $BF[H_n]$++. To remove the item, the same entries are decremented by one. An n-port direct mapped cache of M k-bit entries is used.

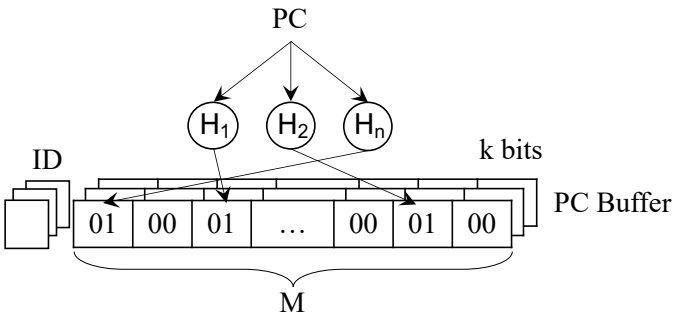


Figure 3.5: SB with multiple PC buffers organized as counting Bloom filters.

A counting Bloom filter can suffer false positives, which in our case are harmless. In addition, it can also suffer false negatives. A false negative occurs when a Victim should be in the PC Buffer but it is deemed not to. In *Jamais Vu*, they are caused in two ways. The first one is when a non-Victim instruction *NV* to be inserted in the ROB is believed to be in the filter because it conflicts with existing entries in the filter. When *NV* reaches its VP, it removes state from Victim instructions from the filter. Later, when such Victims are checked for membership, they are not

found. The second case is when the filter does not have enough bits per entry and, as a new Victim is inserted, an entry saturates. In this case, information is lost. Later, Victim state that should be in the filter will not be found in the filter.

False negatives reduce security because no fence is inserted where there should be one. In practice, as we will see in Section 3.10.3, only 0.06% or fewer of the accesses are false negatives. Furthermore, the randomness introduced by the hashes makes it very hard for the attacker to force the replay of a specific instruction.

Handling Epoch Overflow.

The SB has a limited number of {ID, PC-buffer} pairs. Therefore, it is possible that, on a squash, the Victim instructions belong to more epochs than PC buffers exist in the SB. For this reason, *Epoch* augments the SB with one extra ID not associated with any PC buffer called *OverflowID*. To understand how it works, recall that epoch IDs are monotonically increasing. Hence, we may find that Victims from a set of high-numbered epochs have no PC buffer to go. In this case, we store the ID of the highest-numbered epoch in *OverflowID*. From now on, when a new instruction is inserted in the ROB, if it belongs to an epoch whose ID: (i) owns no PC buffer and (ii) is no higher than the one in *OverflowID*, we place a fence before the instruction. The reason is that, since we have lost information about that epoch, we do not know whether the instruction is a Victim. When the epoch whose ID is in *OverflowID* is fully retired, *OverflowID* is cleared.

As an example, consider Figure 3.6(a), which shows a ROB full of instructions. The figure groups the instructions according to their epoch and labels the group with the epoch ID. Assume that all of these instructions are squashed and that the SB only has four {ID, PC-buffer} pairs. Figure 3.6(b) shows the resulting assignment of epochs to {ID, PC-buffer} pairs. Epochs 14 and 15 overflow and, therefore, *OverflowID* is set to 15. Any future insertion in the ROB of an instruction from epochs 14 and 15 will be preceded by a fence. Eventually, some {ID, PC-buffer} pairs will free up and may be used by other epochs such as 16. However, all instructions from 14 and 15 will always be fenced.

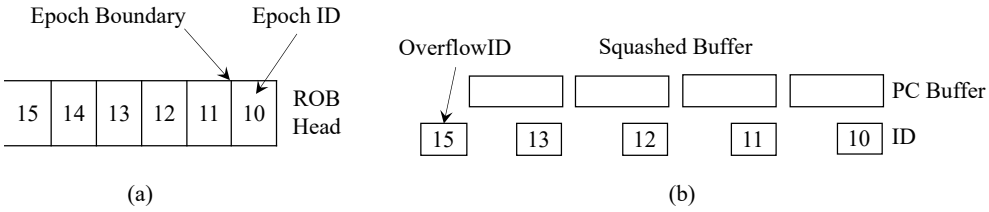


Figure 3.6: Handling epoch overflow.

3.7.3 Implementing *Counter*

To implement *Counter*, *Jamais Vu* stores the counters for all the instruction in data pages, and the core has a small *Counter Cache* (CC) that keeps the recently-used counters close to the pipeline for easy access. Since the most frequently-executed instructions are in loops, a small CC typically captures the majority of counters needed.

We propose a simple design where, for each page of code, there is a data page at a fixed Virtual Address (VA) *Offset* that holds the counters of the instructions in that page of code. Further, the VA offset between each instruction and its counter is fixed. In effect, this design increases the memory consumed by a program by the size of its instruction page working set

Figure 3.7(a) shows a page of code and its page of counters at a fixed VA offset. When the former is brought into physical memory, the latter is also brought in. The figure shows a line with several instructions and the line with their counters. We envision each counter to be 4 bits.

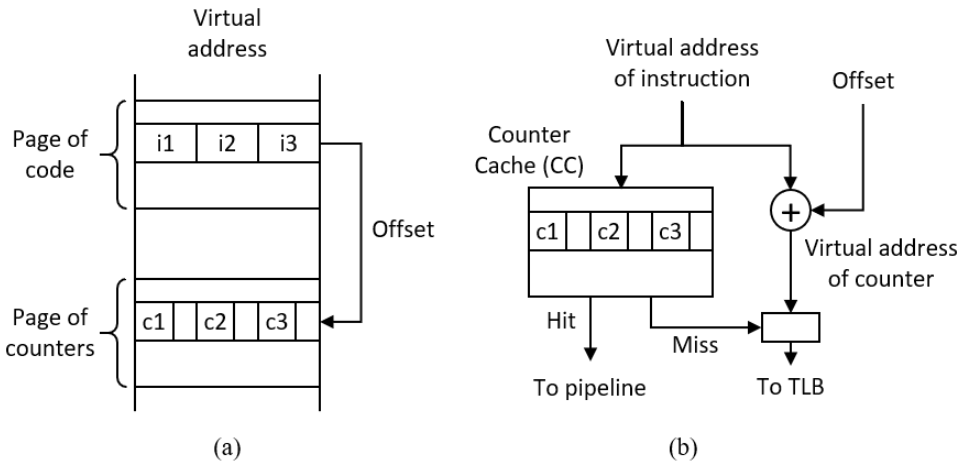


Figure 3.7: Allocating and caching instruction counters.

Figure 3.7(b) shows the action taken when an instruction is about to be inserted in the ROB. The VA of the instruction is sent to the CC, which is a small, set-associative cache that contains the most recently-used lines of counters. Due to good instruction locality, the CC hits most of the time. On a hit, the corresponding counter is sent to the pipeline.

If, instead, the CC misses, a *CounterPending* signal is sent to the pipeline. Further, the *Offset* is added to the instruction's VA to obtain the VA of the counter. This address is sent to the TLB to obtain the Physical Address (PA). After that, *but only when the corresponding instruction reaches its Visibility Point (VP)*, a request will be sent to the cache hierarchy to obtain the line of counters, store it in the CC, and pass the counter to the pipeline. We will see later why this delay is necessary.

The actions in the pipeline are as follows. If the counter is not zero or a *CounterPending* signal is received, two actions are taken. First, a fence is inserted in the ROB before the instruction. Second,

when the instruction reaches its VP, the counter is decremented and stored back in the CC or, if a CounterPending signal was received, the request mentioned above is sent to the cache hierarchy to obtain the counter. When the counter is returned, if it is not zero, the counter is decremented and stored back in the CC.

We delay this request for the counter until the instruction reaches its VP because we want the CC to add no side channels. Specifically, on an CC miss, no request can be sent to the cache hierarchy to get counters until the instruction reaches its VP. The goal is to provide no speculative information to the attacker. In addition, on CC hit, the CC's LRU bits are not updated until the instruction reaches its VP.

3.7.4 Handling Context Switches

To operate correctly, *Jamais Vu* performs the following actions at context switches. In *Clear-on-Retire* and *Epoch*, the SB state is saved to and restored from memory as part of the context. This enables the defense to remember the state when execution resumes. In *Counter*, the CC is flushed to memory to leave no traces behind that could potentially lead to a side-channel exploitable by the newly scheduled process. The new process loads the CC on demand.

3.8 COMPILER PASS

Epoch includes a program analysis pass that places "start-of-epoch" markers in a program. The pass accepts as input a program in source code or binary. Source code is preferred, since it contains more information and allows a better analysis.

We consider two designs: one that uses loops as epochs and one that uses loop iterations as epochs. In the former, an epoch includes the instructions between the beginning and the end of a loop, or between the end of a loop and the beginning of the next loop; in the latter, an epoch includes the instructions between the beginning and the end of an iteration, or between the end of the last iteration in a loop and the beginning of the first iteration in the next loop.

The analysis is intra-procedural and uses conventional control flow compiler techniques [97]. It searches for back edges in the control flow of each function, and from there identifies the natural loops. Once back edges and loops are identified, the *Epoch* compiler inserts the epoch boundary markers.

To mark an x86 program, our compiler places a previously-ignored instruction prefix in front of every first instruction in an epoch. This approach has been used by Intel for lock elision [98]. The processor ignores this prefix, and our simulator recognizes that a new epoch starts. This approach

changes the executable, but because current processors ignore this prefix, the new executable runs on any x86 machine. The size of the executable increases by only 1 byte for every static epoch.

Finally, in both *Epoch* designs, procedure calls and returns are also epoch boundaries. For these, the compiler does not need to mark anything. The hardware recognizes the x86 procedure call and return instructions and starts a new epoch.

3.9 EXPERIMENTAL SETUP

Architectures Modeled. We model the architecture shown in Table 3.5 using cycle-level simulations with Gem5 [99]. All the side effects of instructions are modeled. The baseline architecture is called UNSAFE, because it has no protection against MRAs. The defense schemes are: (i) *Clear-on-Retire* (COR), (ii) *Epoch* with iteration (EPOCH-ITER), (iii) *Epoch-Rem* with iteration (EPOCH-ITER-REM), (iv) *Epoch* with loop (EPOCH-LOOP), (v) *Epoch-Rem* with loop (EPOCH-LOOP-REM), and (vi) *Counter* (COUNTER).

From the table, we can compute the sizes of the *Jamais Vu* hardware structures. *Clear-on-Retire* uses 1 non-counting Bloom filter. The size is 1232 bits. *Epoch* uses 12 counting Bloom filters. The total size is 12 times 4,928 bits, or slightly above 7KB. The Counter Cache (CC) in *Counter* contains 128 entries, each with the counters of an I-cache line. Since the shortest x86 instruction is 1 byte and a counter is 4 bits, each line in the CC is shifted 4 bits every byte, compacting the line into 32B. Hence, the CC size is 4KB. We used Cacti [100] for 22nm to obtain the area, dynamic energy of a read, and leakage power of these structures. Table 3.5 shows the results.

Application and Analysis Pass. We run SPEC17 [101] with the reference input size. Because of simulation issue with Gem5, we exclude 2 applications out of 23 from SPEC17. For each application, we use SimPoint [102] methodology to generate up to 10 representative intervals that accurately characterize the end-to-end performance. Each interval consists of 50 million instructions. We run Gem5 on each interval with syscall emulation with 1M warm-up instructions.

Our compiler pass is implemented on top of Radare2 [103], a state-of-the-art open-source binary analysis tool. We extend Radare2 to perform epoch analysis on x86 binaries.

3.10 EVALUATION

3.10.1 Thwarting an MRA Proof-of-Concept (PoC) Attack

To demonstrate *Jamais Vu*'s ability to thwart MRAs, we implement a PoC MRA on gem5 similar to the port contention attack in the previous chapter. Based on a secret, the victim thread performs

Parameter	Value
Architecture	2.0 GHz out-of-order x86 core
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, TAGE branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
L1-D Cache	64 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
L2 Cache	2 MB, 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
Counter Cache	32 sets, 4-way, 2 cycle RT latency, 4b/counter Area: 0.053 mm^2; Dyn. rd energy: 14.3pJ; Leak. power: 3.14mW
Bloom Filter	1232 entries, 7 hash functions. Non-counting: 1b/entry. Counting: 4b/entry Counting: Area: 0.009 mm^2; Dyn. rd energy: 0.97pJ; Leak. power: 1.72mW
{ID, PC-buffer}	12 pairs

Table 3.5: Parameters of the simulated architecture.

a division rather than a multiplication operation. The attacker picks 10 Squashing instructions that precede the test on the secret and causes 5 squashes on each in turn. The code is similar to Figure 3.2(a). We measure that the attacker causes 50 replays of the division operation on UNSAFE, 10 replays on *Clear-on-Retire*, and only 1 replay on *Epoch* and *Counter*.

3.10.2 Execution Time

Figure 3.8 shows the normalized execution time of SPEC17 applications on all configurations but *Epoch* without removals, which we consider later. Time is normalized to UNSAFE.

Jamais Vu proposes several designs that represent different performance, security, and implementation complexity trade-offs. Among all the schemes, COR has the lowest execution time

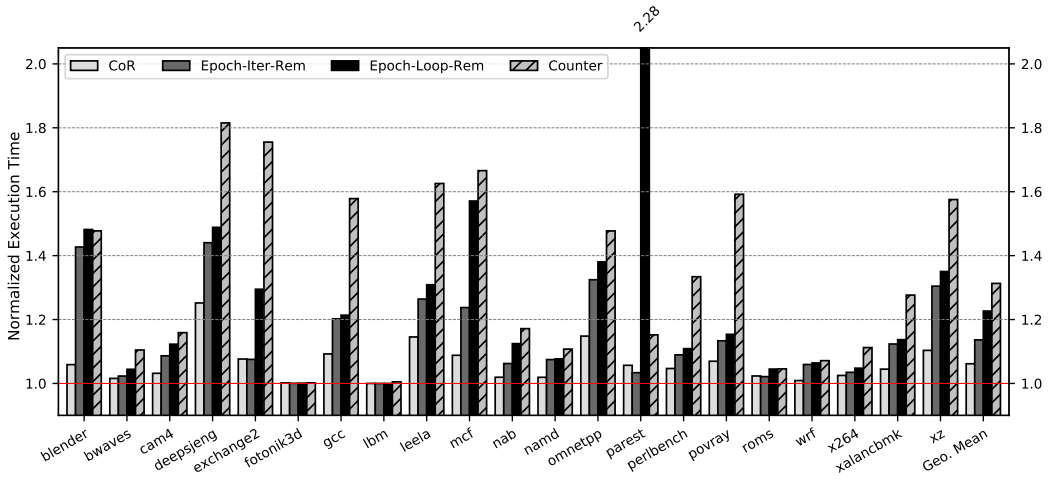


Figure 3.8: Normalized execution time for all the schemes except *Epoch* without removals.

overhead. It incurs only a geometric mean overhead of 6.2% over UNSAFE. It is also the simplest but least secure design (Table 3.4). EPOCH-ITER-REM has the next lowest average execution overhead, namely 13.6%. This design is also very simple and is more secure, especially as we will see that false negatives are very rare. The next design, EPOCH-LOOP-REM, has higher average execution time overhead, namely 22.7%. However, it has simple hardware and is one of the two most secure designs (Table 3.4)—again, given that, as we will see, false negatives are very rare. Finally, COUNTER has the highest average execution overhead, namely 31.3%. It is one of the two most secure schemes, but the implementation proposed is not as simple as the other schemes.

The schemes not shown in the figure, namely EPOCH-ITER and EPOCH-LOOP are not competitive. They have an average execution overhead of 27.2% and 69.4%, respectively. These are substantial increases over the schemes with removals, with modest gains in simplicity and security.

3.10.3 Sensitivity Study

Each *Jamais Vu* design has several architectural parameters that set its hardware requirements and efficiency. Recall that COR uses a Bloom filter, while EPOCH-ITER-REM and EPOCH-LOOP-REM use counting Bloom filters. To better understand the different Bloom filters, we first perform a sensitivity study of their parameters. Then, we evaluate several Counter Cache organizations for COUNTER.

Number of Bloom Filter Entries. Figure 3.9 shows the geometric mean of the normalized execution time and the false positive rates on SPEC17, when varying the size of the Bloom filter. We consider several sizes, which we measure in number of entries. Recall that each entry is 1 bit for COR and 4 bits for the other schemes. We pick each of these number of entries by first selecting a

projected element count (the number in parenthesis) and running an optimization pass [104] for a target false positive probability of 0.01. From the figure, we see that a Bloom filter of 1232 entries strikes a good balance between execution and area overhead, with a false positive rate of less than 3.6% for all schemes.

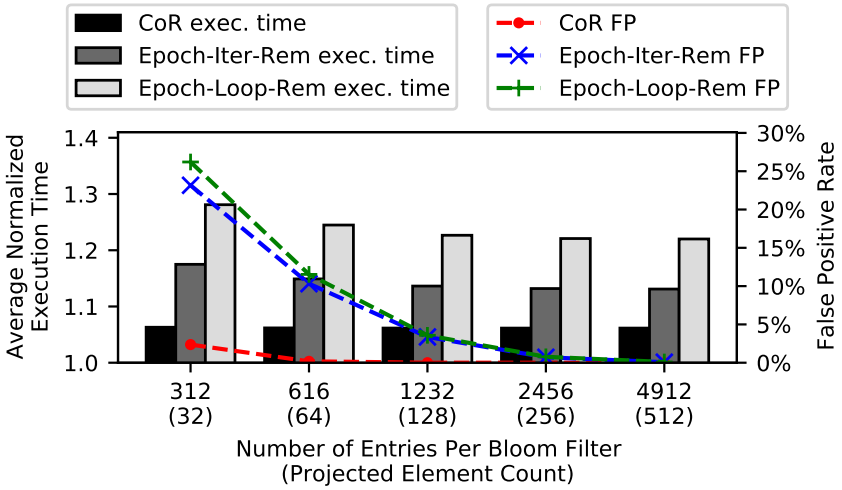


Figure 3.9: Average normalized execution time and false positive rate when varying the number of entries per Bloom filter.

Number of {ID, PC-buffer} Pairs. Another design decision for EPOCH-ITER-REM and EPOCH-LOOP-REM is how many {ID, PC-buffer} pairs to have. If they are too few, overflow will be common. Figure 3.10 shows the average normalized execution time and the overflow rates on SPEC17, when varying the number of {ID, PC-buffer} pairs. The overflow rate is the fraction of insertions into a PC buffer that overflow. From the figure, we see that, as the number of {ID, PC-buffer} pairs decreases, the execution time and overflow rates increase. Supporting 12 {ID, PC-buffer} pairs is a good design point.

Number of Bits Per Counting Bloom Filter Entry. The counting Bloom filters in EPOCH-ITER-REM and EPOCH-LOOP-REM use a few bits per entry to keep the count. Figure 3.11 shows the average normalized execution time and the false negative rates on SPEC17, when varying the number of bits per entry. False negatives can be caused either by conflicts in the filter or by not having enough bits in an entry. In the latter case, when the counter in the entry saturates, it cannot record further squashes and information is lost. We see from the figure that the number of bits per counter has little impact on the performance. However, the number of bits has to be 4 or more to have a false negative rate of 0.06% or less.

Counter Cache (CC) Geometry. Figure 3.12 shows the CC hit rate as we vary the ways and sets of the CC. The cache hit rate increases with the number of entries, but changing the associativity

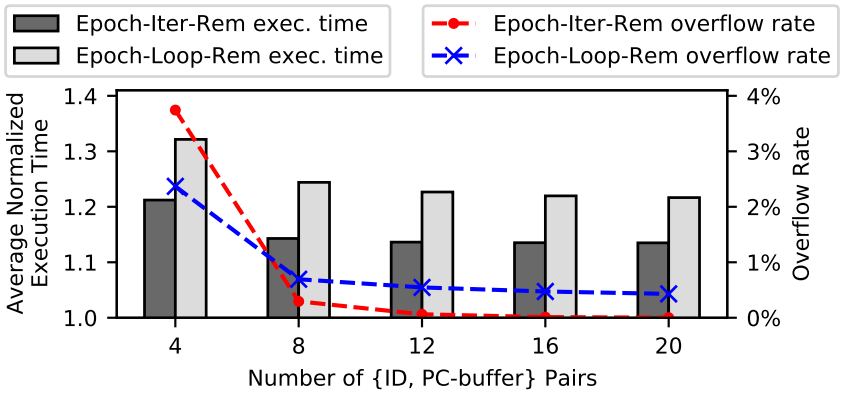


Figure 3.10: Average normalized execution time and overflow rate when varying the number of {ID, PC-buffer} pairs.

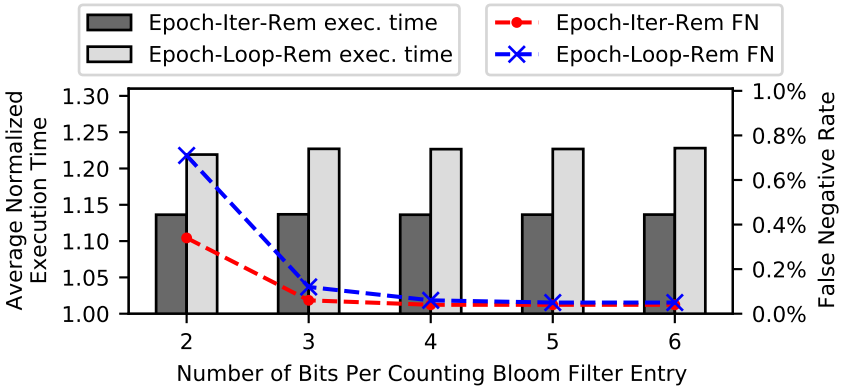


Figure 3.11: Average normalized execution time and false negative rate when varying the number of bits per counting Bloom filter entry.

of the CC from 4 to full does not help. Overall, our default configuration of 32 sets and 4 ways performs well. It attains a high hit rate while a larger cache improves it only a little. A smaller cache hurts the hit rate substantially.

3.11 RELATED WORK

We now discuss prior works related to mitigating MRAs.

Preventing Pipeline Squashes. The literature features several solutions that can mitigate specific instances of MRAs. For example, page fault protection schemes [64, 105, 106, 107] can be used to mitigate MRAs that rely on page faults to cause pipeline squashes. The goal of these countermeasures is to block controlled-channel attacks [35, 36] by terminating victim execution when

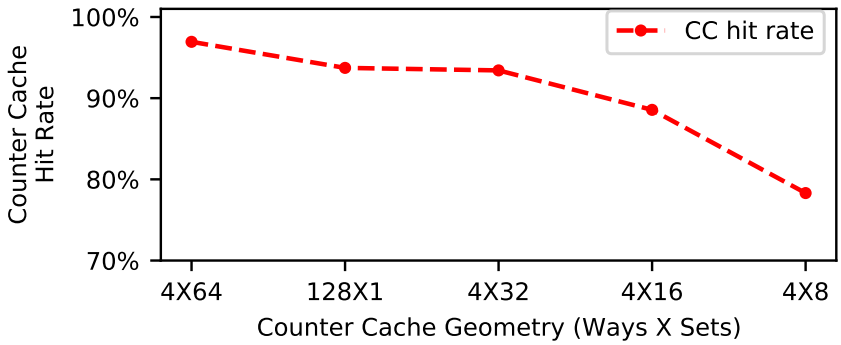


Figure 3.12: CC hit rate when varying the cache geometry.

an OS-induced page fault is detected. The most recent of these defenses, Autarky [105], achieves this through a hardware/software co-design that delegates paging decisions to the enclave. However, attacks that rely on events other than page faults to trigger pipeline squashes (Section 3.4.1) would still overcome these point mitigation strategies. In contrast, *Jamais Vu* is the first comprehensive defense that addresses the root cause of MRAs, namely that instructions can be forced to execute more than once.

Preventing Side Channel Leakage. Another strategy to mitigate MRAs is to prevent speculative instructions from leaking data through side channels. For example, several works have proposed to mitigate side channels by isolating or partitioning microarchitectural resources [106, 108, 109, 110, 111, 112, 113, 114, 115], thus preventing the attacker from accessing them during the victim's process execution. These defenses prevent adversaries from leaking data through specific side channels, which ultimately makes MRAs's ability to denoise these channels less useful. In practice, however, no holistic solution exists that can block all side channels. Further, new adversarial applications of MRAs may be discovered that go beyond denoising side channel attacks.

3.12 CONCLUSION

This chapter presented *Jamais Vu*, the first technique to thwart MRAs. *Jamais Vu* detects when an instruction is squashed and, as it is re-inserted into the ROB, places a fence before it. The three main *Jamais Vu* designs are *Clear-on-Retire*, *Epoch*, and *Counter*, which select different trade-offs between execution overhead, security, and complexity. Our preferred design, called *Epoch-Rem* with loop, can effectively mitigate MRAs, has an average runtime overhead of 22.7% in benign executions, and only needs hardware counting Bloom filters.

Chapter 4: Operating System and Architectural Support for System Call Security

4.1 INTRODUCTION

Protecting the OS kernel is a significant concern, given its capabilities and shared nature. In recent years, there have been reports of a number of security attacks on the OS kernel through system call vectors [116, 117, 118, 119, 120, 121, 122, 123, 124, 125].

A popular technique to protect OS kernels against untrusted user processes is *system call checking*. The idea is to limit the system calls that a given process can invoke at runtime, as well as the actual set of argument values used by the system calls. This technique is implemented by adding code at the OS entry point of a system call. The code compares the incoming system call against a list of allowed system calls and argument set values, and either lets the system call continue or flags an error. All modern OSes provide kernel support for system call checking, such as Seccomp for Linux [126], Pledge [127] and Tame [128] for OpenBSD, and System Call Disable Policy for Windows [129].

Linux's Seccomp (Secure Computing) module is the most widely-used implementation of system call checking. It is used in a wide variety of today's systems, ranging from mobile systems to web browsers, and to containers massively deployed in cloud and data centers. Today, every Android app is isolated using Seccomp-based system call checking [130]. Systemd, which is Linux's init system, uses Seccomp to support user process sandboxing [131]. Low-overhead virtualization technologies such as Docker [132], LXC/LXD [133], Google's gVisor [134], Amazon's Firecraker [135, 136], CoreOS rkt [137], Singularity [138], Kubernetes [139], and Mesos Containerizer [140] all use Seccomp. Further, Google's recent Sandboxed API project [141] uses Seccomp to enforce sandboxing for C/C++ libraries. Overall, Seccomp provides "*the most important security isolation boundary*" for containerized environments [136].

Unfortunately, checking system calls incurs overhead. Oracle identified large Seccomp programs as a root cause that slowed down their customers' applications [142, 143]. Seccomp programs can be application specific, and complex applications tend to need large Seccomp programs. Recently, a number of Seccomp overhead measurements have been reported [142, 143, 144]. For example, a micro benchmark that repeatedly calls `getppid` runs 25% slower when Seccomp is enabled [144]. Seccomp's overhead on ARM processors is reported to be around 20% for simple checks [145].

The overhead becomes higher if the checks include system call argument values. Since each individual system call can have multiple arguments, and each argument can take multiple distinct values, comprehensively checking arguments is slow. For example, Kim and Zeldovich [146] show

that Seccomp causes a 45% overhead in a sandboxed application. Our own measurements on an Intel Xeon server show that the average execution time of macro and micro benchmarks is 1.14× and 1.25× higher, respectively, than without any checks. For this reason, current systems tend to perform only a small number of argument checks, despite being well known that systematically checking arguments is critical for security [147, 148, 149].

The overhead of system call and argument value checking is especially concerning for applications with high-performance requirements, such as containerized environments. The overhead diminishes one of the key attractions of containerization, namely lightweight virtualization.

To minimize this overhead, this chapter proposes *Draco*, a new architecture that *caches* system call IDs and argument set values after they have been checked and validated. System calls are first looked-up in a special cache and, on a hit, skip the checks. The insight behind Draco is that the patterns of system calls in real-world applications have locality—the same system calls are issued repeatedly, with the same sets of argument values.

In this chapter, we present both a software and a hardware implementation of Draco. We build the software one as a component of the Linux kernel. While this implementation is faster than Seccomp, it still incurs substantial overhead.

The hardware implementation of Draco introduces novel microarchitecture to eliminate practically all of the checking overhead. It introduces the *System Call Lookaside Buffer* (SLB) to keep recently-validated system calls, and the *System Call Target Buffer* (STB) to preload the SLB in advance.

In our evaluation, we execute representative workloads in Docker containers. We run Seccomp on an Intel Xeon server, checking both system call IDs and argument values. We find that, with Seccomp, the average execution time of macro and micro benchmarks is 1.14× and 1.25× higher, respectively, than without performing any security checks. With our software Draco, the average execution time of macro and micro benchmarks reduces to 1.10× and 1.18× higher, respectively, than without any security checks. Finally, we use full-system simulation to evaluate our hardware Draco. The average execution time of the macro and micro benchmarks is within 1% of a system that performs no security checks.

With more complex checks, as expected in the future, the overhead of conventional Seccomp checking increases substantially, while the overhead of Draco's software implementation goes up only modestly. Moreover, the overhead of Draco's hardware implementation remains within 1% of a system without checks.

Contributions. This chapter makes the following contributions.

- A characterization of the system call checking overhead for various Seccomp configurations.

- The new Draco architecture that caches validated system call IDs and argument values for reuse. We introduce a software and a hardware implementation.
- An evaluation of the software and hardware implementations of Draco.

4.2 BACKGROUND

4.2.1 System Calls

System calls are the interface an OS kernel exposes to the user space. In x86-64 [98], a user space process requests a system call by issuing the `syscall` instruction, which transfers control to the OS kernel. The instruction invokes a system call handler at privilege level 0. In Linux, `syscall` supports from zero to six distinct arguments that are passed to the kernel through general-purpose registers. Specifically, the x86-64 architecture stores the system call ID in the `rax` register, and the arguments in registers `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`. The return value of the system call is stored in the `rax` register. The `syscall` instruction is a serializing instruction [98], which means that it cannot execute until all the older instructions are completed, and that it prevents the execution of all the younger instructions until it completes. Finally, note that the work in this chapter is not tied to Linux, but applies to different OS kernels.

4.2.2 System Call Checking

A core security technique to protect the OS kernel against untrusted user processes is system call checking. The most widely-used implementation of such a technique is the Secure Computing (Seccomp) module [126], which is implemented in Linux. Seccomp allows the software to specify which system calls a given process can make, and which argument values such system calls can use. This information is specified in a *profile* for the process, which is expressed as a Berkeley Packet Filter (BPF) program called a *filter* [150]. When the process is loaded, a Just-In-Time compiler in the kernel converts the BPF program into native code. The Seccomp filter executes in the OS kernel every time the process performs a system call. The filter may allow the system call to proceed or, if it is illegal, capture it. In this case, the OS kernel may take one of multiple actions: kill the process or thread, send a `SIGSYS` signal to the thread, return an error, or log the system call [126]. System call checking is also used by other modern OSes, such as OpenBSD with Pledge [127] and Tame [128], and Windows with System Call Disable Policy [129]. The idea behind our proposal, Draco, can be applied to all of them.

Figure 4.1 shows an example of a system call check. In user space, the program loads the system call argument and ID in registers `rdi` and `rax`, respectively, and performs the system call. This ID corresponds to the `personality` system call. As execution enters the OS kernel, Seccomp checks if this combination of system call ID and argument value is allowed. If it is allowed, it lets the system call execution to proceed (Line 8); otherwise, it terminates the process (Line 11).

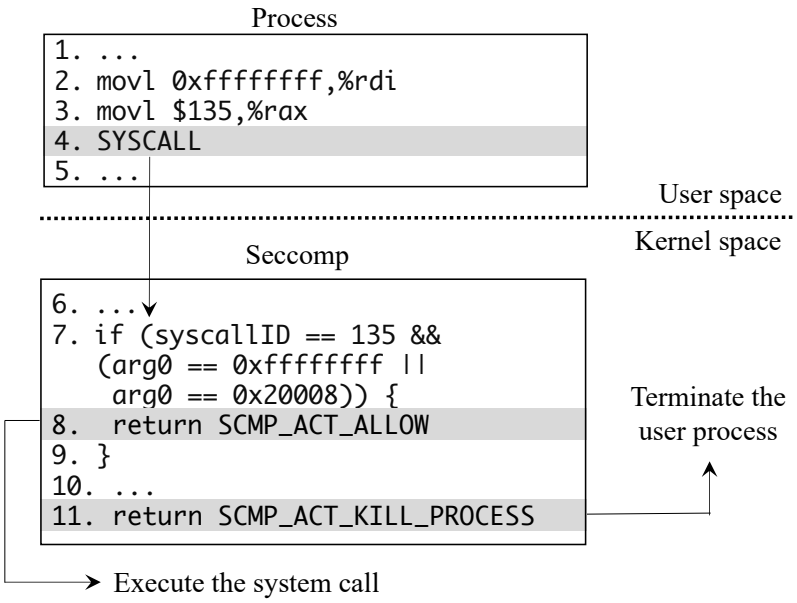


Figure 4.1: Checking a system call with Seccomp.

Figure 4.1 shows that a Seccomp profile is a long list of `if` statements that are executed in sequence. Finding the target system call and the target combination of argument values in the list can take a long time, which is the reason for the often high overhead of Seccomp.

Seccomp does not check the values of arguments that are pointers. This is because such a check does not provide any protection: a malicious user could change the contents of the location pointed to by the pointer after the check, creating a Time-Of-Check-Time-Of-Use (TOCTOU) attack [151, 152].

Seccomp could potentially do advanced checks such as checking the value range of an argument, or the result of an operation on the arguments. However, our study of real-world Seccomp profiles shows that most real-world profiles simply check system call IDs and argument values based on a whitelist of exact IDs and values [134, 153, 154, 155, 156, 157].

4.2.3 System Call Checking in Modern Systems

System call checking with Seccomp is performed in almost all of the modern Linux-based system infrastructure. This includes modern container and lightweight VM runtimes such as Docker [132], Google's gVisor [134], Amazon's Firecracker [135, 136], and CoreOS rkt [137]. Other environments, such as Android and Chrome OS also use Seccomp [130, 158, 159, 160]. The systemd Linux service manager [131] has adopted Seccomp to further increase the isolation of user containers.

Google's Sandboxed API [141] is an initiative that aims to isolate C/C++ libraries using Seccomp. This initiative, together with others [161, 162, 163, 164], significantly reduce the barrier to apply Seccomp profiles customized for applications.

Seccomp profiles. In this chapter, we focus on container technologies, which have both high performance and high security requirements. The default Seccomp profiles provided by existing container runtimes typically contain hundreds of system call IDs and fewer argument values. For example, *Docker's default profile* allows 358 system calls, and only checks 7 unique argument values (of the `clone` and `personality` system calls). This profile is widely used by other container technologies such as CoreOS Rtk and Singularity, and is applied by container management systems such as Kubernetes, Mesos, and RedHat's RedShift. In this chapter, we use Docker's default Seccomp profile as our baseline profile.

Other profiles include the default gVisor profile, which is a whitelist of 74 system calls and 130 argument checks. Also, the profile for the AWS Firecracker microVMs contains 37 system calls and 8 argument checks [135].

In this chapter, we also explore more secure Seccomp profiles tailored for some applications. *Application-specific* profiles are supported by Docker and other projects [132, 139, 164, 165, 166, 167].

4.3 THREAT MODEL

We adopt the same threat model as existing system call checking systems such as Seccomp, where untrusted user space processes can be adversarial and attack the OS kernel. In the context of containers, the containerized applications are deployed in the cloud, and an adversary tries to compromise a container and exploit vulnerabilities in the host OS kernel to achieve privilege escalation and arbitrary code execution. The system call interface is the major attack vector for user processes to attack the OS kernel.

Prior work [168, 169, 170] has shown that system call checking is effective in defending against real-world attacks, because most applications only use a small number of different system calls and

argument values [117, 118, 119, 120, 121]. For example, the mitigation of CVE-2014-3153 [118] is to disallow `FUTEX_REQUEUE` as the value of the `futex_op` argument of the `futex` system call.

Our focus is not to invent new security analyses or policies, but to minimize the execution overhead that hinders the deployment of comprehensive security checks. As will be shown in this chapter, existing software-based security checking is often costly. This forces users to sacrifice either performance or security. Draco provides both high performance and a high level of security.

4.4 MEASURING OVERHEAD & LOCALITY

To motivate our proposal, we benchmark the overhead of state-of-the-art system call checking. We focus on understanding the overhead of: 1) using generic system call profiles, 2) using smaller, application-specific system call profiles, and 3) using profiles that also check system call arguments.

4.4.1 Methodology

We run macro and micro benchmarks in Docker containers with each of the following four Seccomp profiles:

insecure: Seccomp is disabled. It is an insecure baseline where no checks are performed.

docker-default: Docker's default Seccomp profile. It is automatically used in all Docker containers and other container technologies (e.g., CoreOS rkt and Singularity) as part of the Moby project [171]. This profile is deployed by container management systems like Kubernetes, RedHat RedShift, and Mesos Containerizers.

syscall-noargs: Application-specific profiles without argument checks, where the filter whitelists the exact system call IDs used by the application.

syscall-complete: Application-specific profiles with argument checks, where the filter whitelists the exact system call IDs and the exact argument values used by the application. These profiles are the most secure filters that include both system call IDs and their arguments.

syscall-complete-2x: Application-specific profiles that consist of running the above syscall-complete profile *twice in a row*. Hence, these profiles perform twice the number of checks as syscall-complete. The goal is to model a near-future environment that performs more extensive security checks.

Section 4.10.2 describes our toolkits for automatically generating the syscall-noargs, syscall-complete, and syscall-complete-2x profiles for a given application. The workloads are described in Section 4.10.1, and are grouped into macro benchmarks and micro benchmarks. All workloads

run on an Intel Xeon (E5-2660 v3) system at 2.60GHz with 64 GB of DRAM, using Ubuntu 18.04 with the Linux 5.3.0 kernel. We disable the `spec_store_bypass`, `spectre_v2`, `mds`, `pti`, and `l1tf` vulnerability mitigations due to their heavy performance impact—many of these kernel patches will be removed in the future anyway.

We run the workloads with the Linux BPF JIT compiler enabled. Enabling the JIT compiler can achieve 2–3× performance increases [145]. Note that JIT compilers are disabled by default in many Linux distributions to prevent kernel JIT spraying attacks [172, 173, 174]. Hence, the Seccomp performance that we report in this section represents the highest performance for the baseline system.

4.4.2 Execution Overhead

Figure 4.2 shows the latency or execution time of the workloads using the five types of Seccomp profiles described in Section 4.4.1. For each workload, the results are normalized to `insecure` (i.e., Seccomp disabled).

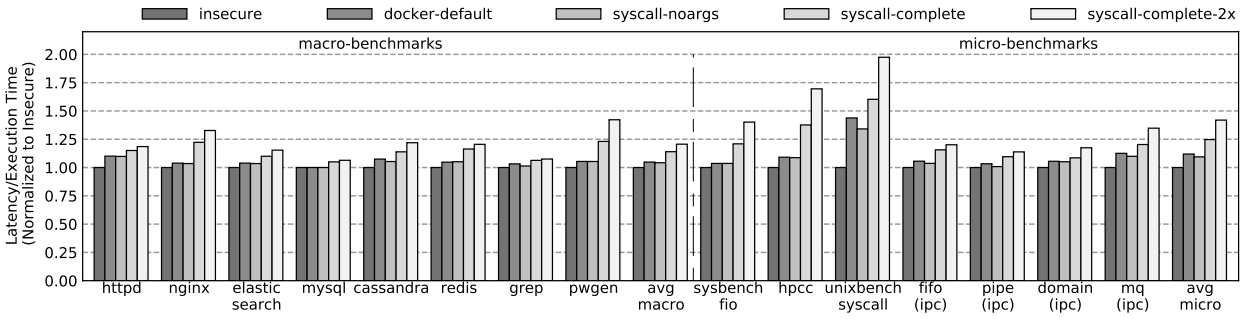


Figure 4.2: Latency or execution time of the workloads using different Seccomp profiles. For each workload, the results are normalized to `insecure` (i.e., Seccomp disabled).

Overhead of checking system call IDs. To assess the overhead of checking system call IDs, we compare insecure to docker-default. As shown in Figure 4.2, enabling Seccomp using the Docker default profile increases the execution time of the macro and micro benchmarks by 1.05× and 1.12× on average, respectively.

Comparing docker-default to syscall-noargs, we see the impact of using application-specific profiles. Sometimes, the overhead is visibly reduced. This is because the number of instructions needed to execute the syscall-noargs profile is smaller than that of docker-default. Overall, the average performance overhead of using syscall-noargs profiles is 1.04× for macro benchmarks and 1.09× for micro benchmarks, respectively.

Overhead of checking system call arguments. To assess the overhead of checking system call arguments, we first compare syscall-noargs with syscall-complete. The number of system calls in

these two profiles is the same, but syscall-complete additionally checks argument values. We can see that checking arguments brings significant overhead. On average, compared to syscall-noargs, the macro and micro benchmarks increase their execution time from 1.04× to 1.14× and from 1.09× to 1.25×, respectively.

We now double the number of checks by going from syscall-complete to syscall-complete-2x. We can see that the benchmark overhead often nearly doubles. On average, compared to syscall-complete, the macro and micro benchmarks increase their execution time from 1.14× to 1.21× and from 1.25× to 1.42×, respectively.

Implications. Our results lead to the following conclusions. First, checking system call IDs can introduce noticeable performance overhead to applications running in Docker containers. This is the case even with Seccomp, which is the most efficient implementation of the checks.

Second, checking system call arguments is significantly more expensive than checking only system call IDs. This is because the number of arguments per system call and the number of distinct values per argument are often large.

Finally, doubling the security checks in the profiles almost doubles the performance overhead.

4.4.3 System Call Locality

We measured all the system calls and arguments issued by our macro benchmarks. Figure 4.3 shows the frequency of the top calls which, together, account for 86% of all the calls. For example, `read` accounts for about 18% of all system calls. We further break down each bar into the different argument sets used by the system call. Each color in a bar corresponds to the frequency of a different argument set (or none).

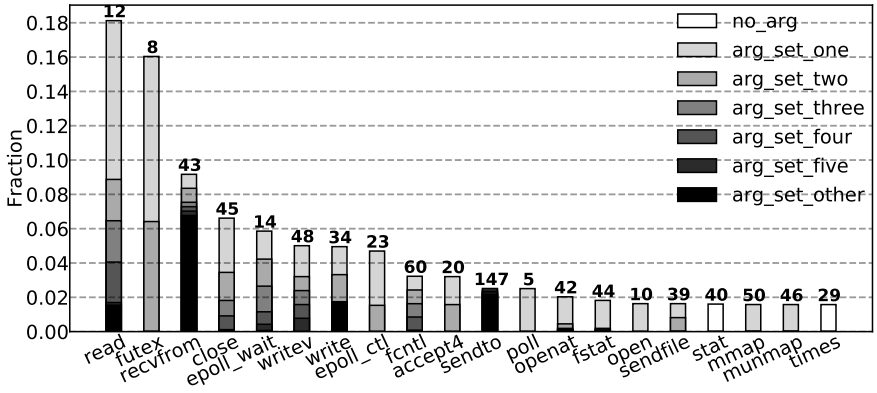


Figure 4.3: Frequency of the top system calls and average reuse distance collected from the macro benchmarks.

We see that system calls have a high locality: 20 system calls account for 86% of all the calls.

Further, individual system calls are often called with three or fewer different argument sets. At the top of each bar, we show the average *reuse distance*, defined as the number of other system calls between two system calls with the same ID and argument set. As shown in Figure 4.3, the average distance is often only a few tens of system calls, indicating high locality in reusing system call checking results.

4.5 DRACO SYSTEM CALL CHECKING

To minimize the overhead of system call checking, this chapter proposes a new architecture called *Draco*. Draco avoids executing the many `if` statements of a Seccomp profile at every system call (Figure 4.1). Draco *caches* system call IDs and argument set values after they have been checked and validated once. System calls are first looked-up in the cache and, on a hit, the system call and its arguments are declared validated, skipping the execution of the Seccomp filter.

Figure 4.4 shows the workflow. On reception of a system call, a table of validated system call IDs and argument values is checked. If the current system call and arguments match an entry in the table, the system call is allowed to proceed. Otherwise, the OS kernel executes the Seccomp filter and decides if the system call is allowed. If so, the table is updated with the new entry and the system call proceeds; otherwise the program is killed or other actions are taken (Section 4.2.2).

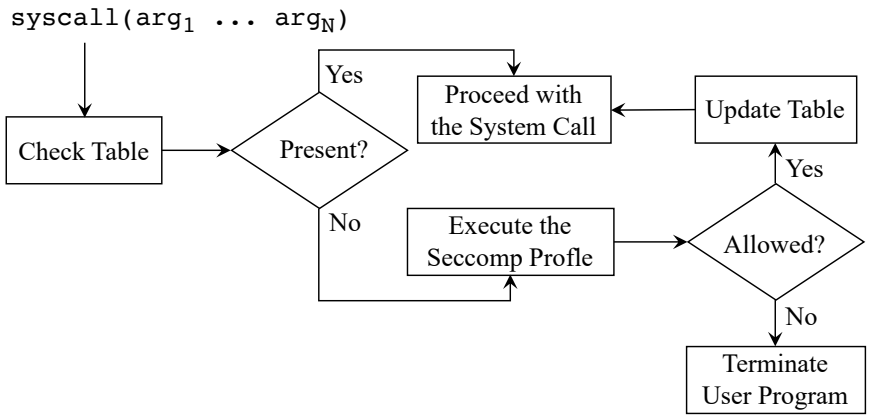


Figure 4.4: Workflow of Draco system call checking.

This approach is correct because Seccomp profiles are stateless. This means that the output of the Seccomp filter execution depends only on the input system call ID and arguments—not on some other state. Hence, a validation that succeeded in the past does not need to be repeated.

Draco can be implemented in software or in hardware. In the following, we first describe the basic design, and then how we implement it in software and in hardware. We start with a design that checks system call IDs only, and then extend it to check system call argument values as well.

4.5.1 Checking System Call IDs Only

If we only want to check system call IDs, the design is simple. It uses a table called *System Call Permissions Table* (SPT), with as many entries as different system calls. Each entry stores a single *Valid* bit. If the Valid bit is set, the system call is allowed. In Draco's hardware implementation, each core has a hardware SPT. In both hardware and software implementations, the SPT contains information for one process.

When the System Call ID (SID) of the system call is known, Draco finds the SPT entry for that SID, and checks if the Valid bit is set. If it is, the system call proceeds. Otherwise, the Seccomp filter is executed.

4.5.2 Checking System Call Arguments

To check system call arguments, we enhance the SPT and couple it with a *software structure* called *Validated Argument Table* (VAT). Both SPT and VAT are private to the process. The VAT is the same for both software and hardware implementations of Draco.

Figure 4.5 shows the structures. The SPT is still indexed with the SID. An entry now includes, in addition to the Valid bit, a *Base* and an *Argument Bitmask* field. The Base field stores the virtual address of the section of the VAT that holds information about this system call. The Argument Bitmask stores information to determine what arguments are used by this system call. Recall that a system call can take up to 6 arguments, each 1 to 8 bytes long. Hence, the Argument Bitmask has one bit per argument byte, for a total of 48 bits. A given bit is set if this system call uses this byte as an argument—e.g., for a system call that uses two arguments of one byte each, the Argument Bitmask has bits 0 and 8 set.

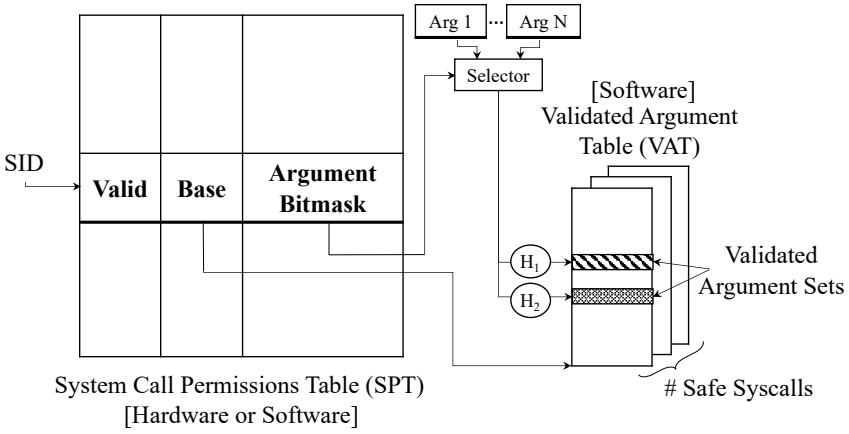


Figure 4.5: Checking a system call and its arguments.

The VAT of a process has one structure for each system call allowed for this process. Each structure is a hash table of limited size, indexed with two hash functions. If an entry in the hash table is filled, it holds the values of an argument set that has been found to be safe for this particular system call.

When a system call is encountered, to find the correct entry in the VAT, Draco hashes the argument values. Specifically, when the system call's SID and argument set are known, the SPT is indexed with the SID. Draco uses the Arguments Bitmask to select which parts of the arguments to pass to the hash functions and generate hash table indices for the VAT. For example, if a system call uses two arguments of one byte each, only the eight bits of each argument are used to generate the hash table indices.

The address in Base is added to the hash table indices to access the VAT. Draco fetches the contents of the two entries, and compares them to the values of the arguments of the system call. If any of the two comparisons produces a match, the system call is allowed.

Draco uses two hash functions (H_1 and H_2 in Figure 4.5) for the following reason. To avoid having to deal with hash table collisions in a VAT structure, which would result in multiple VAT probes, each VAT structure uses 2-ary cuckoo hashing [175, 176]. Such a design resolves collisions gracefully. However, it needs to use two hash functions to perform two accesses to the target VAT structure in parallel. On a read, the resulting two entries are checked for a match. On an insertion, the cuckoo hashing algorithm is used to find a spot.

4.5.3 A Software Implementation

Draco can be completely implemented in software. Both the SPT and the VAT are software structures in memory. We build Draco as a Linux kernel component. In the OS kernel, at the entry point of system calls, we insert instructions to read the SID and argument set values (if argument checking is configured). Draco uses the SID to index into the SPT and decides if the system call is allowed or not based on the Valid bit. If argument checking is configured, Draco further takes the correct bits from the arguments to perform the hashes, then reads the VAT, compares the argument values, and decides whether the system call passes or not.

4.5.4 An Initial Hardware Implementation

An initial hardware implementation of Draco makes the SPT a hardware table in each core. The VAT remains a software structure in memory. The checks can only be performed when either the SID or both the SID and argument set values are available. To simplify the hardware, Draco waits until the system call instruction reaches the Reorder Buffer (ROB) head. At that point, all

the information about the arguments is guaranteed to be available in specific registers. Hence, the Draco hardware indexes the SPT and checks the Valid bit. If argument checking is enabled, the hardware further takes the correct bits from the arguments, performs the hashes, reads the VAT, compares the argument values, and determines if the system call passes. If the combination of system call and argument set are found not to have been validated, the OS is invoked to run the Seccomp filter.

4.6 DRACO HARDWARE IMPLEMENTATION

The initial hardware implementation of Section 4.5.4 has the shortcoming that it requires memory accesses to read the VAT—in fact, two accesses, one for each of the hash functions. While some of these accesses may hit in caches, the performance is likely to suffer. For this reason, this section extends the hardware implementation of Draco to perform the system call checks *in the pipeline* with very low overhead.

4.6.1 Caching the Results of Successful Checks

To substantially reduce the overhead of system call checking, Draco introduces a new hardware cache of recently-encountered and validated system call argument sets. The cache is called *System Call Lookaside Buffer* (SLB). It is inspired by the TLB (Translation Lookaside Buffer).

Figure 4.6 shows the SLB. It is indexed with the system call's SID and number of arguments that it requires. Since different system calls have different numbers of arguments, the SLB has a set-associative sub-structure for each group of system calls that take the same number of arguments. This design minimizes the space needed to cache arguments—each sub-structure can be sized individually. Each entry in the SLB has an SID field, a *Valid* bit, a *Hash* field, and a validated argument set denoted as $<Arg_1$ ... $Arg_N>$. The Hash field contains the hash value generated using this argument set via either the H_1 or H_2 hash functions mentioned in Section 4.5.2.

Figure 4.7 describes the SLB operations performed by Draco alongside the SPT and VAT. When a system call is detected at the head of the ROB, its SID is used to access the SPT (①). The SPT uses the Argument Bitmask (Figure 4.5) to generate the argument count used by the system call. This information, together with the SID, is used to index the SLB (②).

On an SLB miss, the corresponding VAT is accessed. Draco takes the current system call argument set and the Argument Bitmask from the SPT and, using hash functions H_1 and H_2, generates two hash values (③). Such hash values are combined with the Base address provided by the SPT to access the VAT in memory. The resulting two VAT locations are fetched in parallel, and their

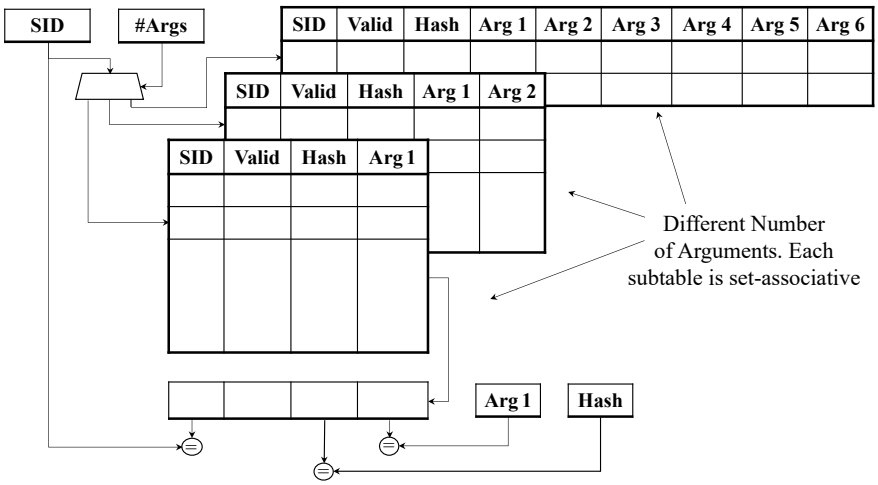


Figure 4.6: System Call Lookaside Buffer (SLB) structure.

contents are compared to the actual system call argument set. On a match, the SLB entry is filled with the SID, the validated argument set, and the one hash value that fetched the correct entry from the VAT (④). The system call is then allowed to resume execution.

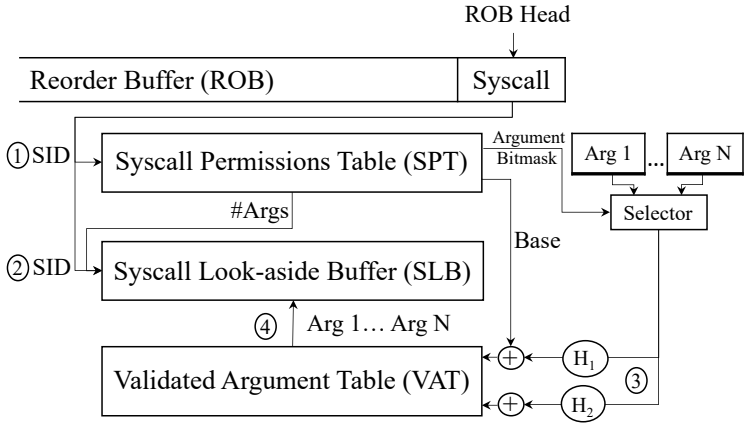


Figure 4.7: Flow of operations in an SLB access.

On subsequent accesses to the SLB with the same system call's SID and argument set, the SLB will hit. A hit occurs when an entry is found in the SLB that has the same SID and argument set values as the incoming system call. In this case, the system call is allowed to proceed *without requiring any memory hierarchy access*. In this case, which we assume is the most frequent one, the checking of the system call and arguments has negligible overhead.

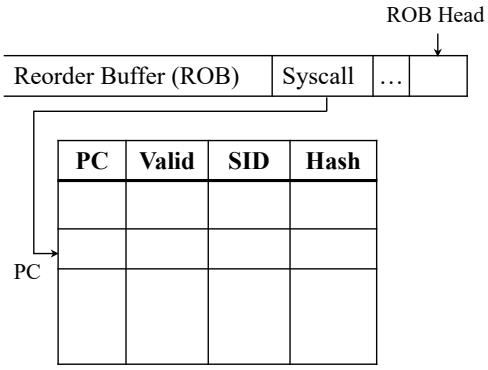


Figure 4.8: System Call Target Buffer (STB) structure.

4.6.2 Preloading Argument Sets

The SLB can significantly reduce the system call checking time by caching recently validated arguments. However, on a miss, the system call stalls at the ROB head until its arguments are successfully checked against data coming from the VAT in memory. To avoid this problem, we want to hide all the stall time by *preloading the SLB entry early*.

This function is performed by the *System Call Target Buffer* (STB). The STB is inspired by the Branch Target Buffer. Its goal is to preload a potentially useful entry in the SLB as soon as a system call is placed in the ROB. Specifically, it preloads in the SLB an argument set that is in the VAT and, therefore, has been validated in the past for the same system call. When the system call reaches the head of the ROB and tries to check its arguments, it is likely that the correct argument set is already preloaded into the SLB.

Fundamentally, the STB operates like the BTB. While the BTB predicts the target location that the upcoming branch will jump to, the STB predicts the location in the VAT that stores the validated argument set that the upcoming system call will require. Knowing this location allows the hardware to preload the information into the SLB in advance.

System Call Target Buffer and Operation. The STB is shown in Figure 4.8. Each entry stores the program counter (PC) of a system call, a *Valid* bit, the SID of the system call, and a *Hash* field. The latter contains the one hash value (of the two possible) that fetched this argument set from the VAT when the entry was loaded into the STB.

The preloading operation into the SLB is shown in Figure 4.9. As soon as an instruction is inserted in the ROB, Draco uses its PC to access the STB (①). A hit in the STB is declared when the PC in the ROB matches one in the STB. This means that the instruction is a system call. At that point, the STB returns the SID and the predicted hash value. Note that the SID is the correct one because there is only one single type of system call in a given PC. At this point, the hardware knows the system call. However, it does not know the actual argument values because, unlike in

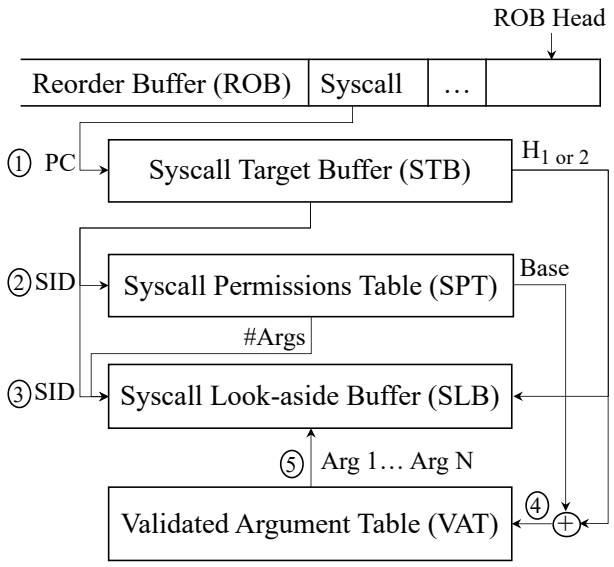


Figure 4.9: Flow of operations in an SLB preloading.

Section 4.6.1, the system call is not at the ROB head, and the actual arguments may not be ready yet.

Next, Draco accesses the SPT using the SID (②), which provides the Base address of the structure in the VAT, and the number of arguments of the system call.

At this point, the hardware has to: (i) check if the SLB already has an entry that will likely cause an SLB hit when the system call reaches the ROB head and, (ii) if not, find such an entry in the VAT and preload it in the SLB. To perform (i), Draco uses the SID and the number of arguments to access the set-associative subtable of the SLB (③) that has the correct argument count (Figure 4.6). Since we do not yet know the actual argument set, we consider a hit in the SLB when the hash value provided by the STB matches the one stored in the SLB entry. This is shown in Figure 4.6. We call this an *SLB Preload hit*. No further action is needed because the SLB likely has the correct entry. Later, when the system call reaches the head of the ROB and the system call argument set is known, the SLB is accessed again with the SID and the argument set. If the argument set matches the one in the SLB entry, it is an *SLB Access hit*; the system call has been checked without any memory system access at all.

If, instead, no SLB Preload hit occurs, Draco performs the action (ii) above. Specifically, Draco reads from the SPT the Base address of the structure in the VAT, reads from the STB the hash value, combines them, and indexes the VAT (④). If a valid entry is found in the VAT, its argument set is preloaded into the SLB (⑤). Again, when the system call reaches the head of the ROB, this SLB entry will be checked for a match.

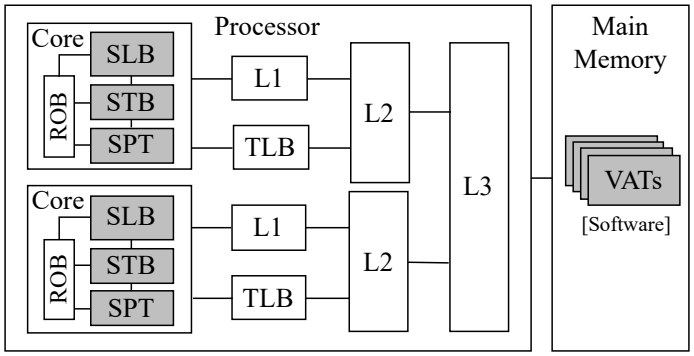


Figure 4.10: Multicore with Draco structures in a shaded pattern.

4.6.3 Putting it All Together

Figure 4.10 shows the Draco hardware implementation in a multi-core chip. It has three per-core hardware tables: SLB, STB, and SPT. In our conservatively-sized design, they use 8KB, 4KB, and 4KB, respectively. The SLB holds a process' most popular *checked* system calls and their *checked* argument sets. When a system call reaches the head of the ROB, the SLB is queried. The SLB information is backed up in the per-process software VAT in memory. VAT entries are loaded into the SLB on demand by hardware.

As indicated, Draco does not wait for the system call to reach the ROB head before checking the SLB. Instead, as soon as a system call enters the ROB, Draco checks the SLB to see if it can possibly have the correct entry. If not, it accesses the VAT and loads a good-guess entry into the SLB.

For a core to access its SLB, Draco includes two local hardware structures: the STB—which takes an instruction PC and provides an SID—and the SPT—which takes the SID and provides its argument count and its VAT location.

Draco execution flows. Table 4.1 shows the possible Draco execution flows. Flow ① occurs when the STB access, SLB preload, and SLB access all hit. It results in a fast execution. Flow ② occurs when the STB access and SLB preload hit, but the actual SLB access does not find the correct argument set. In this case, Draco fetches the argument set from the VAT, and fills an entry in the STB with the correct hash, and in the SLB with the correct hash and argument set. This flow is marked as slow in Table 4.1. Note however, that the argument set may be found in the caches, which saves accesses to main memory. Finally, if the correct argument set is not found in the VAT, the OS is invoked and executes the Seccomp filter. If such execution validates the system call, the VAT is updated as well with the validated SID and argument set.

Flow ③ occurs when the STB access hits, the SLB preload misses, and Draco initiates an SLB preload that eventually delivers an SLB access hit. As soon as the preload SLB miss is declared,

Execution Flow	STB Access	SLB Preload	SLB Access	Action	Speed
①	Hit	Hit	Hit	None.	Fast
②	Hit	Hit	Miss	After the SLB access miss fetch the argument set from the VAT, and fill an entry in the STB with the correct hash, and in the SLB with the correct hash and arguments.	Slow
③	Hit	Miss	Hit	After the SLB preload miss, fetch the argument set from the VAT, and fill an entry	Fast
④	Hit	Miss	Miss	After the SLB preload miss, do as ③. After the SLB access miss, do as ②.	Slow
⑤	Miss	N/A	Hit	After the SLB access hit, fill an entry in the STB with the correct SID and hash.	Fast
⑥	Miss	N/A	Miss	After the SLB access miss, fetch the argument set from the VAT, and fill an entry in the STB with correct SID and hash, and in the SLB with correct SID, hash, and arguments.	Slow

Table 4.1: Possible Draco execution flows. The *Slow* cases can have different latency, depending on whether the VAT accesses hit or miss in the caches. If the VAT does not have the requested entry, the OS is invoked and executes the Seccomp filter.

Draco fetches the argument set from the VAT, and fills an entry in the SLB with the correct SID, hash, and arguments. When the system call reaches the ROB head and checks the SLB, it declares an SLB access hit. This is a fast case.

Flow ④ occurs when the STB access hits, the SLB preload misses, Draco's SLB preload brings incorrect data, and the actual SLB access misses. In this case, after the SLB preload miss, Draco performs the actions in Flow ③; after the SLB access miss, Draco performs the actions in Flow ②.

Flows ⑤ and ⑥ start with an STB miss. Draco does not preload the SLB because it does not know the SID. When the system call reaches the head of the ROB, the SLB is accessed. In Flow

⑤, the access hits. In this case, after the SLB access hit, Draco fills an entry in the STB with the correct SID and hash. In Flow ⑥, the SLB access misses, and Draco has to fill an entry in both STB and SLB. ⑤ is fast and ⑥ is slow.

4.7 SYSTEM SUPPORT

4.7.1 VAT Design and Implementation

The OS kernel is responsible for filling the VAT of each process. The VAT of a process has a two-way cuckoo hash table for each system call allowed to the process. The OS sizes each table based on the number of argument sets used by corresponding system call (e.g., based on the given Seccomp profile). To minimize insertion failures in the hash tables, the size of each table is over-provisioned two times the number of estimated argument sets. On an insertion to a table, if the cuckoo hashing fails after a threshold number of attempts, the OS makes room by evicting one entry.

During a table lookup, either the OS or the hardware (in the hardware implementation) accesses the two ways of the cuckoo hash table. For the hash functions, we use the *ECMA* [177] and the $\neg$ *ECMA* polynomials to compute the Cyclic Redundancy Check (CRC) code of the system call argument set (Figure 4.5).

The base addresses in the SPT are stored as virtual addresses. In the hardware implementation of Draco, the hardware translates this base address before accessing a VAT structure. Due to the small size of the VAT (several KBs for a process), this design enjoys good TLB translation locality, as well as natural caching of translations in the cache hierarchy. If a page fault occurs on a VAT access, the OS handles it as a regular page fault.

4.7.2 Implementation Aspects

Invocation of Software Checking. When the Draco hardware does not find the correct SID or argument set in the VAT, it sets a register called *SWCheckNeeded*. As the system call instruction completes, the system call handler in the OS first checks the SWCheckNeeded register. If it is set, the OS runs system call checking (e.g., Seccomp). If the check succeeds, the OS inserts the appropriate entry in the VAT and continues; otherwise the OS does not allow the system call to execute.

Data Coherence. System call filters are not modified during process runtime to limit attackers' capabilities. Hence, there is no need to add support to keep the three hardware structures (SLB,

STB, and SPT) coherent across cores. Draco only provides a fast way to clear all these structures in one shot.

Context Switches. A simple design would simply invalidate the three hardware structures on a context switch. To reduce the start-up overhead after a context switch, Draco improves on this design with two simple supports. First, on a context switch, the OS saves to memory a few key SPT entries for the process being preempted, and restores from memory the saved SPT entries for the incoming process. To pick which SPT entries to save, Draco enhances the SPT with an *Accessed* bit per entry. When a system call hits on one of the SPT entries, the entry's Accessed bit is set. Periodically (e.g., every 500μs), the hardware clears the Accessed bits. On a context switch, only the SPT entries with the Accessed bit set are saved.

The second support is that, if the process that will be scheduled after the context switch is the same as the one that is being preempted, the structures are not invalidated. This is safe with respect to side-channels. If a different process is to be scheduled, the hardware structures are invalidated.

Simultaneous Multithreading (SMT) Support. Draco can support SMT by partitioning the three hardware structures and giving one partition to each SMT context. Each context accesses its partition.

4.8 GENERALITY OF DRACO

The discussions so far presented a Draco design that is broadly compatible with Linux's Seccomp, so we could describe a whole-system solution. In practice, it is easy to apply the Draco ideas to other system call checking mechanisms such as OpenBSD's Pledge and Tame [127, 128], and Window's System Call Disable Policy [178]. Draco is generic to modern OS-level sandboxing and containerization technologies.

Specifically, recall that the OS populates the SPT with system call information. Each SPT entry corresponds to a system call ID and has argument information. Hence, different OS kernels will have different SPT contents due to different system calls and different arguments.

In our design, the Draco hardware knows which registers contain which arguments of system calls. However, we can make the design more general and usable by other OS kernels. Specifically, we can add an OS-programmable table that contains the mapping between system call argument number and general-purpose register that holds it. This way, we can use arbitrary registers.

The hardware structures proposed by Draco can further support other security checks that relate to the security of transitions between different privilege domains. For example Draco can support security checks in virtualized environments, such as when the guest OS invokes the hypervisor through hypercalls. Similarly, Draco can be applied to user-level container technologies such as

Google's gVisor [134], where a user-level guardian process such as the Sentry or Gofer is invoked to handle requests of less privileged application processes. Draco can also augment the security of library calls, such as in the recently-proposed Google Sandboxed API project [141].

In general, the Draco hardware structures are most attractive in processors used in domains that require both high performance and high security. A particularly relevant domain is interactive web services, such as on-line retail. Studies by Akamai, Google, and Amazon have shown that even short delays can impact online revenue [179, 180]. Furthermore, in this domain, security in paramount, and the expectation is that security checks will become more extensive in the near future.

4.9 SECURITY ISSUES

To understand the security issues in Draco, we consider two types of side-channels: those in the Draco hardware structures and those in the cache hierarchy.

Draco hardware structures could potentially provide side channels due to two reasons: (i) Draco uses speculation as it preloads the SLB before the system call instruction reaches the head of the ROB, and (ii) the Draco hardware structures are shared by multiple processes. Consider the first reason. An adversary could trigger SLB preloading followed by a squash, which could then speed-up a subsequent benign access that uses the same SLB entry and reveal a secret. To shield Draco from this speculation-based attack, we design the preloading mechanism carefully. The idea is to ensure that preloading leaves no side effect in the Draco hardware structures until the system call instruction reaches the ROB head.

Specifically, if an SLB preload request hits in the SLB, the LRU state of the SLB is not updated until the corresponding non-speculative SLB access. Moreover, if an SLB preload request misses, the requested VAT entry is not immediately loaded into the SLB; instead, it is stored in a Temporary Buffer. When the non-speculative SLB access is performed, the entry is moved into the SLB. If, instead, the system call instruction is squashed, the temporary buffer is cleared. Fortunately, this temporary buffer only needs to store a few entries (i.e., 8 in our design), since the number of system call instructions in the ROB at a time is small.

The second potential reason for side channels is the sharing of the Draco hardware structures by multiple processes. This case is eliminated as follows. First, in the presence of SMT, the SLB, STB, and SPT structures are partitioned on a per-context basis. Second, in all cases, when a core (or hardware context) performs a context switch to a different process, the SLB, STB, and SPT are invalidated.

Regarding side channels in the cache hierarchy, they can be eliminated using existing proposals

against them. Specifically, for cache accesses due to speculative SLB preload, we can use any of the recent proposals that protect the cache hierarchy against speculation attacks (e.g., [67, 68, 181, 182, 183, 184]). Further, for state left in the cache hierarchy as Draco hardware structures are used, we can use existing proposals like cache partitioning [109, 185]. Note that this type of potential side channel *also occurs in Seccomp*. Indeed, on a context switch, Draco may leave VAT state in the caches, but Seccomp may also leave state in the caches that reveals what system calls where checked. For these reasons, we consider side channels in the cache hierarchy beyond the scope of this chapter.

4.10 EVALUATION METHODOLOGY

We evaluate both the software and the hardware implementations of Draco. We evaluate the software implementation on the Intel Xeon E5-2660 v3 multiprocessor system described in Section 4.4.1; we evaluate the hardware implementation of Draco with cycle-level simulations.

4.10.1 Workloads and Metrics

We use fifteen workloads split into macro and micro benchmarks. The macro benchmarks are long-running applications, including the Elasticsearch [186], HTTPD, and NGINX web servers, Redis (an in-memory cache), Cassandra (a NoSQL database), and MySQL. We use the Yahoo! Cloud Serving Benchmark (YCSB) [187] using `workloada` and `workloadc` with 10 and 30 clients to drive Elasticsearch and Cassandra, respectively. For HTTPD and NGINX, we use `ab`, the Apache HTTP server benchmarking tool [188] with 30 concurrent requests. We drive MySQL with the OLTP workload of SysBench [189] with 10 clients. For Redis, we use the redis-benchmark [190] with 30 concurrent requests. We also evaluate Function-as-as-Service scenarios, using functions similar to the sample functions of OpenFaaS [191]. Specifically, we use a `pwgen` function that generates 10K secure passwords and a `grep` function that searches patterns in the Linux source tree.

For micro benchmarks, we use FIO from SysBench [189] with 128 files of a total size of 512MB, GUPS from the HPC Challenge Benchmark (HPCC) [192], Syscall from UnixBench [193] in mix mode, and fifo, pipe, domain, and message queue from IPC Bench [194] with 1000B packets.

For macro benchmarks, we measure the mean request latency in HTTPD, NGINX, Elasticsearch, Redis, Cassandra, and MySQL, and the total execution time in the functions. For micro benchmarks, we measure the message latency in the benchmarks from IPC Bench, and the execution time in the remaining benchmarks.

4.10.2 System Call Profile Generation

There are a number of ways to create application-specific system call profiles using both dynamic and static analysis. They include system call profiling (where system calls not observed during profiling are not allowed in production) [163, 170, 195, 196, 197] and binary analysis [161, 162, 168].

We build our own software toolkit for automatically generating the syscall-noargs, syscall-complete, and syscall-complete-2x profiles described in Section 4.4.1 for target applications. The toolkit has components for (1) attaching `strace` onto a running application to collect the system call traces, and (2) generating the Seccomp profiles that only allow the system call IDs and argument sets that appeared in the recorded traces. We choose to build our own toolkit because we find that no existing system call profiling tool includes arguments in the profiles.

For syscall-complete-2x, we run the syscall-complete profile twice in a row. Hence, the resulting profile performs twice the number of checks as syscall-complete. The goal of syscall-complete-2x is to model a near-future environment where we need more extensive security checks.

4.10.3 Modeling the Hardware Implementation of Draco

We use cycle-level full-system simulations to model a server architecture with 10 cores and 32 GB of main memory. The configuration is shown in Table 4.2. Each core is out-of-order (OOO) and has private L1 instruction and data caches, and a private unified L2 cache. The banked L3 cache is shared.

We integrate the Simics [198] full-system simulator with the SST framework [199], together with the DRAMSim2 [200] memory simulator. Moreover, we utilize Intel SAE [201] on Simics for OS instrumentation. We use CACTI [100] for energy and access time evaluation of memory structures and the Synopsys Design Compiler [202] for evaluating the RTL implementation of the hash functions. The system call interface follows the semantics of x86 [98]. The Simics infrastructure provides the actual memory and control register contents for each system call. To evaluate the hardware components of Draco, we model them in detail in SST. For the software components, we modify the Linux kernel and instrument the system call handler and Seccomp.

For HTTPD, NGINX, Elasticsearch, Redis, Cassandra, and MySQL, we instrument the applications to track the beginning of the steady state. We then warm-up the architectural state by running 250 million instructions, and finally measure for two billion instructions. For functions and micro benchmarks, we warm-up the architectural state for 250 million instructions and measure for two billion instructions.

The software stack for the hardware simulations uses CentOS 7.6.1810 with Linux 3.10 and

Processor Parameters	
Multicore chip	10 OOO cores, 128-entry Reorder Buffer, 2GHz
L1 (D, I) cache	32KB, 8 way, write back, 2 cyc. access time (AT)
L2 cache	256KB, 8 way, write back, 8 cycle AT
L3 cache	8MB, 16 way, write back, shared, 32 cycle AT
Per-Core Draco Parameters	
STB	256 entries, 2 way, 2 cycle AT
SLB (1 arg)	32 entries, 4 way, 2 cycle AT
SLB (2 arg)	64 entries, 4 way, 2 cycle AT
SLB (3 arg)	64 entries, 4 way, 2 cycle AT
SLB (4 arg)	32 entries, 4 way, 2 cycle AT
SLB (5 arg)	32 entries, 4 way, 2 cycle AT
SLB (6 arg)	16 entries, 4 way, 2 cycle AT
Temporary Buffer	8 entries, 4 way, 2 cycle AT
SPT	384 entries, 1 way, 2 cycle AT
Main-Memory Parameters	
Capacity; Channels	32GB; 2
Ranks/Channel	8
Banks/Rank	8
Freq; Data rate	1GHz; DDR
Host and Docker Parameters	
Host OS	CentOS 7.6.1810 with Linux 3.10
Docker Engine	18.09.3

Table 4.2: Architectural configuration used for evaluation.

Docker Engine 18.09.03. This Linux version is older than the one used for the real-system measurements of Section 4.4 and the evaluation of the software implementation of Draco in Section 4.11.1, which uses Ubuntu 18.04 with Linux 5.3.0. Our hardware simulation infrastructure could not boot the newer Linux kernel. However, note that the Draco hardware evaluation is mostly independent of the kernel version. The only exception is during the cold-start phase of the application, when the VAT structures are populated. However, our hardware simulations mostly model steady state and, therefore, the actual kernel version has negligible impact.

4.11 EVALUATION

4.11.1 Performance of Draco

Draco Software Implementation. Figure 4.11 shows the performance of the workloads using the software implementation of Draco. The figure takes three Seccomp profiles from Section 4.4.1 (`syscall-noargs`, `syscall-complete`, and `syscall-complete-2x`) and compares the performance of the workloads on a conventional system (labeled *Seccomp* in the figure) to a system aug-

mented with software Draco (labeled *DracoSW* in the figure). The figure is organized as Figure 4.2, and all the bars are normalized to the `insecure` baseline that performs no security checking.

We can see that software Draco reduces the overhead of security checking relative to Seccomp, especially for complex argument checking. Specifically, when checking both system call IDs and argument sets with `syscall-complete`, the average execution times of the macro and micro benchmarks with Seccomp are 1.14× and 1.25× higher, respectively, than `insecure`. With software Draco, these execution times are only 1.10× and 1.18× higher, respectively, than `insecure`. When checking more complex security profiles with `syscall-complete-2x`, the reductions are higher. The corresponding numbers with Seccomp are 1.21× and 1.42×; with software Draco, these numbers are 1.10× and 1.23× only.

However, we also see that the checking overhead of software Draco is still substantial, especially for some workloads. This is because the software implementation of argument checking requires expensive operations, such as hashing, VAT access, and argument comparisons. Note that our software implementation is extensively optimized. In particular, we experimented with highly-optimized hash functions used by the kernel (which use the x86 hashing instructions) and selected the most effective one.

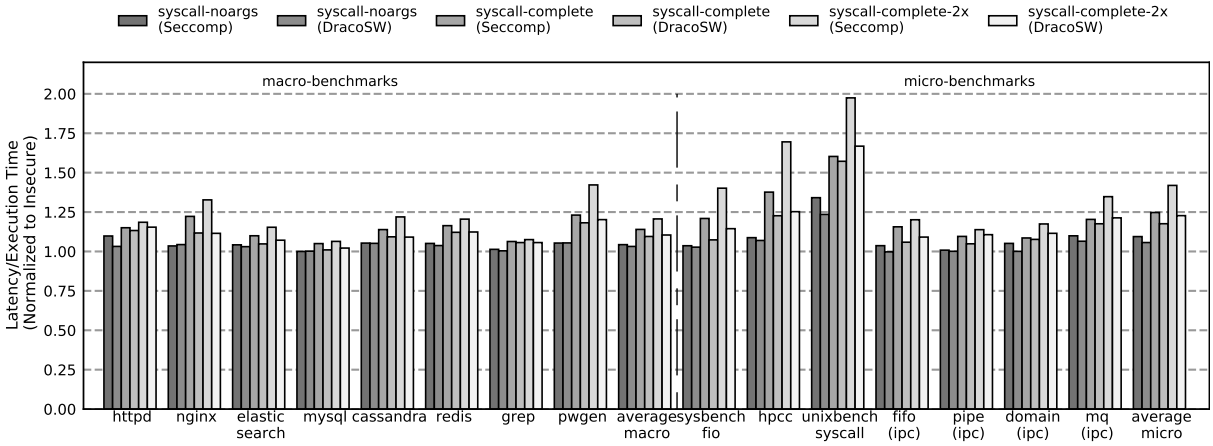


Figure 4.11: Latency or execution time of the workloads using the software implementation of Draco and other environments. For each workload, the results are normalized to `insecure`.

Draco Hardware Implementation. Figure 4.12 shows the workload performance using the hardware implementation of Draco. The figure shows `insecure` and hardware Draco running three Seccomp profiles from Section 4.4.1, namely `syscall-noargs`, `syscall-complete`, and `syscall-complete-2x`. The figure is organized as Figure 4.2, and all the bars are normalized to `insecure`.

Figure 4.12 shows that hardware Draco eliminates practically all of the checking overhead under all the profiles, both when only checking system call IDs and when checking system call IDs and

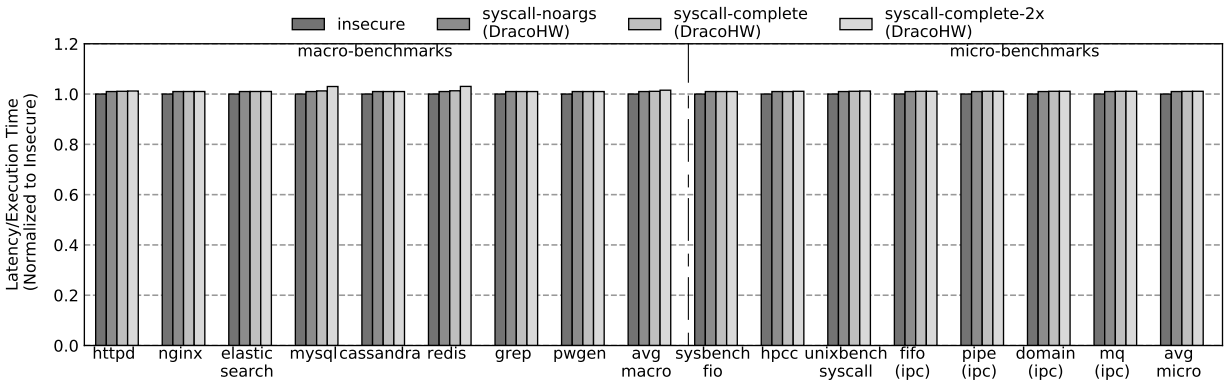


Figure 4.12: Latency or execution time of the workloads using the hardware implementation of Draco. For each workload, the results are normalized to `insecure`.

argument set values (including the double-size checks). In all cases, the average overhead of hardware Draco over `insecure` is 1%. Hence, hardware Draco is a secure, overhead-free design.

4.11.2 Hardware Structure Hit Rates

To understand hardware Draco's performance, Figure 4.13 shows the hit rates of the STB and SLB structures. For the SLB, we show two bars: Access and Preload. *SLB Access* occurs when the system call is at the head of the ROB. *SLB Preload* occurs when the system call is inserted in the ROB and the STB is looked-up. An SLB Preload hit means only that the SLB likely contains the desired entry. An SLB Preload miss triggers a memory request to preload the entry.

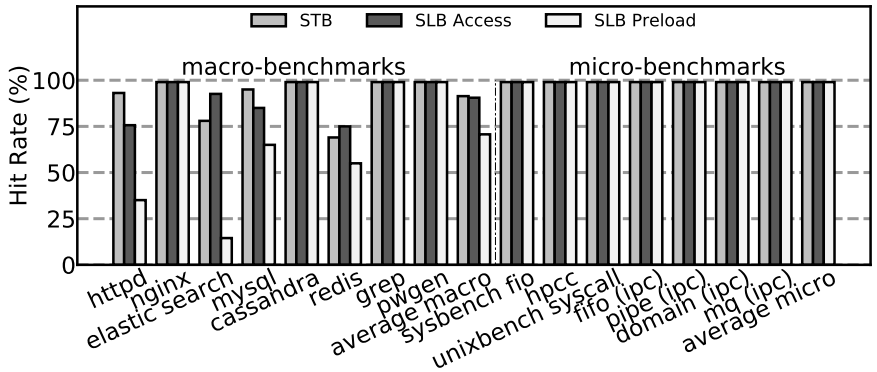


Figure 4.13: Hit rates of STB and SLB (access and preload).

The figure shows that the STB hit rate is very high. It is over 93% for all the applications except for Elasticsearch and Redis. The STB captures the working set of the system calls being used. This is good news, as an STB hit is a requirement for an SLB preload.

Consider the *SLB Preload* bars. For all applications except HTTPD, Elasticsearch, MySQL, and Redis, the hit rates are close to 99%. This means that, in most cases, the working set of the system call IDs and argument set values being used fits in the SLB.

However, even for these four applications, the *SLB Access* hit rates are 75–93%. This means that SLB preloading is successful in bringing most of the needed entries into the SLB on time, to deliver a hit when the entries are needed. Hence, we recommend the use of SLB preloding.

4.11.3 Draco Resource Overhead

Hardware Components. Hardware Draco introduces three hardware structures (SPT, STB, and SLB), and requires a CRC hash generator. In Table 4.3, we present the CACTI analysis for the three hardware structures, and the Synopsys Design Compiler results for the hash generator implemented in VHDL using a linear-feedback shift register (LFSR) design. For each unit, the table shows the area, access time, dynamic energy of a read access, and leakage power. In the SLB, the area and leakage analysis includes all the subtables for the different argument counts and the temporary buffer. For the access time and dynamic energy, we show the numbers for the largest structure, namely, the three-argument subtable.

Parameter	SPT	STB	SLB	CRC Hash
Area (mm2)	0.0036	0.0063	0.01549	0.0019
Access time (ps)	105.41	131.61	112.75	964
Dyn. rd energy (pJ)	1.32	1.78	2.69	0.98
Leak. power (mW)	1.39	2.63	3.96	0.106

Table 4.3: Draco hardware analysis at 22 nm.

Since all the structures are accessed in less that 150 ps, we conservatively use a 2-cycle access time for these structures. Further, since 964 ps are required to compute the CRC hash function, we account for 3 cycles in our evaluation.

SLB Sizing. Figure 4.14 uses a violin plot to illustrate the probability density of the number of arguments of system calls. The first entry (`linux`) corresponds to the complete Linux kernel system call interface, while the remaining entries correspond to the different applications. Taking HTTPD as an example, we see that, of all the system calls that were checked by Draco, most have three arguments, some have two and only a few have one or zero. Recall that, like Seccomp, Draco does not check pointers. For generality, we size the SLB structures based on the Linux distribution of arguments per system call.

VAT Memory Consumption. In a VAT, each system call has a hash table that is sized based on the number of argument sets used by that system call (e.g., based on the given Seccomp profile). Since

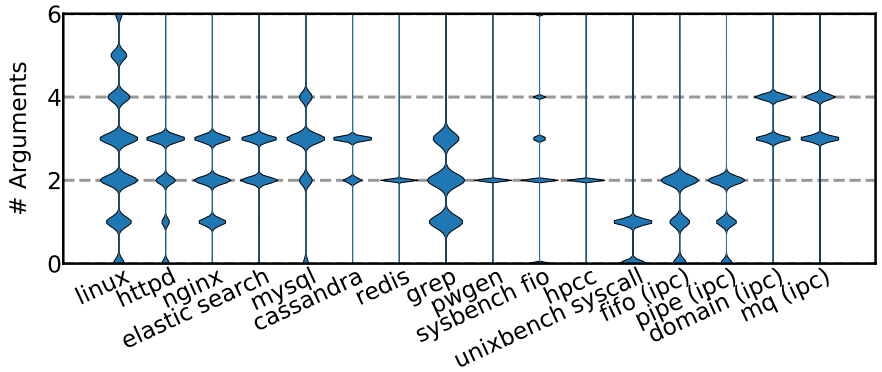


Figure 4.14: Number of arguments of system calls.

the number of system calls is bounded, the total size of the VAT is bounded. In our evaluation, we find that the geometric mean of the VAT size for a process is 6.98KB across all evaluated applications.

4.11.4 Assessing the Security of Application-Specific Profiles

We compare the security benefits of an application-specific profile (`syscall-complete` from Section 4.4.1) to the generic, commonly deployed `docker-default` profile. Recall that `syscall-complete` checks both syscall IDs and arguments, while `docker-default` only checks syscall IDs.

Figure 4.15 shows the number of different system calls allowed by the different profiles. First, `linux` shows the total number of system calls in Linux, which is 403. The next bar is `docker-default`, which allows 358 system calls. For the remaining bars, the total height is the number allowed by `syscall-complete` for each application. We can see that `syscall-complete` only allows 50–100 system calls, which increases security substantially. Moreover, the figure shows that not all of these system calls are application-specific. There is a fraction of about 20% (remaining in dark color) that are required by the container runtime. Note that, as shown in Section 4.4, while application-specific profiles are smaller, their checking overhead is still substantial.

Figure 4.16 shows the total number of system call arguments checked, and the number of unique argument values allowed. Linux does not check any arguments, while `docker-default` checks a total of three arguments and allows seven unique values (Section 4.2.3). The `syscall-complete` profile checks 23–142 arguments per application, and allows 127–2458 distinct argument values per application. All these checks substantially reduce the attack surface.

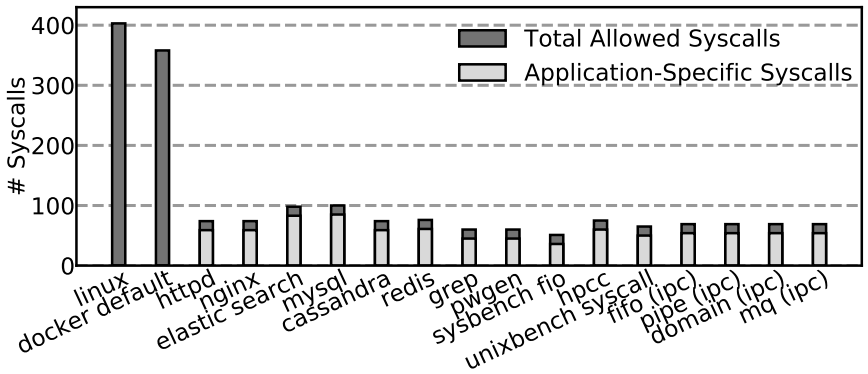


Figure 4.15: Number of system calls allowed.

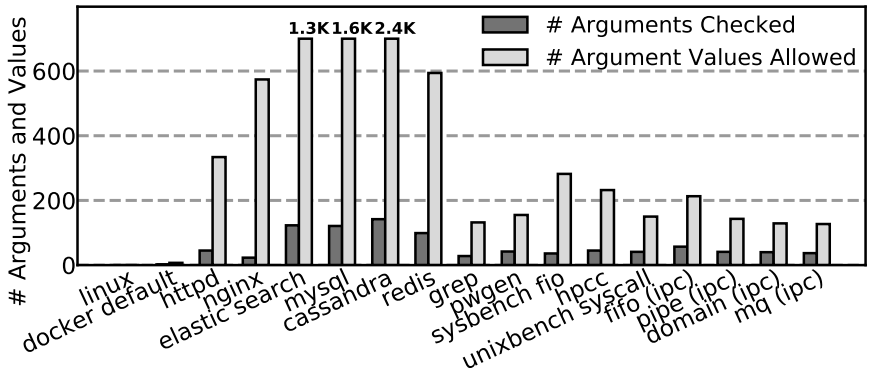


Figure 4.16: Number of system call arguments checked & values allowed.

4.12 OTHER RELATED WORK

There is a rich literature on system call checking [203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214]. Early implementations based on kernel tracing [203, 204, 205, 206, 207, 208, 209] incur excessive performance penalty, as every system call is penalized by at least two additional context switches.

A current proposal to reduce the overhead of Seccomp is to use a binary tree in `libseccomp`, to replace the branch instructions of current system call filters [142]. However, this does not fundamentally address the overhead. In Hromatka's own measurement, the binary tree-based optimization still leads to 2.4× longer system call execution time compared to Seccomp disabled [142]. Note that this result is without any argument checking. As shown in Section 4.4, argument checking brings further overhead, as it leads to more complex filter programs. Speculative methods [215, 216, 217] may not help reduce the overhead either, as their own overhead may surpass that of the checking code—they are designed for heavy security analysis such as virus scanning and taint analysis.

A few architectural security extensions have been proposed for memory-based protection, such

as CHERI [218, 219, 220], CODOMs [221, 222], PUMP [223], REST [224], and Califorms [225]. While related, Draco has different goals and resulting design—it checks system calls rather than load/store instructions, and its goal is to protect the OS.

4.13 CONCLUSION

To minimize system call checking overhead, we proposed *Draco*, a new architecture that caches validated system call IDs and argument values. System calls first check a special cache and, on a hit, are declared validated. We presented both a software and a hardware implementation of Draco. Our evaluation showed that the average execution time of macro and micro benchmarks with conventional Seccomp checking was 1.14× and 1.25× higher, respectively, than on an insecure baseline that performs no security checks. With software Draco, the average execution time was reduced to 1.10× and 1.18× higher, respectively, than on the insecure baseline. Finally, with hardware Draco, the execution time was within 1% of the insecure baseline.

We expect more complex security profiles in the near future. In this case, we found that the overhead of conventional Seccomp checking increases substantially, while the overhead of software Draco goes up only modestly. The overhead of hardware Draco remains within 1% of the insecure baseline.

Chapter 5: Elastic Cuckoo Page Tables

5.1 INTRODUCTION

Virtual memory is a cornerstone abstraction of modern computing systems, as it provides memory virtualization and process isolation. A central component of virtual memory is the page table, which stores the virtual-to-physical memory translations. The design of the page table has a significant impact on the performance of memory-intensive workloads, where working sets far exceed the TLB reach. In these workloads, frequent TLB misses require the fetching of the virtual-to-physical memory translations from the page table, placing page table look-ups on the critical path of execution.

The *de facto* current design of the page table, known as *radix page table*, organizes the translations into a multi-level tree [98, 226, 227]. The x86-64 architecture uses a four-level tree, while a fifth level will appear in next-generation architectures such as Intel's Sunny Cove [228, 229], and is already supported by Linux [230]. A radix page table can incur a high performance overhead because looking-up a translation involves a page table walk that *sequentially* accesses potentially all the levels of the tree from the memory system.

Despite substantial efforts to increase address translation efficiency with large and multi-level TLBs, huge page support, and Memory Management Unit (MMU) caches for page table walks, address translation has become a major performance bottleneck. It can account for 20–50% of the overall execution time of emerging applications [231, 232, 233, 234, 235, 236, 237, 238, 239, 240]. Further, page table walks may account for 20–40% of the main memory accesses [240]. Such overhead is likely to be exacerbated in the future, given that: (1) TLB scaling is limited by access time, space, and power budgets, (2) modern computing platforms can now be supplied with terabytes and even petabytes of main memory [241, 242], and (3) various memory-intensive workloads are rapidly emerging.

We argue that the radix page table was devised at a time of scarce memory resources that is now over. Further, its sequential pointer-chasing operation misses an opportunity: it does not exploit the ample memory-level parallelism that current computing systems can support.

For these reasons, in this chapter, we explore a fundamentally different solution to minimize the address translation overhead. Specifically, we explore a new page table structure that eliminates the pointer-chasing operation and uses parallelism for translation look-ups.

A natural approach would be to replace the radix page table with a *hashed page table*, which stores virtual-to-physical translations in a hash table [243, 244, 245, 246, 247]. Unfortunately, hashed page tables have been plagued by a number of problems. One major problem is the need

to handle hash collisions. Existing solutions to deal with hash collisions in page tables, namely collision chaining [248] and open addressing [249], require sequential memory references to walk over the colliding entries with special OS support. Alternatively, to avoid collisions, the hash page table needs to be dynamically resized. However, such operation is very costly and, hence, proposed hashed page table solutions avoid it by using a large global page table shared by all processes. Unfortunately, a global hashed page table cannot support page sharing between processes or multiple page sizes without adding an extra translation level.

To solve the problems of hashed page tables, this chapter presents a novel design called *Elastic Cuckoo Page Tables*. These page tables organize the address translations using *Elastic Cuckoo Hashing*, a novel extension of cuckoo hashing [175] designed to support gradual, dynamic resizing. Elastic cuckoo page tables efficiently resolve hash collisions, provide process-private page tables, support multiple page sizes and page sharing among processes, and efficiently adapt page table sizes dynamically to meet process demands. As a result, elastic cuckoo page tables transform the sequential pointer-chasing operation of traditional radix page tables into fully parallel look-ups, allowing address translation to exploit memory-level parallelism for the first time.

We evaluate elastic cuckoo page tables with full-system simulations of an 8-core processor running a set of graph analytics, bioinformatics, HPC, and system workloads. Elastic cuckoo page tables reduce the address translation overhead by an average of 41% over radix page tables. The result is a 3–18% speed-up in application execution.

5.2 BACKGROUND

5.2.1 Radix Page Tables

All current architectures implement *radix page tables*, where the page table is organized in a multi-level radix tree. The steps of address translation are described in Section 2.2.1. The process described is called a *page table walk*. It is performed in hardware on a TLB miss. A page table walk requires four *sequential* cache hierarchy accesses.

To increase the reach of the TLB, the x86-64 architecture supports two large page sizes, 2MB and 1GB. When a large page is used, the page table walk is shortened. Specifically, a 2MB page translation is obtained from the PMD table, while a 1GB page translation is obtained from the PUD table.

To alleviate the overhead of page table walks, the MMU of an x86-64 processor has a small cache or caches called Page Walk Caches (PWCs). The PWCs store recently-accessed PGD, PUD, and PMD table entries (but not PTE entries) [98, 233, 236, 250, 251]. On a TLB miss, before the

hardware issues any request to the cache hierarchy, it checks the PWCs. It records the lowest level table at which it hits. Then, it generates an access to the cache hierarchy for the next lower level table.

Struggles with Emerging Workloads

Emerging workloads, e.g., in graph processing and bioinformatics, often have multi-gigabyte memory footprints and exhibit low-locality memory access patterns. Such behavior puts pressure on the address translation mechanism. Recent studies have reported that address translation has become a major performance bottleneck [231, 232, 233, 234, 235, 236, 237, 238, 239, 240]. It can consume 20–50% of the overall application execution time. Further, page table walks may account for 20–40% of the main memory accesses [240].

To address this problem, one could increase the size of the PWCs to capture more translations, or increase the number of levels in the translation tree to increase the memory addressing capabilities. Sadly, neither approach is scalable. Like for other structures close to the core such as the TLB, the PWCs' access time needs to be short. Hence, the PWCs have to be small. They can hardly catch up with the rapid growth of memory capacity.

Increasing the number of levels in the translation tree makes the translation slower, as it may involve more cache hierarchy accesses. Historically, Intel has gradually increased the depth of the tree, going from two in the Intel 80386 to four in current processors [98, 226]. A five-level tree is planned for the upcoming Intel Sunny Cove [228, 229], and has been implemented in Linux [230]. This approach is not scalable.

5.2.2 Hashed Page Tables

The alternative to radix page tables is *hashed page tables*. Here, address translation involves hashing the virtual page number and using the hash key to index the page table. Assuming that there is no hash collision, only one memory system access is needed for address translation.

Hashed page tables [243, 244, 245, 246, 247] have been implemented in the IBM PowerPC, HP PA-RISC, and Intel Itanium architectures. The support for a hashed page table in the Itanium architecture was referred to as the long-format Virtual Hash Page Table (VHPT) [248, 252, 253]. In that design, the OS handles hash collisions. Upon a hash collision, the VHPT walker raises an exception that invokes the OS. The OS handler resolves the collision by searching collision chains and other auxiliary OS-defined data structures [236].

Challenges in Hashed Page Tables

Barr et al. [236] summarize three limitations of hashed page tables. The first one is the loss of spatial locality in the accesses to the page table. This is caused by hashing, which scatters the page table entries of contiguous virtual pages. The second limitation is the need to associate a hash tag (e.g., the virtual page number) with each page table entry, which causes page table entries to consume more memory space. The third one is the need to handle hash collisions, which leads to more memory accesses, as the system walks collision chains [236].

Yaniv and Tsafrir [249] recently show that the first two limitations are addressable by careful design of page table entries. Specifically, they use Page Table Entry Clustering, where multiple contiguous page table entries are placed together in a single hash table entry that has a size equal to a cache line. Further, they propose Page Table Entry Compaction, where unused upper bits of multiple contiguous page table entries are re-purposed to store the hash tag.

Unfortunately, hash collisions are a significant concern and remain unsolved. Existing strategies such as collision chaining [248] and open addressing [249] require expensive memory references needed to walk over the colliding entries.

To assess the importance of collisions, we take the applications of Section 5.7 and model a global hash table. We evaluate the following scenario: (1) the table has as many entries as the sum of all the translations required by all the applications, and (2) the hash function is the computationally-expensive BLAKE cryptographic function [254] which minimizes the probability of collisions.

Figure 5.1 shows the probability of random numbers mapping to the same hash table entry. The data is shown as a cumulative distribution function (CDF). The figure also shows the CDF for a global hash table that is over-provisioned by 50%. For the baseline table, we see that only 35% of the entries in the hash table have no collision (i.e, the number of colliding entries is 1). On the other hand, there are entries with a high number of collisions, which require time-consuming collision resolution operations. Even for the over-provisioned table, only half of the entries in the table have no collision.

Drawbacks of a Single Global Hash Table

One straightforward design for a hashed page table system is to have a single global hash table that includes page table entries from all the active processes in the machine. This design is attractive because 1) the hash table is allocated only once, and 2) the table can be sized to minimize the need for dynamic table resizing, which is very time consuming.

Sadly, such design has a number of practical drawbacks that make it undesirable [246, 249]. First, neither multiple page sizes (e.g., huge pages) nor page sharing between processes can be

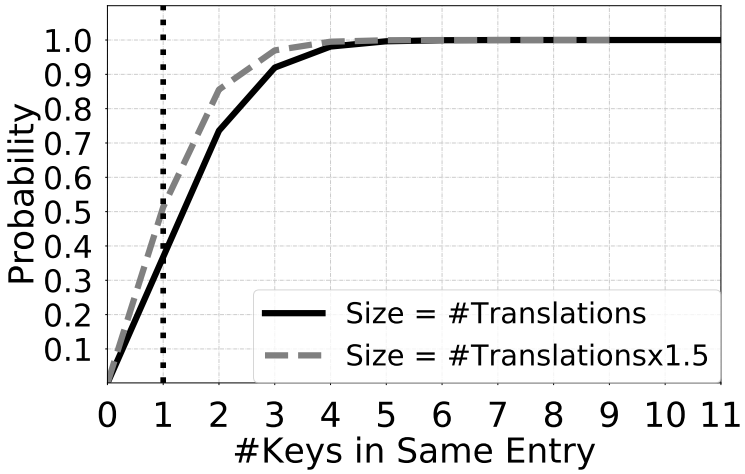


Figure 5.1: Cumulative distribution function of the number of keys mapping to the same hash table entry.

supported without additional complexity. For example, to support these two features, the IBM PowerPC architecture uses a two-level translation procedure for each memory reference [255]. Second, when a process is killed, the system needs to perform a linear scan of the entire hash table to find and delete the associated page table entries. Note that deleting an entry may also be costly: it may require a long hash table look-up (for open addressing) or a collision chain walk. Further, deleting a page table entry in open addressing may affect the collision probes in future look-ups.

Resizing Hashed Page Tables

To reduce collisions, hash table implementations set an *occupancy threshold* that, when reached, triggers the resizing of the table. However, resizing is an expensive procedure if done all at once. It requires allocating a new larger hash table and then, for each entry in the old hash table, rehash the tag with the new hash function and move the (tag, value) pair to the new hash table. In the context of page tables, the workload executing in the machine needs to pause and wait for the resizing procedure to complete. Further, since the page table entries are moved to new memory locations, their old copies cached in the cache hierarchy of the processor become useless. The correct copies are now in new addresses.

An alternative approach is to *gradually* move the entries, and maintain both the old and the new hash tables in memory for a period of time. Insertions place entries in the new hash table only, so the old hash table will eventually become empty and will then be deallocated. Further, after each insertion, the system also moves one or more entries from the old table to the new one.

Unfortunately, a look-up needs to access both the old and the new hash tables, since the desired entry could be present in either of them.

In the context of page tables, gradual rehashing has two limitations. First, keeping both tables approximately doubles the memory overhead. Second, a look-up has to fetch entries from both tables, doubling the accesses to the cache hierarchy. Unfortunately, half of the fetched entries are useless and, therefore, the caches may get polluted.

5.2.3 Cuckoo Hashing

Cuckoo hashing is a collision resolution algorithm that allows an element to have multiple possible hashing locations [175]. The element is stored in at most one of these locations at a time, but it can move between its hashing locations. Assume a cuckoo hash table with two hash tables or ways T_1 and T_2, indexed with hash functions H_1 and H_2, respectively. Because there are two tables and hash functions, the structure is called a *2-ary* cuckoo hash table. Insertion in cuckoo hashing places an element x in one of its two possible entries, namely $T_1[H_1(x)]$ or $T_2[H_2(x)]$. If the selected entry is occupied, the algorithm kicks out the current occupant y, and re-inserts y in y's other hashing location. If that entry is occupied, the same procedure is followed for its occupant. The insertion and eviction operations continue until no occupant is evicted or until a maximum number of displacements is reached (e.g., 32). The latter case is an insertion failure.

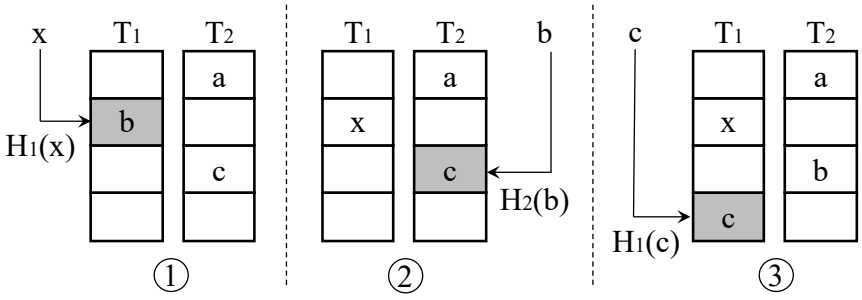


Figure 5.2: An example of insertion in cuckoo hashing.

A look-up in cuckoo hashing checks all the possible hashing locations of an element, and succeeds if it is found in either of them. In the example above, $T_1[H_1(x)]$ and $T_2[H_2(x)]$ are checked. The locations are checked *in parallel*. Hence, a look-up takes a constant time. A deletion operation proceeds like a look-up; then, if the element is found, it is removed.

Figure 5.2 shows an example of inserting element x into a 2-ary cuckoo hash table. Initially, in Step ①, the table has three elements: a, b, and c. The insertion of x at $T_1[H_1(x)]$ kicks out the previous occupant b. In Step ②, the algorithm inserts b in $T_2[H_2(b)]$, and kicks out c. Finally, in Step

③, a vacant entry is found for c. This example can be generalized to *d-ary* cuckoo hashing [256], which uses d independent hash functions to index d hash tables.

Like any hash table, the performance of cuckoo hash tables deteriorates with high occupancy. To gain insight, Figure 5.3 characterizes a d-ary cuckoo hash table (where $d \in \{2,3,4,8\}$) as a function of the occupancy. We take random numbers and, like before, hash them with the BLAKE cryptographic hash function [254]. The actual size of the cuckoo hash table does not matter, but only the hash table occupancy.

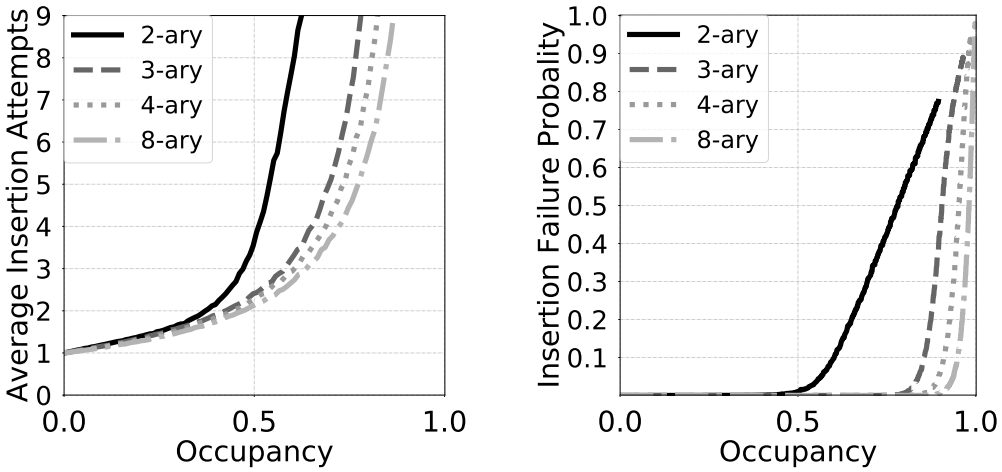


Figure 5.3: Characterizing d-ary cuckoo hashing.

Figure 5.3(a) shows the average number of insertion attempts required to successfully insert an element as a function of the table occupancy. We see that, for low occupancy, we either insert the key on the first attempt or require a single displacement. As occupancy increases, insertion performance deteriorates — e.g., after 50% occupancy in the 2-ary cuckoo, and after ∼70% in the 3, 4, and 8-ary cuckoo hash tables. Figure 5.3(b) shows the probability of insertion failures after 32 attempts. We see that the 2-ary cuckoo hash table has a non-zero probability of insertion failures after 50% occupancy, while the 3, 4, and 8-ary hash tables exhibit insertion failures only after 80% occupancy.

5.3 RETHINKING PAGE TABLES

As indicated in Section 5.2.1, radix page tables are not scalable. Further, as shown in Section 5.2.2, having a single global hashed page table is not a good solution either. We want to provide *process-private hashed page tables*, so that we can easily support page sharing among processes and multiple page sizes. However, a default-sized hashed page table cannot be very

large, lest we waste too much memory in some processes. Hence, we are forced to have modest-sized hashed page tables, which will suffer collisions.

One promising approach to deal with collisions is to use cuckoo hashing (Section 5.2.3). Unfortunately, any default-sized cuckoo hash table will eventually suffer insertion failures due to insufficient capacity. Hence, it will be inevitable to resize the cuckoo hash table.

Sadly, resizing cuckoo hash tables is especially expensive. Indeed, recall from Section 5.2.2 that, during gradual resizing, a look-up requires twice the number of accesses — since both the old and new hash tables need to be accessed. This requirement especially hurts cuckoo hashing because, during normal operation, a look-up into a d-ary cuckoo hash table already needs d accesses; hence, during resizing, a look-up needs to perform $2 \times d$ accesses.

To address this problem, in this chapter, we extend cuckoo hashing with a new algorithm for gradual resizing. Using this algorithm, during the resizing of a d-ary cuckoo hash table, a look-up *only requires d accesses*. In addition, the algorithm never fetches into the cache page table entries from the old hash table whose contents have already moved to the new hash table. Hence, it minimizes cache pollution from such entries. We name the algorithm *Elastic Cuckoo Hashing*. With this idea, we later build per-process *Elastic Cuckoo Page Tables* as our proposed replacement for radix page tables.

5.4 ELASTIC CUCKOO HASHING

5.4.1 Intuitive Operation

Elastic cuckoo hashing is a novel algorithm for cost-effective gradual resizing of d-ary cuckoo hash tables. It addresses the major limitations of existing gradual resizing schemes. To understand how it works, consider first how gradual resizing works with a baseline d-ary cuckoo hash table.

Cuckoo Hashing. Recall that the d-ary cuckoo hash table has d ways, each with its own hash function. We represent each way and hash function as T_i and H_i, respectively, where $i \in 1..d$. We represent the combined ways and functions as T_D and H_D, respectively. When the occupancy of T_D reaches a *Rehashing Threshold*, a larger d-ary cuckoo hash table is allocated. In this new d-ary table, we represent each way and hash function as T_i' and H_i', respectively, where $i \in 1..d$, and the combined ways and functions as T_D' and H_D', respectively.

As execution continues, a collision-free insert operation only accesses a randomly-selected single way of the new d-ary hash table — if there are collisions, multiple ways of the new d-ary hash table are accessed. In addition, after every insert, the system performs one *rehash* operation. A rehash consists of removing one element from the old d-ary hash table and inserting it into the new

d-ary hash table. Unfortunately, a look-up operation requires probing all d ways of both the old and the new d-ary hash tables, since an element may reside in any of the d ways of the two hash tables. When all the elements have been removed from the old d-ary table, the latter is deallocated.

Elastic Cuckoo Hashing. A d-ary *elastic* cuckoo hash table works differently. Each T_i way in the old d-ary hash table has a *Rehashing Pointer* P_i, where $i \in 1..d$. The set of P_i pointers is referred to P_D. At every T_i, P_i is initially zero. When the system wants to rehash an element from T_i, it removes the element pointed to by P_i, inserts the element into the new d-ary table, and increments P_i. At any time, P_i divides its T_i into two regions: the entries at lower indices than P_i (*Migrated Region*) and those at equal or higher indices than P_i (*Live Region*). Figure 5.4 shows the two regions for a 2-ary elastic cuckoo hash table. As gradual rehashing proceeds, the migrated regions in T_D keep growing. Eventually, when the migrated regions cover all the entries in T_D, the old d-ary table is deallocated.

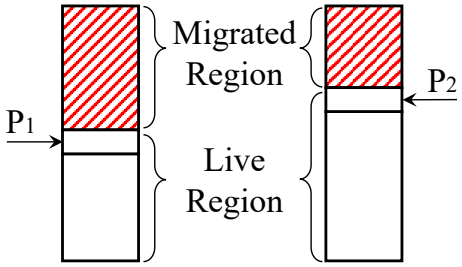


Figure 5.4: 2-ary elastic cuckoo hash table during resizing.

The insertion of an element in a d-ary elastic cuckoo hash table proceeds as follows. The system randomly picks one way from the old d-ary table, say T_i. The element is hashed with H_i. If the hash value falls in the live region of T_i, the element is inserted in T_i; otherwise, the element is hashed with the hash function H_i' of the *same way* T_i' of the *new* d-ary table, and the element is inserted in T_i'.

Thanks to this algorithm, a look-up operation for an element only requires d probes. Indeed, the element is hashed using all H_D hash functions in the old d-ary table. For each way i, if the hash value falls in the live region of T_i, T_i is probed; otherwise, the element is hashed with H_i', and T_i' in the new d-ary table is probed.

Elastic cuckoo hashing improves gradual resizing over cuckoo hashing in two ways. First, it performs a look-up with only d probes rather than $2 \times d$ probes. Second, it minimizes cache pollution by never fetching entries from the migrated regions of the old d-ary table; such entries are useless, as they have already been moved to the new d-ary table.

5.4.2 Detailed Algorithms

We now describe the elastic cuckoo algorithms in detail.

Rehash. A rehash takes the element pointed to by the rehashing pointer P_i of way T_i of the old d-ary hash table, and uses the hash function H_i' to insert it in the *same* way T_i' of the new d-ary table. P_i is then incremented.

Figure 5.5 shows an example for a 2-ary elastic cuckoo hash table. On the left, we see a hash table before rehashing, with P_1 and P_2 pointing to the topmost entries. On the right, we see the old and new hash tables after the first entry of T_1 has been rehashed. The system has moved element d from T_1 to T_1' at position $T_1'[H_1'(d)]$.

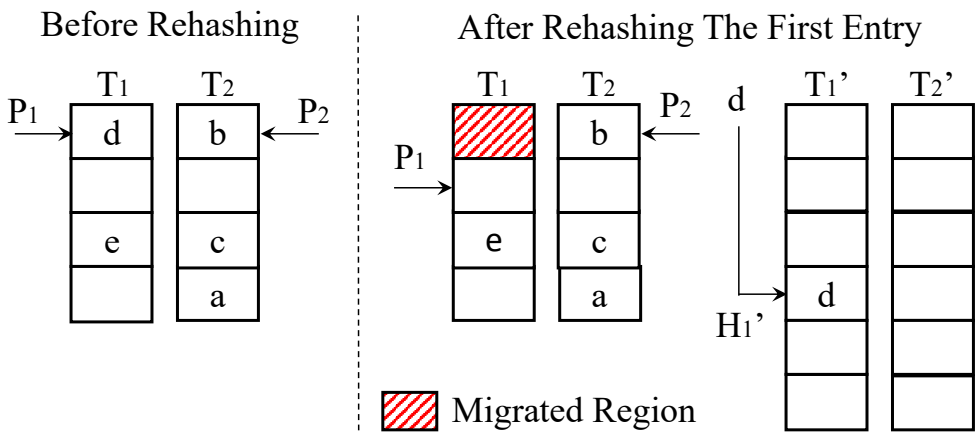


Figure 5.5: Example of the rehash operation.

Look-up. The look-up of an element x involves computing $H_i(x)$ for all the ways of the old d-ary hash table and, for each way, comparing the result to P_i. For each way i, if $H_i(x)$ belongs to the live region (i.e., $H_i(x) \geq P_i$), way T_i in the old hash table is probed with $H_i(x)$; otherwise, way T_i' in the new hash table is probed with $H_i'(x)$. The algorithm is:

Algorithm 5.1: Elastic Cuckoo Hashing Look-Up.

```
function LOOK-UP(x):
    for each way i in the old d-ary hash table do:
        if H_i(x) < P_i then:
            if T_i'[H_i'(x)] == x then return true;
        else:
            if T_i[H_i(x)] == x then return true;
    return false
```

As an example, consider 2-ary elastic cuckoo hash tables. A look-up of element x involves two probes. Which structure is probed depends on the values of P_1, P_2, $H_1(x)$, and $H_2(x)$. Figure 5.6 shows the four possible cases. The figure assumes that P_1 and P_2 currently point to the third and second entry of T_1 and T_2. If $H_1(x) \geq P_1 \wedge H_2(x) \geq P_2$ (Case ①), entries $T_1[H_1(x)]$ and $T_2[H_2(x)]$ are probed. If $H_1(x) < P_1 \wedge H_2(x) < P_2$ (Case ②), entries $T_1'[H_1'(x)]$ and $T_2'[H_2'(x)]$ are probed. If $H_1(x) < P_1 \wedge H_2(x) \geq P_2$ (Case ③), entries $T_1'[H_1'(x)]$ and $T_2[H_2(x)]$ are probed. Finally, if $H_1(x) \geq P_1 \wedge H_2(x) < P_2$ (Case ④), $T_1[H_1(x)]$ and $T_2'[H_2'(x)]$ are probed.

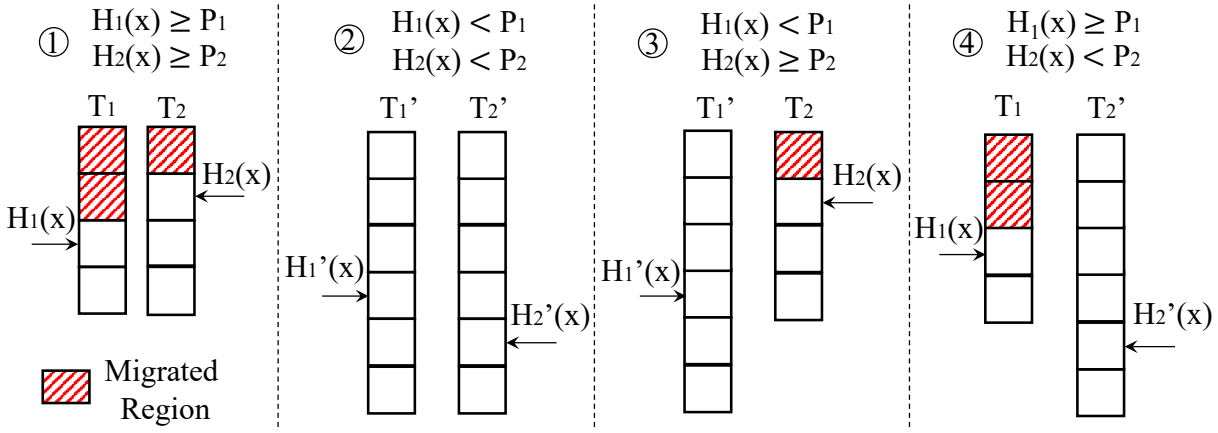


Figure 5.6: Different cases of the look-up operation.

Observation 5.1 Given the parallelism in a look-up operation, a look-up during resizing takes the time needed to perform a hash of an element, a comparison to the value in the rehashing pointer, and either a probe in one way of the old hash table or a second hash and a probe in one way of the new hash table.

Delete. A delete follows the look-up procedure and, on finding the element, clears the entry. Therefore, it takes the same time as a look-up plus a write to the target way.

Insert. The insert of an element x involves randomly picking one way i in the old d-ary hash table and checking if $H_i(x) < P_i$ is true. If it is not, the element is inserted at $T_i[H_i(x)]$; otherwise, the element is inserted in the same way of the new d-ary hash table at $T_i'[H_i'(x)]$. In either case, if the insertion causes the eviction of another element y, the system randomly picks a different way than the one just updated, and repeats the same procedure for y. This process may repeat multiple times, every time picking a different way than *the immediately previous* one. It terminates when either an insertion does not cause any eviction or a maximum number of iterations is reached.

The following algorithm describes the insert operation. In the algorithm, RAND_PICK returns a random way from a set of ways. We use $x \leftrightarrow y$ to denote the swapping of the values x and y, and use $\perp$ to denote an empty value.

Algorithm 5.2: Elastic Cuckoo Hashing Insert.

```
function INSERT(x):
    i ← RAND_PICK({1, ··· , d});
    for loop = 1 to MAX_ATTEMPTS do:
        if H_i(x) < P_i then:
            x ↔ T'_i[H'_i(x)];
            if x == ⊥ then return true;
        else:
            x ↔ T_i[H_i(x)];
            if x == ⊥ then return true;
        i ← RAND_PICK({1, ··· , d} − {i});
    return false
```

Hash Table Resize. Two parameters related to elastic cuckoo hash table resizing are the *Rehashing Threshold* (r_t) and the *Multiplicative Factor* (k). When the fraction of a hash table that is occupied reaches r_t, the system triggers a hash table resize, and a new hash table that is k times bigger than the old one is allocated.

We select a r_t that results in few insertion collisions and a negligible number of insertion failures. As shown in Figure 5.3, for 3-ary tables, a good r_t is 0.6 or less. We select a k that is neither so large that it wastes substantial memory nor so small that it causes continuous resizes. Indeed, if k is too small, as entries are moved into the new hash table during resizing, the occupancy of the new hash table may reach the point where it triggers a new resize operation.

Later in the section we show how to set k. Specifically we show that $k > (r_t + 1)/r_t$. For example, for $r_t = 0.4$, $k > 3.5$; hence $k = 4$ is good. For $r_t = 0.6$, $k > 2.6$; hence $k = 3$ is good. However, a k equal to a power of two is best for hardware simplicity.

To minimize collisions in the old hash table during resizing, it is important to limit the occupancy of the live region during resizing. Our algorithm tracks the fraction of used entries in the live region. As long as such fraction does not exceed the one for the whole table that triggered the resizing, each insert is followed by only a single rehash. Otherwise, each insert is followed by the rehashing of multiple elements until the live region falls back to the desired fraction of used entries.

We select the Rehashing Threshold to be low enough that the frequency of insertion failures is negligible. However, insertion failures can still occur. If an insertion failure occurs outside a resize, we initiate resizing. If an insertion failure occurs during a resize, multiple elements are rehashed

into other ways and then the insertion is tried again. In practice, as we show in Section 5.8.1, by selecting a reasonable Rehashing Threshold, we completely avoid insertion failures.

Elastic cuckoo hash tables naturally support downsizing when the occupancy of the tables falls below a given threshold. We use a *Downsizing Threshold* (d_t) and reduce the table by a *Downsizing Factor* (g). For gradual downsizing, we use an algorithm similar to gradual resizing.

Finding a Good Value for the Multiplicative Factor To find a good value for the Multiplicative Factor k in a resize operation, we compute the number of entries that the new table receives during the resize operation. To simplify the analysis, we neglect insertion collisions, which move elements from one way to another way of the same table. Hence, we only need to consider 1 way of the old table (which has T entries) and 1 way of the new table (which has $k \times T$ entries). A similar analysis can be performed considering collisions, following the procedure outlined in [175, 256].

Recall that, during resizing, an insert operation can insert the entry in the old hash table (*Case Old*) or in the new hash table (*Case New*). In either case, after the insert, one entry is moved from the old to the new table, and the Rehashing Pointer is advanced. Therefore, in Case Old, the new table receives one entry; in Case New, it receives two.

If all the inserts during resizing were of Case Old, the new table would receive at most T elements during resizing, since the Rehashing Pointer would by then reach the end of the old table. If all the inserts during resizing were of Case New, there could only be $r_t \times T$ insertions, since the old table had $r_t \times T$ elements and, by that point, all elements would have moved to the new table. Hence, the new table would receive $2r_t \times T$ elements during resizing.

The worst case occurs in a resize that contains some Case Old and some Case New inserts. The scenario is as follows. Assume that all the $r_t \times T$ elements in the old table are at the top of the table. During resizing, there are first $r_t \times T$ Case New inserts, and then $(1 - r_t) \times T$ Case Old inserts. Hence, the new table receives $(r_t + 1) \times T$ elements. This is the worst case. Hence, to avoid reaching the point where a new resize operation is triggered inside a resize operation, it should hold that $(r_t + 1) \times T < r_t \times k \times T$, or $k > (r_t + 1)/r_t$.

5.5 ELASTIC CUCKOO PAGE TABLE DESIGN

Elastic cuckoo page tables are process-private hashed page tables that scale on demand according to the memory requirements of the process. They resolve hash collisions and support multiple page sizes and page sharing among processes. In this section, we describe their organization, the cuckoo walk tables, and the cuckoo walk caches.

5.5.1 Elastic Cuckoo Page Table Organization

An elastic cuckoo page table is organized as a d-ary elastic cuckoo hash table that is indexed by hashing a Virtual Page Number (VPN) tag. A process in a core has as many elastic cuckoo page tables as page sizes. Figure 5.7 shows the elastic cuckoo page tables of a process for the page sizes supported by x86: 1GB, 2MB, and 4KB. In the figure, each page table uses a 2-ary elastic cuckoo hash table. The tables are named using x86 terminology: PUD, PMD, and PTE. Each table entry contains multiple, consecutive page translation entries. In the example, we assume 8 of them per table entry. Hence, using an x86-inspired implementation, the tables are indexed by the following VA bits: 47-33 for PUD, 47-24 for PMD, and 47-15 for PTE. All the hash functions are different.

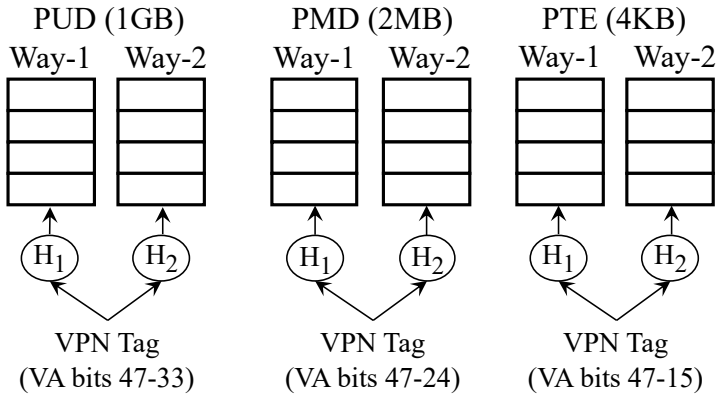


Figure 5.7: Elastic cuckoo page tables for a process.

By using multiple independent page tables, this design can support any page sizes. This is contrast to radix page tables which, due to the way they perform the translation, can only support a few rigid page sizes.

In addition, this design can attain high performance by exploiting two levels of parallelism. The first one is between tables of different page sizes. Each elastic cuckoo page table is independent from the others and can be looked-up in parallel. This is in contrast to radix page tables, where each tree level has to be walked sequentially. The second level of parallelism is between the different d ways of a d-ary table. A translation may reside in any of the d ways. Hence, all ways can be accessed in parallel. Overall, virtual page translation using elastic cuckoo page tables can exploit the memory-level parallelism provided by modern processors.

Page Table Entry

The entries in an elastic cuckoo page table use the ideas of page table entry clustering and compaction [243, 249]. The goal is to improve spatial locality and to reduce the VPN tag over-

head. Specifically, a single hash table entry contains a VPN tag and multiple consecutive physical page translation entries packed together. The number of such entries packed together is called the clustering factor, and is selected to make the tag and entries fit in one cache line.

In machines with 64-byte cache lines, we can cluster eight physical page translation entries and a tag in a cache line. This is feasible with the compaction scheme proposed in [249], which re-purposes some unused bits from the multiple contiguous translations to encode the tag [98, 249].

As an example, consider placing in a cache line eight PTE entries of 4KB pages, which require the longest VPN tag — i.e, the 33 bits corresponding to bits 47–15 of the VA. In an x86 system, a PTE typically uses 64 bits. To obtain these 33 bits for the tag, we need to take 5 bits from each PTE and re-purpose them as tag. From each PTE, our implementation takes 4 bits that the Linux kernel currently sets aside for the software to support experimental uses [257], and 1 bit that is currently used to record the page size — i.e., whether the page size is 4KB or more. The latter information is unnecessary in elastic cuckoo page tables. With these choices, we place eight PTE entries and a tag in a cache line. We can easily do the same for the PMD and PUD tables, since we only need to take 3 and 2 bits, respectively, from each physical page translation entry in these hash tables.

Cuckoo Walk

We use the term *Cuckoo Walk* to refer to the procedure of finding the correct translation in elastic cuckoo page tables. A cuckoo walk fundamentally differs from the sequential radix page table walk: it is a *parallel* walk that may look-up multiple hash tables in parallel. To perform a cuckoo walk, the hardware page table walker takes a VPN tag, hashes it using the hash functions of the different hash tables, and then uses the resulting keys to index multiple hash tables in parallel.

As an example, assume that we have a 2-ary PTE elastic cuckoo page table. Figure 5.8 illustrates the translation process starting from a VPN tag. Since the clustering factor is 8, the VPN tag is bits 47–15 of the VA. These bits are hashed using the two hash functions H_1 and H_2. The resulting values are then added to the physical addresses of the bases of the two hash tables. Such bases are stored in control registers. To follow x86 terminology, we call these registers $\texttt{CR3-PageSize}_j\texttt{-way}_i$. The resulting physical addresses are accessed and a hit is declared if any of the two tags match the VPN tag. On a hit, the PTE Offset (bits 14-12) is used to index the hash table entry and obtain the desired PTE entry.

Assume that the system supports S different page sizes. In a translation, if the size of the page requested is unknown, the hardware has to potentially look-up the d ways of each of the S elastic cuckoo page tables. Hence, a cuckoo walk may need to perform up to $S \times d$ parallel look-ups. In the next section, we show how to substantially reduce this number.

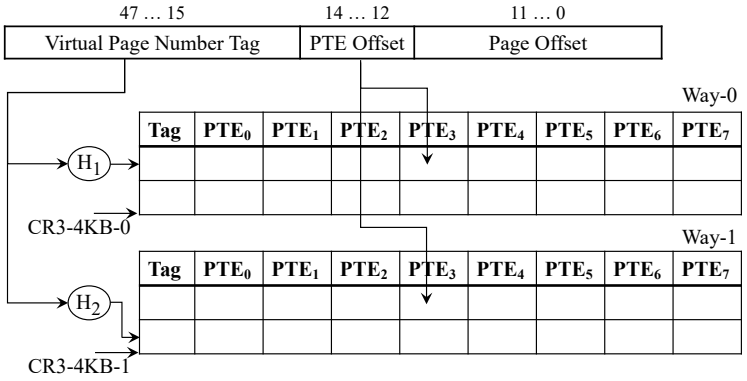


Figure 5.8: Accessing a 2-ary PTE elastic cuckoo page table.

Similarly to radix page table entries, the entries of elastic cuckoo page tables are cached on demand in the cache hierarchy. Such support accelerates the translation process, as it reduces the number of requests sent to main memory.

5.5.2 Cuckoo Walk Tables

We do not want cuckoo page walks to have to perform $S \times d$ parallel look-ups to obtain a page translation entry. To reduce the number of look-ups required, we introduce the *Cuckoo Walk Tables* (CWTs). These software tables contain information about which way of which elastic cuckoo page table should be accessed to obtain the desired page translation entry. The CWTs are updated by the OS when the OS performs certain types of updates to the elastic cuckoo page tables (Section 5.5.2). The CWTs are automatically read by the hardware, and effectively prune the number of parallel look-ups required to obtain a translation.

Using the CWTs results in the four types of cuckoo walks shown in Figure 5.9. The figure assumes three page sizes (1GB, 2MB, and 4KB) and 2-ary elastic cuckoo page tables.

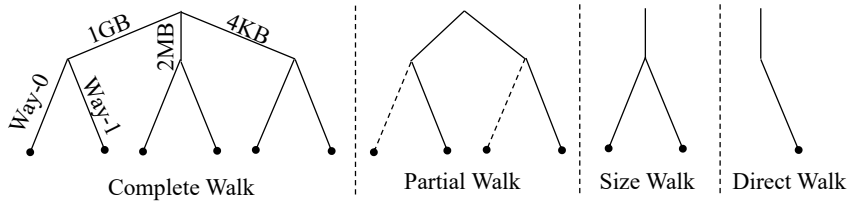


Figure 5.9: Different types of cuckoo walks.

Complete Walk: In this case, the CWTs provide no information and, hence, the hardware accesses all the ways of the elastic cuckoo page tables of all the page sizes.

Partial Walk: The CWTs indicate that the page is not of a given size. Hence, the hardware accesses potentially all the ways of the rest of the elastic cuckoo page tables. In some cases, the search of some of the ways may be avoided. This is represented in the figure with dashed lines.
Size Walk: The CWTs indicate that the page is of a given size. As a result, the hardware accesses all the ways of a single elastic cuckoo page table.
Direct Walk: The CWTs indicate the size of the page and which way stores the translation. In this case, only one way of a single elastic cuckoo page table is accessed.

Ideally, we have one CWT associated with each of the elastic cuckoo page tables. In our case, this means having a PUD-CWT, a PMD-CWT, and a PTE-CWT, which keep progressively finer-grain information. These tables are accessed in sequence, and each may provide more precise information than the previous one.

However, these software tables reside in memory. To make them accessible with low latency, they are cached in special caches in the MMU called the *Cuckoo Walk Caches*. These caches replace the page walk caches of radix page tables. They are described in Section 5.5.3. In our design, we find that caching the PTE-CWT would provide too little locality to be profitable — an observation consistent with the fact that the current page walk caches of radix page tables do not cache PTE entries. Hence, we only have PUD-CWT and PMD-CWT tables, and cache them in the cuckoo walk caches.

The PUD-CWT and PMD-CWT are updated by OS threads using locks. They are designed as d-ary elastic cuckoo hash tables like the page tables. We discuss the format of their entries next.

Cuckoo Walk Table Entries

An entry in a CWT contains a VPN tag and several consecutive *Section Headers*, so that the whole CWT entry consumes a whole cache line. A section header provides information about a given Virtual Memory *Section*. A section is the range of virtual memory address space translated by one entry in the corresponding elastic cuckoo page table. A section header specifies the sizes of the pages in that section and which way in the elastic cuckoo page table holds the translations for that section.

To make this concept concrete, we show the exact format of the entries in the PMD-CWT. Figure 5.10 shows that an entry is composed of a VPN tag and 64 section headers. Each section header provides information about the virtual memory section mapped by one entry in the PMD elastic cuckoo page table. For example, Figure 5.10 shows a shaded section header which provides information about the virtual memory section mapped by the shaded entry in the PMD elastic cuckoo page table (shown in the bottom of the figure). Such entry, as outlined in Section 5.5.1, includes a VPN tag and 8 PMD translations to fill a whole cache line.

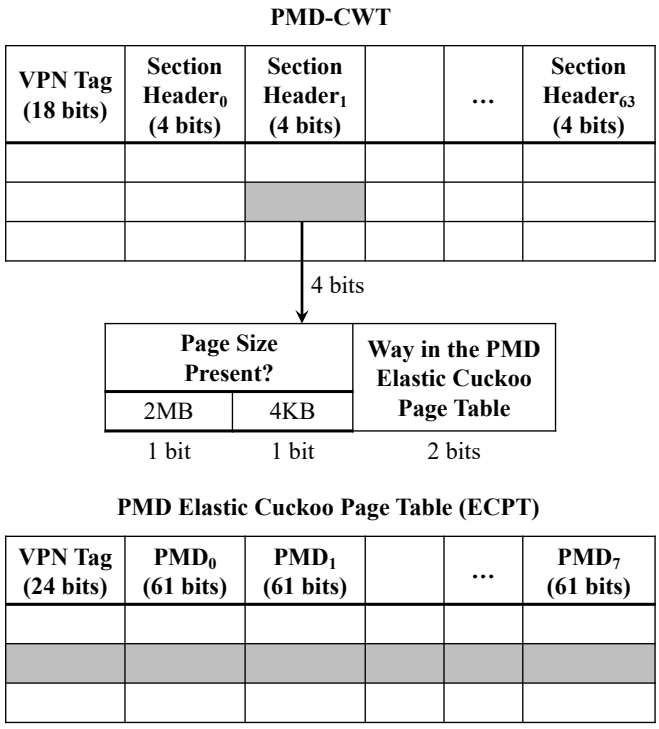


Figure 5.10: PMD-CWT entry layout.

Since a PMD page is 2MB, and a virtual memory section (i.e., a row or entry in the PMD elastic cuckoo page table of Figure 5.10) may map up to 8 PMD pages, a virtual memory section comprises 16MB. Note that part or all of this virtual memory section may be populated by 4KB pages (whose translations are in the PTE elastic cuckoo page table) rather than by 2MB pages.

Then, a PMD-CWT row or entry provides information for 64 of these sections, which corresponds to a total of 1 GB. Hence, the VPN tag of a PMD-CWT entry contains bits 47-30 of the virtual address. Given a 64-byte line, this design allows at least 4 bits for each section header of a PMD-CWT entry — but we cannot have more section headers in a line.

These 4 bits encode the information shown in Figure 5.10. The first bit (*2MB Bit*) indicates whether this memory section maps one or more 2MB pages. The second bit (*4KB Bit*) says whether the section maps one or more 4KB pages. The last two bits (*Way Bits*) are only meaningful if the 2MB Bit is set. They indicate the way of the PMD elastic cuckoo page table that holds the mapped translations of the 2MB pages in this virtual memory section. Note this encoding assumes that the elastic cuckoo page tables have at most four ways.

Table 5.1 shows the actions taken by the hardware page walker when it reads the target section header from the PMD-CWT. If the 2MB and 4KB Bits are clear, no 2MB or 4KB page is mapped in this section. Therefore, the walker does not access the PMD or PTE elastic cuckoo page tables. If the 2MB Bit is set and the 4KB Bit is clear, the walker performs a *Direct Walk* in the PMD elastic

cuckoo page table using the way indicated by the Way Bits. Only *one access* is needed because all the translation information for this section is present in a single entry of the PMD table.

Page Size Present?		Way in the PMD ECPT	Action
2MB	4KB		
0	0	X	No access to PMD ECPT or PTE ECPT
1	0	i	Direct Walk in PMD ECPT Way i
0	1	X	Size Walk in PTE ECPT
1	1	i	Partial Walk: Direct Walk in PMD ECPT Way i and Size Walk in PTE ECPT

Table 5.1: Actions taken by the page walker for the different values of a PMD-CWT section header. In the table, ECPT means elastic cuckoo page table.

If the 2MB Bit is clear and the 4KB Bit is set, all the translations for this section are in the PTE elastic cuckoo page table. Sadly, there is no information available in the PMD-CWT section header about which way(s) of the PTE elastic cuckoo page table should be accessed. Hence, the walker performs a *Size Walk* in the PTE elastic cuckoo page table. Finally, if both the 2MB and 4KB Bits are set, the target page could have either size. Hence, the walker performs a *Partial Walk*. The walk is composed of a *Direct Walk* in the PMD elastic cuckoo page table (using the way indicated by the Way Bits), and a *Size Walk* in the PTE elastic cuckoo page table.

The PUD-CWT is organized in a similar manner. Each section header now covers a virtual memory section of 8GB. A section header has 5 bits: a *1GB Bit*, a *2MB Bit*, and a *4KB Bit* to indicate whether the section maps one or more 1GB pages, one or more 2MB pages, and/or one or more 4KB pages, respectively; and two *Way Bits* to indicate the way of the PUD elastic cuckoo page table that holds the mapped translations of the 1GB pages in this virtual memory section. The Way Bits are only meaningful if the 1GB Bit is set. The actions taken based on the value of these bits are similar to those for the PMD-CWT. To simplify, we do not detail them. Overall, our design encodes substantial information about virtual memory sections with only a few bits in the CWTs.

With this encoding of the PUD-CWT and PMD-CWT, the OS updates these tables infrequently. Most updates to the elastic cuckoo page tables do not require an update to the CWTs. For example, take a section header of the PMD-CWT. Its 2MB Bit and 4KB Bit are only updated the *first time* that a 2MB page or a 4KB page, respectively, is allocated in the section. Recall that a section's size is equal to 4096 4KB pages. Also, the Way Bits of the section header in the PMD-CWT are not updated when a 4KB page is allocated or rehashed.

Finally, the conceptual design is that, on a page walk, the hardware accesses the PUD-CWT first, then the PMD-CWT and, based on the information obtained, issues a reduced number of page table accesses. The actual design, however, is that the CWT information is cached in and

read from small caches in the MMU that are filled off the critical path. We describe these caches next.

5.5.3 Cuckoo Walk Caches

The PUD-CWT and PMD-CWT reside in memory and their entries are cacheable in the cache hierarchy, like elastic cuckoo page table entries. However, to enable very fast access on a page walk, our design caches some of their entries on demand in *Cuckoo Walk Caches* (CWCs) in the MMU. We call these caches PUD-CWC and PMD-CWC, and replace the page walk caches of radix page tables. The hardware page walker checks the CWCs before accessing the elastic cuckoo page tables and, based on the information in the CWCs, it is able to issue fewer parallel accesses to the page tables.

The PUD-CWC and PMD-CWC differ in a crucial way from the page walk caches of radix page tables: their contents (like those of the CWTs) are *decoupled* from the contents of the page tables. The CWCs store page size and way information. This is unlike in the conventional page walk caches, which store page table entries. As a result, the CWCs and CWTs can be accessed independently of the elastic cuckoo page tables. This fact has two implications.

The first one is that, on a CWC *miss*, the page walker can proceed to access the target page table entry *right away* — albeit by issuing more memory accesses in parallel than otherwise. After the page walk has completed, the TLB has been filled, and execution has restarted, the appropriate CWT entries are fetched and cached in the CWCs, off the critical path. Instead, in the conventional page walk caches, the entries in the page walk caches have to be sequentially generated on the critical path before the target page table entry can be accessed and the TLB can be filled.

The second implication is that a CWC entry is very small. It only includes a few page size and way bits. This is unlike an entry in conventional page walk caches, which needs to include the physical address of the next level of page translation, in addition to the page size and other information. For example, a PMD-CWC section header covers a 16MB region (4096 4KB pages) with only 4 bits, while an entry in the traditional PMD page walk cache only covers a 2MB region (512 4KB pages) using 64 bits. The result is that CWCs have a small size and a very high hit rate.

5.6 IMPLEMENTATION

5.6.1 Steps of a Page Translation

Figure 5.11 presents the steps of a page translation with Elastic Cuckoo Page Tables (ECPTs). For simplicity, we assume that there is no 1GB page. On a TLB miss, the page table walker

hardware first checks the PUD-CWC ①. If the PUD-CWC hits, the corresponding section header is examined. It can indicate that the section contains only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. In the case of only 4KB pages, the page walker performs a Size Walk in the PTE ECPT ⑤, which will either bring the translation to the TLB or trigger a page fault. In the case of a section with only 2MB pages or with both 2MB and 4KB pages, the PMD-CWC is accessed to obtain additional information ⑧.

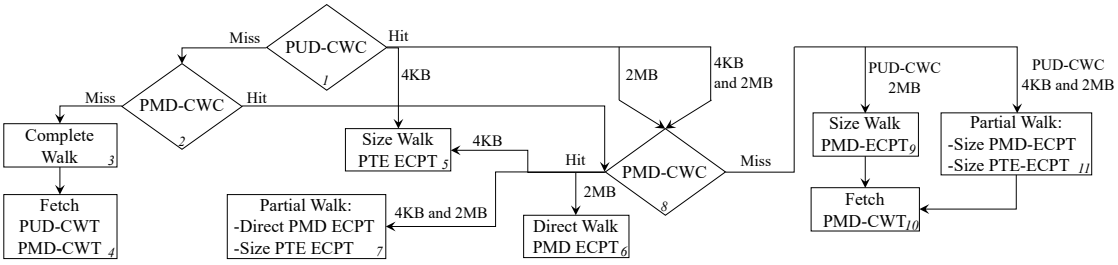


Figure 5.11: Steps of a page translation. ECPT stands for Elastic Cuckoo Page Tables.

If the PMD-CWC hits, the smaller section accessed may again contain only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. In the first case, the page walker performs a Size Walk in the PTE ECPT ⑤; in the second case, it performs a Direct Walk in the PMD ECPT ⑥; and in the third case, it performs a Partial Walk ⑦, which includes a Direct Walk in the PMD ECPT and a Size Walk in the PTE ECPT.

If, instead, the PMD-CWC misses, the walker uses the partial information provided by the PUD-CWC. Specifically, if the PUD section contained only 2MB pages, the walker performs a Size Walk in the PMD ECPT ⑨ (since there is no information on the PMD ECPT ways); if it contained both 2MB and 4KB pages, it performs a Partial Walk ⑪, which includes Size Walks in both the PMD ECPT and the PTE ECPT . In both cases, after the TLB is filled and execution resumes, the hardware fetches the missing PMD-CWT entry into the PMD-CWC ⑩.

A final case is when the access to the PUD-CWC misses. The walker still accesses the PMD-CWC to see if it can obtain some information ②. If the PMD-CWC hits, the section accessed may contain only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. The walker proceeds with actions ⑤, ⑥, and ⑦, respectively. If the PMD-CWC misses, the walker has no information, and it performs a Complete Walk to bring the translation into the TLB ③. After that, the hardware fetches the missing PUD-CWT and PMD-CWT entries into the PUD-CWC and PMD-CWC, respectively ④.

Since we are not considering 1GB pages in this discussion, on a PUD-CWC miss and PMD-CWC hit, there is no need to fetch the PUD-CWT entry. If 1GB pages are considered, the hardware needs to fetch the PUD-CWT entry in this case.

5.6.2 Concurrency Issues

Elastic cuckoo page tables naturally support multi-process and multi-threaded applications and, like radix page tables, abide by the Linux concurrency model regarding page table management. Specifically, multiple MMU page walkers can perform page table look-ups while one OS thread performs page table updates.

When a page table entry is found to have the Present bit clear, a page fault occurs. During the page fault handling, other threads can still perform look-up operations. Our scheme handles these cases like in current radix page tables in Linux: when the OS updates the elastic cuckoo page tables or the CWTs, it locks them. Readers may temporarily get stale information from their CWCs, but they will eventually obtain the correct state.

The OS uses synchronization and atomic instructions to insert entries in the elastic cuckoo page tables, to move elements across ways in a page table, to move entries across page tables in a resizing operation, to update the Rehashing Pointers in a resizing operation, and to update the CWTs.

If the CWCs were coherent, updates to CWT entries would invalidate entries in the CWCs. In our evaluation, we do not make such assumption. In this case, when the page walker uses information found in CWCs to access a translation and the accesses fail, the walker performs the walk again. This time, however, it accesses the *remaining* ways of the target elastic cuckoo page table or, if a resize is in progress, the remaining ways of both tables. This action will find entries that have been moved to another way. After the translation is obtained, the stale entries in the CWCs are refreshed. A similar process is followed for the Rehashing Pointers.

5.7 EVALUATION METHODOLOGY

Modeled Architectures. We use full-system cycle-level simulations to model a server architecture with 8 cores and 64 GB of main memory. We model a baseline system with 4-level radix page tables like the x86-64 architecture, and our proposed system with elastic cuckoo page tables. We model these systems (i) with only 4KB pages, and (ii) with multiple page sizes by enabling Transparent Huge Pages (THP) in the Linux kernel [258]. We call these systems *Baseline 4KB*, *Cuckoo 4KB*, *Baseline THP*, and *Cuckoo THP*.

The architecture parameters of the systems are shown in Table 5.2. Each core is out-of-order and has private L1 and L2 caches, and a portion of a shared L3 cache. A core has private L1 and L2 TLBs and page walk caches for translations. The Cuckoo architectures use 3-ary elastic cuckoo hashing for the Elastic Cuckoo Page Tables (ECPTs) and 2-ary elastic cuckoo hashing for the CWTs. The Baseline architectures use the 4-level radix page tables of x86-64. We size the

CWCs in the Cuckoo architecture to consume less area than the Page Walk Caches (PWC) in the Baseline architectures, which are modeled after x86-64. Specifically, the combined 2 CWCs have 18 entries of 32B each, for a total of 576B; the combined 3 PWCs have 96 entries of 8B each, for a total of 768B.

Processor Parameters	
Multicore chip	8 OoO cores, 256-entry ROB, 2GHz
L1 cache	32KB, 8-way, 2 cycles round trip (RT), 64B line
L2 cache	512KB, 8-way, 16 cycles RT
L3 cache	2MB per core, 16-way, 56 cycles RT
Per-Core MMU Parameters	
L1 DTLB (4KB pages)	64 entries, 4-way, 2 cycles RT
L1 DTLB (2MB pages)	32 entries, 4-way, 2 cycles RT
L1 DTLB (1GB pages)	4 entries, 2 cycles RT
L2 DTLB (4KB pages)	1024 entries, 12-way, 12 cycles RT
L2 DTLB (2MB pages)	1024 entries, 12-way, 12 cycles RT
L2 DTLB (1GB pages)	16 entries, 4-way, 12 cycles RT
PWC	3 levels, 32 entries/level, 4 cycles RT
Elastic Cuckoo Page Table (ECPT) Parameters	
Initial PTE ECPT size	16384 entries $\times$ 3 ways
Initial PMD ECPT size	16384 entries $\times$ 3 ways
Initial PUD ECPT size	8192 entries $\times$ 3 ways
Initial PMD-CWT size	4096 entries $\times$ 2 ways
Initial PUD-CWT size	2048 entries $\times$ 2 ways
Rehashing Threshold	$r_t = 0.6$
Multiplicative Factor	$k = 4$
PMD-CWC; PUD-CWC	16 entries, 4 cycles RT; 2 entries, 4 cycles RT
Hash functions: CRC	Latency: 2 cycles; Area: $1.9 * 10^{-3} mm$
	Dyn. energy: $0.98 pJ$; Leak. power: $0.1 mW$
Main-Memory Parameters	
Capacity; #Channels; #Banks	64GB; 4; 8
t_{RP}-t_{CAS}-t_{RCD}-t_{RAS}	11-11-11-28
Frequency; Data rate	1GHz; DDR

Table 5.2: Architectural parameters used in the evaluation.

Modeling Infrastructure. We integrate the Simics [198] full-system simulator with the SST framework [199, 259] and the DRAMSim2 [200] memory simulator. We use Intel SAE [201] on Simics for OS instrumentation. We use CACTI [100] for energy and access time evaluation of memory structures, and the Synopsys Design Compiler [202] for evaluating the RTL implementation of the hash functions. Simics provides the actual memory and page table contents for each memory address. We model and evaluate the hardware components of elastic cuckoo page tables in detail using SST.

Workloads. We evaluate a variety of workloads that experience different levels of TLB pressure. They belong to the graph analytics, bioinformatics, HPC, and system domains. Specifically, we use eight graph applications from the GraphBIG benchmark suite [260]. Two of them are social graph analysis algorithms, namely Betweenness Centrality (BC) and Degree Centrality (DC); two are graph traversal algorithms, namely Breadth-Fist Search (BFS) and Depth-First Search (DFS); and four are graph analytics benchmarks for topological analysis, graph search/flow and website

relevance, namely Single Source Shortest Path (SSSP), Connected Components (CC), Triangle Count (TC), and PageRank (PR). From the bioinformatics domain, we use MUMmer from the BioBench suite [261], which performs genome-level alignment. From the HPC domain, we use GUPS from HPC Challenge [262], which is a random access benchmark that measures the rate of integer random memory updates. Finally, from the system domain, we select the Memory benchmark from the SysBench suite [189], which stresses the memory subsystem. We call it SysBench.

The memory footprints of these workloads are: 17.3GB for BC, 9.3GB for DC, 9.3GB for BFS, 9GB for DFS, 9.3GB for SSSP, 9.3GB for CC, 11.9GB for TC, 9.3GB for PR, 6.9GB for MUMmer, 32GB for GUPS, and 32GB for SysBench.

For each individual workload, we perform full-system simulations for all the different configurations evaluated. When the region of interest of the workload is reached, the detailed simulations start. We warm-up the architectural state for 50 million instructions per core, and then measure 500 million instructions per core.

5.8 EVALUATION

5.8.1 Elastic Cuckoo Hashing Characterization

Figure 5.12 characterizes the behavior of elastic cuckoo hashing. It shows the average number of insertion attempts to successfully insert an element (left), and the probability of insertion failure after 32 attempts (right), both as a function of the table occupancy. We use 3-ary elastic cuckoo hashing and the BLAKE hash function [254]. Unlike in Figure 5.3, in this figure, the occupancy goes beyond one. The reason is that, when occupancy reaches the rehashing threshold $r_t = 0.6$, we allocate a new hash table with a multiplicative factor $k = 4$ and perform gradual resizing. Recall that, in gradual resizing, every insert is followed by a rehash that moves one element from the current to the new hash table.

The average number of insertion attempts with elastic cuckoo hashing is the *Elastic Total* curve. This curve is the addition of the *Elastic Insert* and the *Elastic Rehash* curves. The latter are the rehashes of elements from the old to the new table during resizing. In more detail, assume that one element is inserted during resizing. This insertion plus any additional re-insertions due to collisions are counted in *Elastic Insert*. The associated rehash of one element from the old to the new hash table plus any additional re-insertions due to collisions are counted in *Elastic Rehash*.

We see that, at an occupancy equal to r_t, *Elastic Rehash* jumps off. As resizing proceeds and elements start spreading over the larger new table, the insertion attempts due to rehashes decrease.

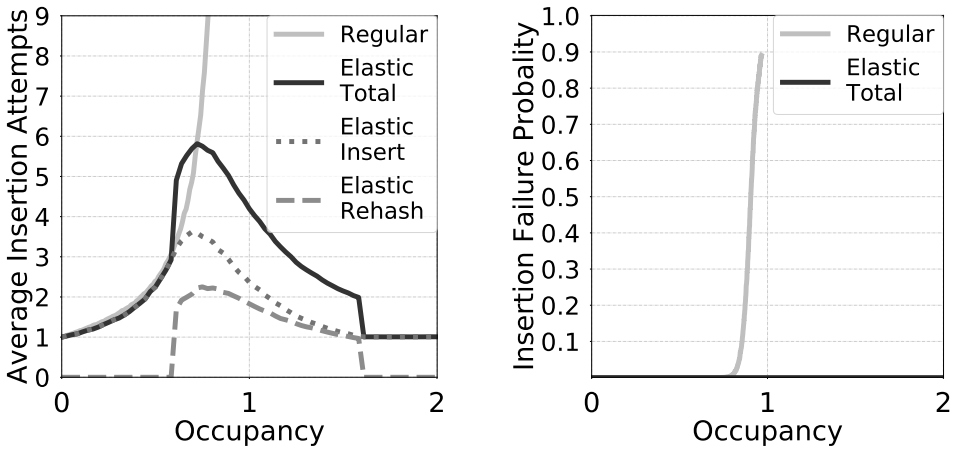


Figure 5.12: Elastic cuckoo hashing characterization.

When the resize terminates at an occupancy of ≈1.6, the *Elastic Rehash* curve falls to zero. The *Elastic Insert* curve gradually decreases as resizing proceeds. As a result, *Elastic Total* has a peak at the beginning of the resize but then goes down to 1 after the resize completes. Another peak would occur later, when the new table is resized.

The figure also shows the attempts with *Regular* cuckoo hashing without resizing. This curve is taken from Figure 5.3. Overall, elastic cuckoo hashing with resizing never reaches a large number of insertion attempts. Further, as shown in the rightmost figure, no insertion failure occurs.

5.8.2 Elastic Cuckoo Page Table Performance

Figure 5.13 evaluates the performance impact of elastic cuckoo page tables. Figure 5.13a shows the speedup of the applications running on *Baseline THP*, *Cuckoo 4KB*, and *Cuckoo THP* over running on *Baseline 4KB*.

The figure shows that, with 4KB pages only, using elastic cuckoo page tables (*Cuckoo 4KB*) results in an application speedup of 3–28% over using conventional radix page tables (*Baseline 4KB*). The mean speedup is 11%. When Transparent Huge Pages (THP) are enabled, the speedups with either radix page tables or elastic cuckoo page tables improve substantially, due to reduced TLB misses. The speedup of *Cuckoo THP* over *Baseline THP* is 3–18%. The mean is 10%. These are substantial application speedups.

In fact, *Cuckoo 4KB* outperforms not only *Baseline 4KB* but also *Baseline THP* in several applications. Some applications such as SSSP and TC do not benefit much from THP. However, *Cuckoo 4KB* also outperforms *Baseline THP* in applications such as MUMmer that leverage 2MB pages extensively.

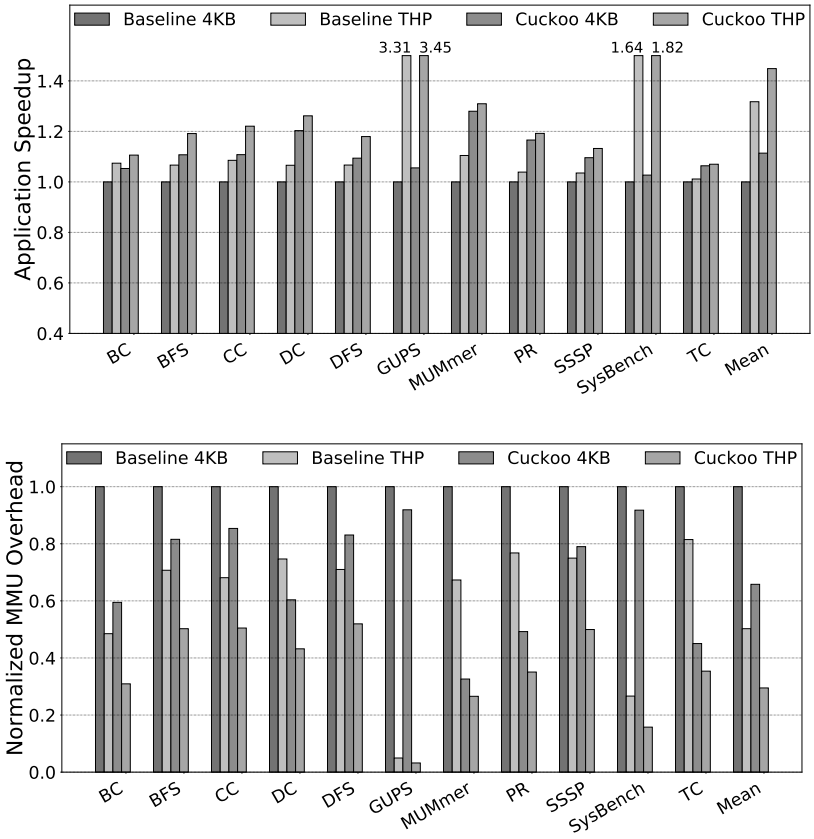


Figure 5.13: Performance impact of elastic cuckoo page tables: (a) application speedup and (b) MMU overhead reduction.

The performance gains attained by elastic cuckoo page tables come from several reasons. First and foremost, a page walk in elastic cuckoo page tables directly fetches the final translation, rather than having to also fetch intermediate levels of translation sequentially as in radix page tables. This ability speeds-up the translation. Second, performance is improved by the high hit rates of CWCs, which are due to the facts that CWCs do not have to store entries with intermediate levels of translation and that each CWC entry is small — again unlike radix page tables. Finally, we observe that when the page walker performs Size and Partial Walks, it brings translations into the L2 and L3 caches that, while not loaded into the TLB, will be used in the future. In effect, the walker is prefetching translations into the caches.

Figure 5.13b shows the time spent by all the memory system requests in the MMU, accessing the TLBs and performing the page walk. The time is shown normalized to the time under *Baseline 4KB*. This figure largely follows the trends in Figure 5.13a. On average, *Cuckoo 4KB* reduces the MMU overhead of *Baseline 4KB* by 34%, while *Cuckoo THP*'s overhead is 41% lower than that of *Baseline THP*. The figure also shows that applications like GUPS and SysBench, which perform

fully randomized memory accesses, benefit a lot from THPs.

We have also evaluated applications that have a lower page walk overhead, especially with THP, such as MCF and Cactus from SPEC2006 [263], Streamcluster from PARSEC [264], and XSBench [265]. Their memory footprints are 1.7GB, 4.2GB, 9.1GB, and 64GB, respectively. In these applications, while elastic cuckoo page tables reduce the MMU overhead compared to radix page tables, address translation is not a bottleneck. Hence, application performance remains the same.

5.8.3 Elastic Cuckoo Page Table Characterization

MMU and Cache Subsystem

Figure 5.14 characterizes the MMU and cache subsystem for our four configurations. From top to bottom, it shows the number of MMU requests Per Kilo Instruction (PKI), the L2 cache Misses Per Kilo Instruction (MPKI), and the L3 MPKI. In each chart, the bars are normalized to *Baseline 4KB*.

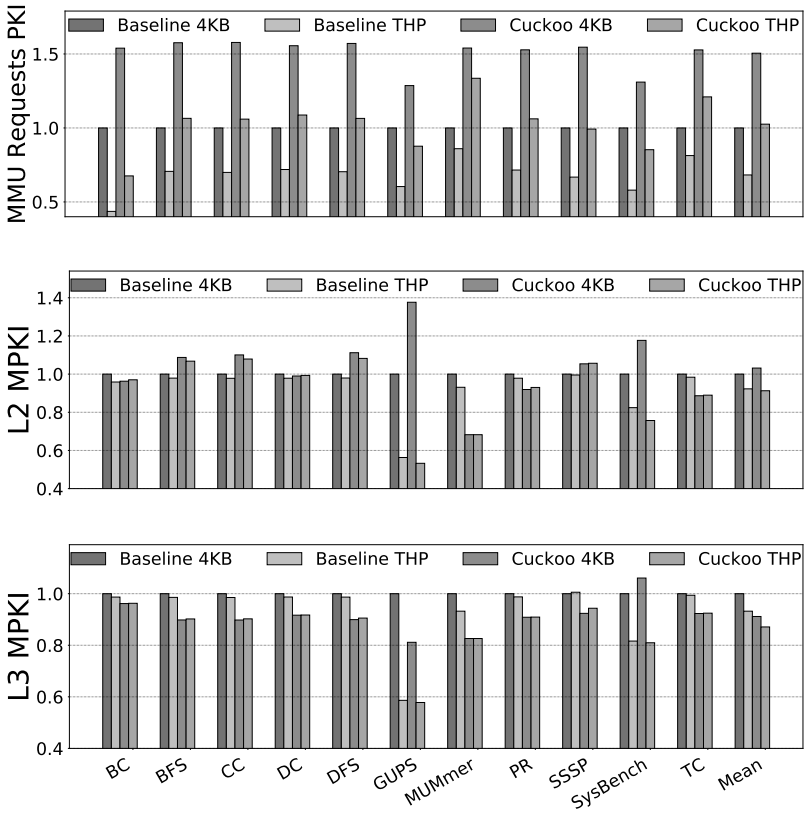


Figure 5.14: Characterizing the MMU and cache subsystem.

MMU requests are the requests that the MMU issues to the cache hierarchy on a TLB miss. For the baseline systems, they are memory requests to obtain page translations for the four radix tree levels; for the cuckoo systems, they are requests for page translations and for CWT entries. From the figure, we see that *Cuckoo 4KB* and *Cuckoo THP* issue many more requests than *Baseline 4KB* and *Baseline THP*, respectively. There are two reasons for this. The first and most important one is that many of the walks of the cuckoo page walker are not Direct Walks. Therefore, they request more page table entries than required — in fact no access to the PTE elastic cuckoo page table can use a Direct Walk because we have no PTE-CWT table. The second reason is the accesses to the CWTs themselves.

Fortunately, most of these additional MMU accesses are intercepted by the L2 and L3 caches and do not reach main memory. Indeed, as we show in the central and bottom chart of Figure 5.14, the L2 MPKI of the Cuckoo systems is similar to that of the Baseline systems, and the L3 MPKI of the Cuckoo systems is lower than that of the Baseline systems. The reason is two fold. First, the CWT entries are designed to cover large sections of memory. Hence, the needed entries fit in a few cache lines, leading to high cache hit rates. The second reason is that the additional elastic cuckoo page table entries that are brought in by accessing multiple ways of a table are typically later reused by another access. Accessing them now prefetches them for a future access. In summary, despite elastic cuckoo page tables issuing more MMU requests, most of the requests hit in the caches and the overall traffic to main memory is lower than with radix page tables.

To illustrate this point, Figure 5.15 considers all the MMU accesses in the MUMmer application and groups them in bins based on the time they take to complete. The figure shows data for *Baseline THP* and *Cuckoo THP*. On top of the bars, we indicate the likely layer of the memory hierarchy accessed for certain ranges of latencies: cache hit, 1st DRAM access, 2nd DRAM access, and 3rd DRAM access. *Cuckoo THP* never performs more than one DRAM access. From the figure, we see that *Cuckoo THP* MMU accesses have much lower latencies; most are intercepted by the caches or at worst take around 200 cycles. *Baseline THP* MMU accesses often perform two or three DRAM accesses, and have a long latency tail that reaches over 500 cycles.

Types of Walks

We now consider the elastic cuckoo page translation process of Figure 5.11, and measure the relative frequency of each type of cuckoo walk shown in Figure 5.9. We use *Cuckoo THP* with 2MB huge pages. As a reference, the average hit rates of PUD-CWC and PMD-CWC across all our applications are 99.9% and 87.7%, respectively.

Figure 5.16 shows the distribution of the four types of cuckoo walks for each application. Starting from the bottom of the bars, we see that the fraction of *Complete Walks* is negligible. A com-

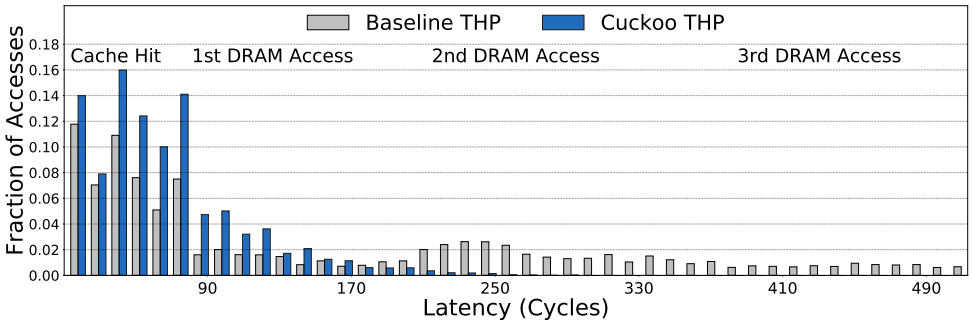


Figure 5.15: Histogram of MMU accesses in MUMmer.

plete walk only happens when both PUD-CWC and PMD-CWC miss, which is very rare. *Partial Walks* occur when the memory section accessed has both huge and regular pages. Situations where both page sizes are interleaved in virtual memory within one section are infrequent. They occur to some extent in BC, and rarely in other applications.

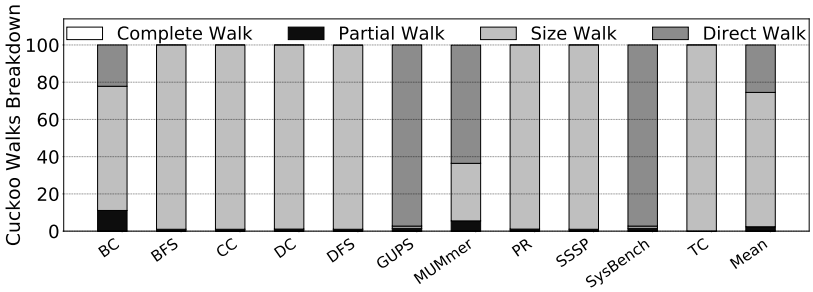


Figure 5.16: Distribution of the types of cuckoo walks.

Fortunately, most walks are of the cheap kinds, namely *Size Walks* and *Direct Walks*. *Size Walks* occur when the memory segment only contains regular pages. Hence, we observe this behavior in applications that do not take advantage of huge pages, such as BFS, CC, DC, DFS, PR, SSSP, TC and, to a lesser extent, BC. These walks are the most common ones. They also occur when the region only has huge pages but the PMD-CWC misses — an infrequent case.

Finally, *Direct Walks* occur when the memory segment only contains huge pages. We observe them in applications that use huge pages extensively, like GUPS, SysBench, and MUMmer. Overall, cheap walks are the most common.

Memory Consumption

Figure 5.17 shows the memory consumption of the page tables in the different applications for various configurations. For the Cuckoo systems, the bars also include the memory consumption

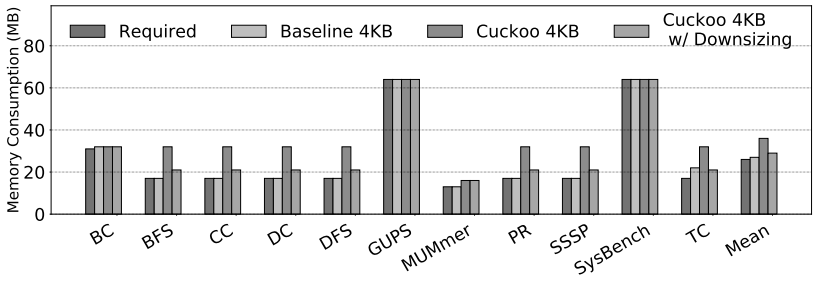


Figure 5.17: Memory consumption of page tables.

of CWTs. The first bar (*Required*) is the number of PTEs used by the application times 8 bytes. The figure then shows *Baseline 4KB*, *Cuckoo 4KB*, and *Cuckoo 4KB* with table downsizing. Note that we have not used downsizing anywhere else in this chapter. On average, *Required* uses 26MB. *Baseline 4KB* consumes on average 27MB, as it also needs to keep the three upper levels of translation. *Cuckoo 4KB* consumes on average 36MB, due to allocating hash tables with a power-of-two number of entries. The CWTs add very little space. Overall, the absolute increase in memory consumption needed to support *Cuckoo 4KB* is tiny, compared to the total memory capacity of modern systems. Finally, *Cuckoo 4KB* with table downsizing consumes on average 29MB.

Comparison to 2-ary Cuckoo Page Tables

We repeat our evaluation of elastic cuckoo page tables using 2-ary elastic cuckoo hashing for the page tables rather than our default 3-ary design. Our results show that using 3-ary structures speeds-up applications over using 2-ary structures: the applications run on average 6.7% faster in *Cuckoo 4KB* and 3.3% faster in *Cuckoo THP*. The reason is that 2-ary structures are liable to hash collisions at lower occupancies and, therefore, need to be resized sooner. We find that, over the course of an application, 2-ary tables require more rehashing operations than 3-ary tables: on average across our applications, they require 63.3% more rehashes in *Cuckoo 4KB* and 84% in *Cuckoo THP*. The additional accesses issued by the MMUs with 3-ary structures have relatively little performance impact because they typically hit in the caches.

5.9 OTHER RELATED WORK

To reduce TLB misses, recent studies have proposed to optimize TLB organizations by clustering, coalescing, contiguity [232, 234, 235, 238, 266, 267, 268, 269, 270], prefetching [231, 271, 272], speculative TLBs [237], and large part-of-memory TLBs [273, 274]. To increase TLB reach, support for huge pages has been extensively studied [232, 275, 276, 277, 278, 279, 280, 281, 282,

283, 284, 285, 286], with OS-level improvements [275, 276, 277, 284]. Other works propose direct segments [239, 287] and devirtualized memory [288], and suggest that applications manage virtual memory [289].

Many of these advances focus on creating translation contiguity: large contiguous virtual space that maps to large contiguous physical space. In this way, fewer translations are required, TLB misses are reduced, and costly multi-step page walks are minimized. Unfortunately, enforcing contiguity hurts the mapping flexibility that the kernel and other software enjoy. Further, enforcing contiguity is often impossible — or counterproductive performance-wise. Instead, in our work, we focus on dramatically reducing the cost of page walks by creating a single-step translation process, while maintaining the existing abstraction for the kernel and other software. As a result, we do not require contiguity and retain all the mapping flexibility of current systems.

5.10 CONCLUSION

This chapter presented Elastic Cuckoo Page Tables, a novel page table design that transforms the sequential address translation walk of radix page tables into fully parallel look-ups, harvesting for the first time the benefits of memory-level parallelism for address translation. Our evaluation showed that elastic cuckoo page tables reduce the address translation overhead by an average of 41% over conventional radix page tables, and speed-up application execution by 3–18%. Our current work involves exploring elastic cuckoo page tables for virtualized environments.

Chapter 6: Fusing Address Translations for Containers

6.1 INTRODUCTION

Cloud computing is ubiquitous. Thanks to its ability to provide scalable, pay-as-you-go computing, many companies choose cloud services instead of using private infrastructure. In cloud computing, a fundamental technology is virtualization. Virtual Machines (VMs) allow users to share resources, while providing an isolated environment to each user.

Recently, cloud computing has been undergoing a radical transformation with the emergence of *Containers*. Like a VM, a container packages an application and all of its dependencies, libraries, and configurations, and isolates it from the system it runs on. However, while each VM requires a guest OS, multiple containers share a single kernel. As a result, containers require significantly fewer memory resources and have lower overheads than VMs. For these reasons, cloud providers such as Google's Compute Engine [290], Amazon's ECS [291], IBM's Cloud [292], and Microsoft's Azure [293] now provide container-based solutions.

Container environments are typically oversubscribed, with many more containers running than cores [294]. Moreover, container technology has laid the foundation for *Serverless* computing [295], a new cloud computing paradigm provided by services like Amazon's Lambda [296], Microsoft's Azure Functions [297], Google's Cloud Functions [298], and IBM's Cloud Functions [299]. The most popular use of serverless computing is known as Function-as-a-Service (FaaS). In this environment, the user runs small code snippets called functions, which are triggered by specified events. The cloud provider automatically scales the number and type of functions executed based on demand, and users are charged only for the amount of time a function spends computing [300, 301].

Our detailed analysis of containerized environments reveals that, very often, the same Virtual Page Number (VPN) to Physical Page Number (PPN) translations, with the same permission bit values, are extensively replicated in the TLB and in page tables. One reason for this is that containerized applications are encouraged to create many containers, as doing so simplifies scale-out management, load balancing, and reliability [302, 303]. In such environments, applications scale with additional containers, which run the same application on different sections of a common data set. While each container serves different requests and accesses different data, a large number of the pages accessed is the same across containers.

Another reason for the replication is that containers are created with forks, which replicate translations. Further, since containers are stateless, data is usually accessed through the mounting of directories and the memory mapping of files. The result is that container instances of the same

application share most of the application code and data pages. Also, both within and across applications, containers often share middleware. Finally, the lightweight nature of containers encourages cloud providers to deploy many containers in a single host [304]. All this leads to numerous replicated page translations.

Unfortunately, state-of-the-art TLB and page table hardware and software are designed for an environment with few and diverse application processes. This has resulted in per-process tagged TLB entries, separate per-process page tables, and lazy page table management — where, rather than updating the page translations at process creation time, they are updated later on demand. In containerized environments, this approach causes high TLB pressure, redundant kernel work during page table management and, generally, substantial overheads.

To remedy this problem, we propose *BabelFish*, a novel architecture to *share translations across containers* in the TLB and page tables — without sacrificing the isolation provided by the virtual memory abstractions. BabelFish eliminates the replication of translations in two ways. First, it modifies the TLB to dynamically share identical {VPN, PPN} pairs and permission bits across containers. Second, it merges page table entries of different processes with the same {VPN, PPN} translations and permission bits. As a result, BabelFish reduces the pressure on the TLB, reduces the cache space taken by translations, and eliminates redundant minor page faults. In addition, it effectively prefetches shared translations into the TLB and caches. The end result is higher performance of containerized applications and functions, and faster container bring-up.

We evaluate BabelFish with simulations of an 8-core processor running a set of Docker containers in an environment with conservative container co-location. On average, under BabelFish, 53% of the translations in containerized workloads and 93% of the translations in FaaS workloads are shared. As a result, BabelFish reduces the mean and tail latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowers the execution time of containerized compute workloads by 11%. Finally, it reduces FaaS function bring-up time by 8% and execution time by 10%–55%.

6.2 BACKGROUND

6.2.1 Containers

Containers are a lightweight software virtualization solution that aims to ease the deployment of applications [305]. A container is defined by an image that specifies all of the requirements of an application, including the application binary, libraries, and kernel packages required for deployment. The container environment is more light-weight than a traditional VM, as it eliminates

the guest operating system. All of the containers share the same kernel and, consequently, many pages can be automatically shared among containers. As a result, containers typically exhibit better performance and consume less memory than VMs [306]. Docker containers [307] is the most prominent container solution. In addition, there are management frameworks, such as Google's Kubernetes [308] and Docker's Swarm [309], which automate the deployment, scaling, and maintenance of containerized applications.

The lightweight nature of containers has led to the Function-as-a-Service (FaaS) paradigm [296, 297, 298, 299, 310]. In FaaS, the user provides small code snippets, called *Functions*, and providers charge users only for the amount of time a function spends computing. In FaaS environments, many functions can be running concurrently, which leads to high consolidation rates.

Containers rely on OS virtualization and, hence, use the process abstraction, rather than the thread one, to provide resource isolation and usage limits. Typically, a containerized application scales out by creating multiple replicated containers, as this simplifies load balancing and reliability [302, 303]. The resulting containers usually include one process each [311], and run the same application but use different sections of data. For example, a graph application exploits parallelism by creating multiple containers, each one with one process. Each process performs different traversals on the shared graph. As another example, a data-serving workload such as Apache HTTPD, creates many containers, each one with one process. Each process serves a different incoming request. In both examples, the containers share many pages.

6.2.2 Address Translation in x86 Linux

Address translation is described in Section 2.2.1.

6.2.3 Replicated Translations

The Linux kernel avoids having multiple copies of the same physical page in memory. For example, when a library is shared among applications, only a single copy of the library is brought into physical memory. As a result, multiple processes may point to the same PPN. The VPNs of the different processes may be the same, and have identical permission bits. Furthermore, on a fork operation, pages are only copied lazily and, therefore, potentially many VPNs in the parent and child are the same and point to the same PPNs. Finally, file-backed mappings created through *mmap* lead to further sharing of physical pages. In all of these cases, there are multiple copies of the same $\{\text{VPN}, \text{PPN}\}$ translation with the same permission bits, in the TLB (tagged with different PCIDs) and across the page tables of different processes.

6.3 BABELFISH DESIGN

BabelFish has two parts. One enables TLB entry sharing, and the other page table entry sharing. In the following, we describe each part in turn, and then present a simple example.

6.3.1 Enabling TLB Entry Sharing

Current TLBs may contain multiple entries with the same {VPN, PPN} pair, the same permission bits, and tagged with different PCIDs. Such replication is common in containerized environments, and can lead to TLB thrashing. To solve this problem, BabelFish combines these entries into a single one with the use of a new identifier called *Container Context Identifier* (CCID). All of the containers created by a user for the same application are given the same CCID. It is expected that the processes within the same CCID group will want to share many TLB entries and page table entries.

BabelFish adds a CCID field to each entry in the TLB. Further, when the OS schedules a process, the OS loads the process' CCID into a register — like it currently does for the process' PCID. Later, when the TLB is accessed, the hardware will look for an entry with a matching VPN tag and a matching CCID. If such an entry is found, the translation succeeds, and the corresponding PPN is read. Figure 6.1 shows an example for a two-way set-associative TLB. This support allows all the processes in the same CCID group to share entries.

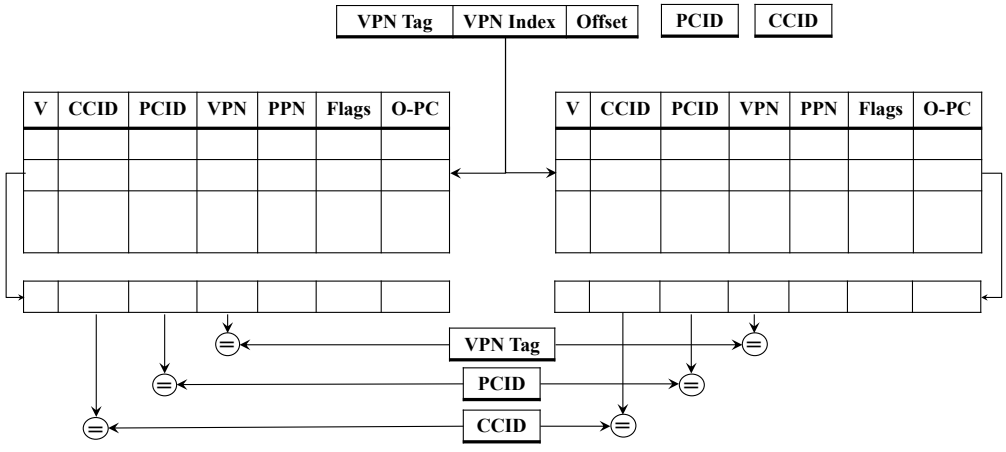


Figure 6.1: Two-way set-associative BabelFish TLB.

The processes of a CCID group may not want to share some pages. In this case, a given VPN should translate to different PPNs for different processes. To support this case, we retain the PCID in the TLB, and add an *Ownership* (O) bit in the TLB. If O is set, it indicates that this page is owned rather than shared, and a TLB hit *also* requires a PCID match.

We also want to support the more advanced case where many of the processes of the CCID group want to share the same $\{VPN_0, PPN_0\}$ translation, but a few other processes do not, and have made their own private copies. For example, one process created $\{VPN_0, PPN_1\}$ and another one created $\{VPN_0, PPN_2\}$. This situation occurs when a few of the processes in the CCID group have written to a Copy-on-Write (CoW) page and have their own private copy of the page, while most of the other processes still share the original clean page. To support this case, we integrate the Ownership bit into a new TLB field called *Ownership-PrivateCopy* (O-PC) (Figure 6.1).

Ownership-PrivateCopy Field.

The O-PC field is expanded in Figure 6.2. It contains a 32-bit *PrivateCopy (PC)* bitmask, one bit that is the logic OR of all the bits in the PC bitmask (OR_{PC}), and the Ownership (O) bit. The PC bitmask has a bit set for each process in the CCID group that has its own private copy of this page. The rest of the processes in the CCID group, *which can be an unlimited number*, still share the clean shared page. We limit the number of private copies to 32 to keep the storage modest. Before we describe how BabelFish assigns bits to processes, we describe how the complete TLB translation in BabelFish works.

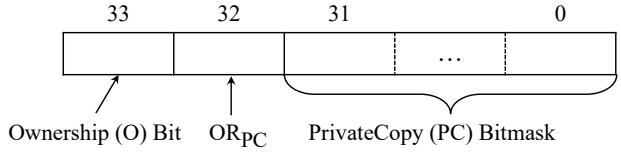


Figure 6.2: Ownership-PrivateCopy (O-PC) field. The PrivateCopy (PC) bitmask has a bit set for each process in the CCID group that has its own private copy of the page. The OR_{PC} bit is the logic OR of all the bits in the PC bitmask.

The BabelFish TLB is indexed as a regular TLB, using the VPN Tag. The hardware looks for a match in the VPN and in the CCID. All of the potentially-matching TLB entries will be in the same TLB set, and more than one match may occur. On a match, the O-PC and PCID fields are checked, and two cases are possible. First, if the O bit is set, this is a private entry. Hence, the entry can be used only if the process' PCID matches the TLB entry's PCID field.

Alternately, if O is clear, this is a shared entry. In this case, before the process can use it, the process needs to check whether the process itself has its own private copy of the page. To do so, the process checks its own bit in the PC bitmask. If the bit is set, the process cannot use this translation because the process already has its own private copy of the page. An entry for such page may or may not exist in the TLB. Otherwise, since the process' bit in the PC bitmask is clear, the process can use this translation.

The O-PC information of a page is part of a TLB entry, but only the O and OR_{PC} bits are stored in the page table entry. The PC bitmask is *not* stored in the page table entry to avoid changing the data layout of the page tables. Instead, it is stored in an OS software structure called the *MaskPage* that is described in Section 6.4.2. Each MaskPage also includes an ordered list (*pid_list*) of up to 32 pids of processes from the CCID group. The order of the pids in this list encodes the mapping of PC bitmask bits to processes. For example, the second pid in the pid_list is the process that is assigned the second bit in the PC bitmask.

In BabelFish, a MaskPage contains the PC bitmasks and pid_list for all the pages of a CCID group mapped by a set of PMD tables. The details are explained in Section 6.4.2.

Actions on a Write to a Copy-on-Write (CoW) Page.

To understand the operation of the pid_list, consider what happens when a process writes to a CoW page. The OS checks whether the process is already in the pid_list in the MaskPage for this PMD table set. If it is not, this is the process' first CoW event in this MaskPage. In this case, the OS performs a set of actions. Specifically, it adds the process' pid to the end of the pid_list in the MaskPage, effectively assigning the next bit in the corresponding PC bitmask to the process. This assignment will be used by the process in the future, to know which bit in the PC bitmask to check when it accesses the TLB. In addition, the OS sets that PC bitmask bit in the MaskPage to 1. Then, the OS makes a copy of a page of 512 *pte_t* translations for the process, sets the Ownership (O) bit for each translation, allocates a physical page for the single page updated by the process, and changes the translation for that single page to point to the allocated physical page. The other 511 pages will be allocated later on demand as needed. We choose to copy a page of 512 translations rather than only one translation to reduce the bookkeeping overhead.

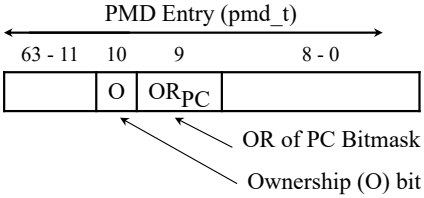
In addition, irrespective of whether this was the process' first CoW event in this MaskPage, the OS has to ensure that the TLB is consistent. Hence, similar to a conventional CoW, the OS invalidates from the local and remote TLBs, the TLB entry for this VPN *that has the O bit equal to zero*. The reason is that this entry has a stale PC bitmask. Note that only this single entry needs to be invalidated, while the remaining (up to 511) translations in the same PTE table can still safely remain in the TLBs. Finally, when the OS gives control back to the writing process, the latter will re-issue the request, which will miss in the TLB and bring its new *pte_t* entry into the TLB, with the O bit set and the updated PC bitmask.

Writable pages (e.g., data set) and read-only pages (e.g., library code) can have an unlimited number of sharers. However, CoW pages, which can be read-shared by an unlimited number of sharers, cannot have more than 32 unique writing processes — since the PC bitmask runs out of space. We discuss the case when the number of writers goes past 32 in Section 6.4.2.

Overall, with this support, BabelFish allows multiple processes in the same CCID group to share the TLB entry for a page — even after other processes in the group have created their own private copies of the page. This capability reduces TLB pressure. This mechanism works for both regular pages and huge pages.

Role of the OR_{PC} Bit in the O-PC Field.

Checking the PC bitmask bits on a TLB hit adds overhead. The same is true for loading the PC bitmask bits into the TLB on a TLB miss. To reduce these overheads, BabelFish uses the OR_{PC} bit (Figure 6.2), which is the logic OR of all the PC bitmask bits. This bit is present in the O-PC field of a TLB entry. It is also present, together with the O bit, in each *pmd_t* entry of the PMD table. Specifically, bits O and OR_{PC} use the currently-unused bits 10 and 9 of *pmd_t* in the x86 Linux implementation [41, 257] (Figure 6.3(a)).



(a)

O bit	ORPC bit	Check PC bitmask on TLB hit?	Fetch PC bitmask on TLB miss?
0	0	No	No
0	1	Yes	Yes
1	X	No	No

(b)

Figure 6.3: OR_{PC} bit: position in the *pmd_t* (a) and impact (b).

The OR_{PC} bit is used to selectively avoid reading the PC bitmask on a TLB hit, and to avoid loading the PC bitmask to the TLB on a TLB miss. The logic is shown in Figure 6.3(b). Specifically, consider the case when the O bit is clear. Then, if the OR_{PC} bit is clear, the two operations above can be safely skipped; if OR_{PC} is set, they need to be performed. Consider now the case when the O bit is set. In this case, the two operations can also be skipped. The reason is that an access to a TLB entry with the O bit set relies only on the PCID field to decide on whether there is a match. In all cases, when the PC bitmask bits are not loaded into the TLB, the hardware clears the corresponding TLB storage. Overall, with OR_{PC}, the two operations are skipped most of the time.

Rationale for Supporting CoW Sharing

Supporting CoW sharing within a CCID group, where a page is read-shared by a potentially unlimited number of processes, and a few other processes have private copies of the page adds complexity to the design. However, it is important to support this feature because it assists in accelerating container bring-up — and fast bring-up is a critical requirement for FaaS environments.

Specifically, during bring-up, containers first read several pages shared by other containers. Then, they write to some of them. This process occurs gradually. At any point, there are some containers in the group that share the page read-only, and others that have created their own copy. Different containers may end-up writing different sets of pages. Hence, not all containers end-up with a private copy of the page.

6.3.2 Enabling Page Table Entry Sharing

In current systems, two processes that have the same {VPN, PPN} mapping and permission bits still need to keep separate page table entries. This situation is common in containerized environments, where the processes in a CCID group may share many pages (e.g., a large library) using the same {VPN, PPN} mappings. Keeping separate page table entries has two costs. First, the many *pte_t* requested from memory could thrash the cache hierarchy [312]. Second, every single process in the group that accesses the page may suffer a minor page fault, rather than only one process suffering a fault. Page fault latency has been shown to add significant overhead [313].

To solve this problem, BabelFish changes the page table structures so that processes with the same CCID can share one or more levels of the page tables. In the most common case, multiple processes will share the table in the last level of the translation. This is shown in Figure 6.4. The figure shows the translation of an address for two processes in the same CCID group that map it to the same physical address. The two processes (one with $CR3_0$ and the other with $CR3_1$) use the same last level page (PTE). They place in the corresponding entries of their previous tables (PMD) the base address of the same PTE table. Now, both processes together suffer only one minor page fault (rather than two), and reuse the cache line that contains the target *pte_t*.

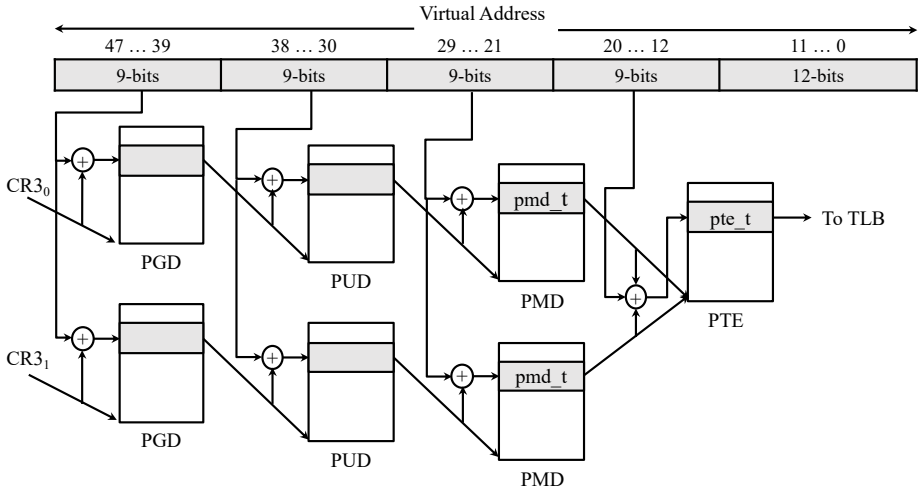


Figure 6.4: Page table sharing in BabelFish.

The default sharing level in BabelFish is a PTE table, which maps 512 4KB pages in x86-64. Sharing can also occur at other levels. For example, it can occur at the PMD level—i.e., entries in multiple PUD tables point to the base of the same PMD table. In this case, multiple processes can share the mapping of 512×512 4KB pages or 512 2MB huge pages. Further, processes can share a PUD table, in which case they can share even more mappings. We always keep the first level of the tables (PGD) private to the process.

Examples of large chunks of shared pages are libraries, and data accessed through mounting directories or memory mapping of files. Note that Figure 6.4 does not imply that all the pages in the shared region are present in memory at the time of sharing; some may be missing. However, it is not possible for two processes to share a table and want to keep private some of pages mapped by the table.

6.3.3 Example of BabelFish Operation

To understand the impact of BabelFish, we describe an example. Consider three containers (*A*, *B*, and *C*) that have the same $\{VPN_0, PPN_0\}$ translation. First, *A* runs on Core *0*, then *B* runs on Core *1*, and then *C* runs on Core *0*. Figure 6.5 shows the timeline of the translation process, as each container, in order, accesses VPN_0 for the first time. The top three rows of the figure correspond to a conventional architecture, and the lower three to BabelFish. To save space, we show the timelines of the three containers on top of each other; in reality, they take place in sequence.

We assume that PPN_0 is in memory but not yet marked as present in memory in any of the *A*, *B*, or *C* *pte_t*s. We also assume that none of these translations is currently cached in the page walk cache (PWC) of any core.

Conventional Architecture. The top three rows of Figure 6.5 show the conventional process. As container *A* accesses VPN_0, the translation misses in the L1 and L2 TLBs, and in the PWC. Then, the page walk requires a memory access for each level of the page table (we assume that, once the PWC has missed, it will not be accessed again in this page walk). First, as the entry in the PGD is accessed, the page walker issues a cache hierarchy request. The request misses in the L2 and L3 caches and hits in main memory. The location is read from memory. Then, the entry in the PUD is accessed. The process repeats for every level, until the entry in the PTE is accessed. Since we assume that PPN_0 is in memory but not marked as present, *A* suffers a minor page fault as it completes the translation (Figure 6.5). Finally, *A*'s page table is updated and a $\{VPN_0, PPN_0\}$ translation is loaded into the TLB.

After that, container *B* running on another core accesses VPN_0. The hardware and OS follow exactly the same process as for *A*. At the end, *B*'s page table is updated and a $\{VPN_0, PPN_0\}$

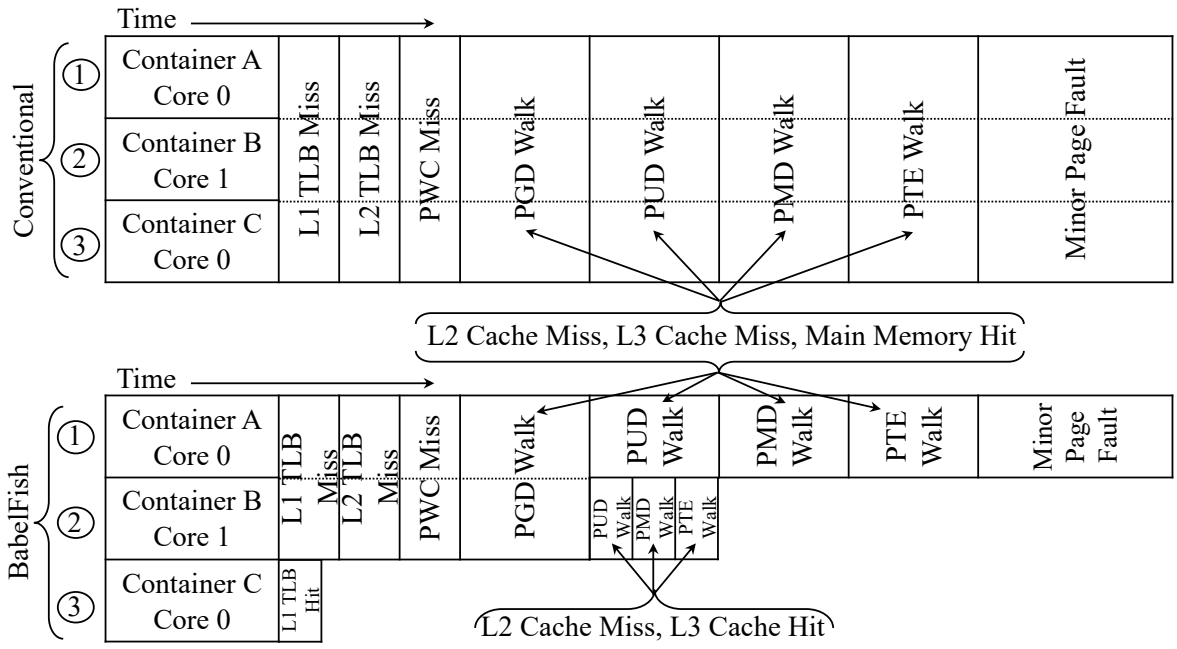


Figure 6.5: Timeline of the translation process in a conventional (top) and BabelFish (bottom) architecture. In the figure, container *A* runs on Core *0*, then container *B* on Core *1*, and then container *C* on Core *0*.

translation is loaded into the TLB.

Finally, container *C* running on the same core as *A* accesses VPN_0. Again, the hardware and OS follow exactly the same process. *C*'s page table is updated, and another $\{VPN_0, PPN_0\}$ translation is loaded into the TLB. The system does not take advantage of the state that *A* loaded into the TLB, PWC, or caches because the state was for a different process.

BabelFish Architecture. The lower three rows of Figure 6.5 show the behavior of BabelFish. Container *A*'s access follows the same translation steps as in the conventional architecture. After that, container *B* running on another core is able to perform the translation substantially faster. Specifically, its access still misses in the TLBs and in the PWC; this is because these are per-core structures. However, during the page walk, the multiple requests issued to the cache hierarchy miss in the local L2 but hit in the shared L3 (except for the PGD access). This is because BabelFish enables container *B* to reuse the page-table entries of container *A* — at any level except at the PGD level. Also, container *B* does not suffer any page fault.

Finally, as *C* runs on the same core as *A*, it performs a very fast translation. It hits in the TLB because it can reuse the TLB translation that container *A* brought into the TLB. Recall that, in the x86 architecture, writes to CR3 do not flush the TLB. This example highlights the benefits in a scenario where multiple containers are co-scheduled on the same physical core, either in SMT

mode, or due to an over-subscribed system.

6.4 IMPLEMENTATION

We now discuss the implementation of BabelFish.

6.4.1 Resolving a TLB Access

Figure 6.6 shows the algorithm that the hardware uses in a TLB access. For simplicity, the figure shows the flowchart assuming a single TLB level; in practice, the checks are performed sequentially in the L1 and L2 TLBs.

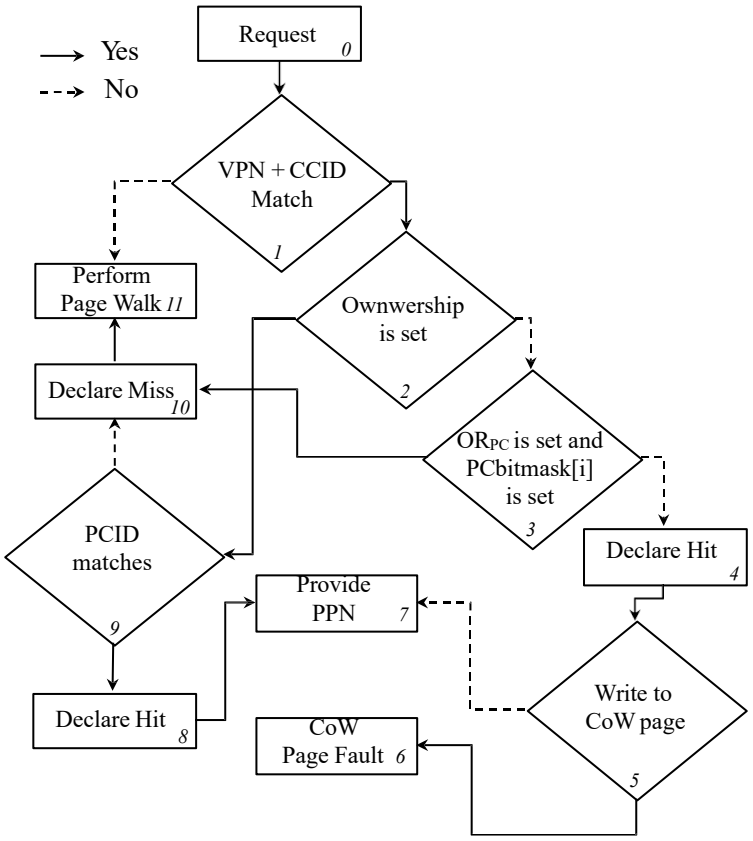


Figure 6.6: Flowchart of a TLB access.

As the TLB is accessed, each way of the TLB checks for a VPN and CCID match (①). If none of the ways matches, a page walk is initiated (⑪). Otherwise, for each of the matching ways, the following process occurs. The hardware first checks if the Ownership bit is set (②). If it is, the hardware checks for a PCID match (⑨). If the PCID matches (and assuming that all the

permissions checks pass) we declare a TLB hit (⑧) and provide the PPN (⑦). Otherwise, it is a TLB miss (⑩).

If the Ownership bit is clear, the hardware checks if the requesting process already has a private copy of the page. It does so by checking first the OR_{PC} bit and, if it is set, checking the bit of the process in the PC bitmask (③). If both are set, it means that the process has a private copy, and a miss is declared (⑩). Otherwise, a hit is declared (④). In this case, the hardware checks whether this is a write to a CoW page (⑤). If it is, a CoW page fault is declared (⑥). Otherwise, assuming that all the permissions checks pass, the PPN is provided (⑦). After all the TLB ways terminate their checks, if no hit has been declared (⑩), a page walk is initiated (⑪).

6.4.2 Storing and Accessing the PC Bitmask

To understand where the PC bitmask is stored and how it is accessed, consider Figure 6.7(a), which shows the page tables of three processes of a CCID group that share a PTE table. BabelFish adds a single *MaskPage* associated with the set of PMD tables of the processes.

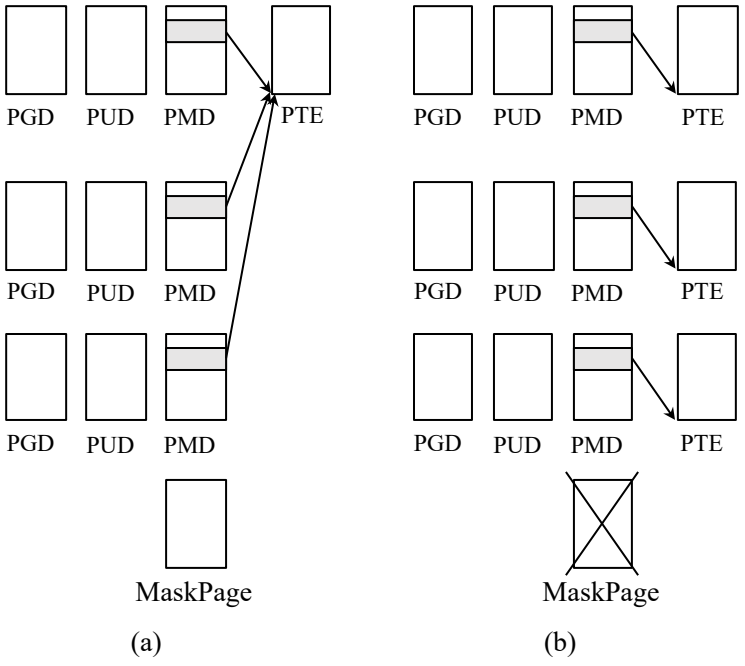


Figure 6.7: Operation of the MaskPage.

The OS populates the MaskPage with the PC bitmask and the pid_list information for all the pages mapped by the PMD table set. Figure 6.8 shows its contents. It contains up to 512 PC bitmasks for the 512 *pmd_t* entries in the PMD table set. Further, it contains a single pid_list, which has the ordered pids of the processes in the CCID group that have performed a CoW on any

of the pages mapped by the PMD table set. The pid_list has at most 32 pids. Hence, there can be at most 32 distinct processes performing CoW on these pages.

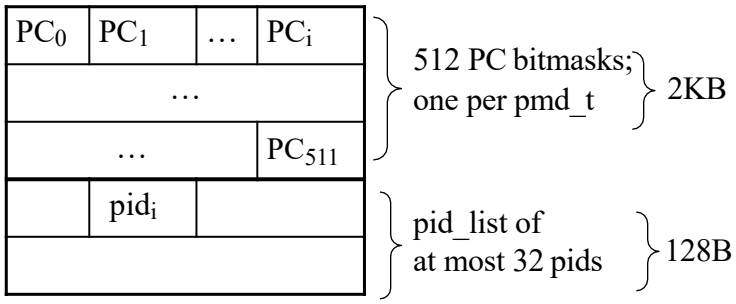


Figure 6.8: MaskPage with 512 PC bitmasks and one pid_list.

As a process performs a CoW on a page in this PMD table set for the first time, the OS puts its pid in the next position in the ordered pid_list. If this is position *i*, it means that the process claims bit *i* in all of the 512 PC bitmasks. Of course, bit *i* is set in only the PC bitmasks of the *pmd_t* entries that reach pages that the process has performed a CoW on.

With this design, on a TLB miss, as a *pmd_t* entry is accessed, the hardware checks the OR_{PC} bit (Section 6.3.1). If it is set, then the hardware accesses the MaskPage in parallel with the request for the *pte_t* entry. The hardware then reads the corresponding PC bitmask and loads it into the L1 TLB.

If more than 32 processes in a CCID group perform CoWs on pages mapped by a PMD table set, this design runs out of space, and all the processes in the group need to revert to non-shared translations — even if many processes in the group share many {VPN, PPN} mappings. Specifically, when a 33rd process performs a CoW, the OS allocates a page of *pte_t* translations for each of the processes in the group that were using shared translations in the PMD page set. In these new translations, the OS sets the Ownership (O) bit. The only physical data page that is allocated is the one updated by the writing process. The result is the organization shown in Figure 6.7(b), for a single PTE table being shared.

Consolidating CoW information from all the pages mapped by a PMD table set in a single MaskPage may sometimes be inefficient. However, we make this choice because selecting a finer granularity will increase space overhead. Furthermore, recall that writable pages (e.g., dataset) and read-only pages (e.g., code) can have an unlimited number of sharers. CoW pages can also be read-shared by an unlimited number of sharers; they just cannot have more than 32 writing processes. It can be shown that, with an extra indirection, one could support more writing processes.

6.4.3 Implementing Shared Page Table Entries

The page walker of current systems uses the contents of a control register to initiate the walk. In x86, it is the CR3 register. CR3 points to the beginning of the PGD table (Figure 6.4), and is unique per process. To minimize OS changes, BabelFish does not change CR3 and, therefore, does not support sharing PGD tables. This is not a practical limitation because processes rarely share the whole range of mapped pages.

BabelFish adds counters to record the number of processes currently sharing pages. One counter is assigned to each table at the translation level where sharing occurs. For example, if sharing occurs at the PTE level like in Figure 6.4, then there is one counter logically associated with each PTE table. When the last sharer of the table terminates or removes its pointer to the table, the counter reaches zero, and the OS can unmap the table. Such counters do not take much space or overhead to update, and are part of the virtual memory metadata.

6.4.4 Comparing BabelFish and Huge Pages

Both BabelFish and huge pages attempt to reduce TLB pressure, and they use orthogonal means. Huge pages merge the translations of multiple pages belonging to *the same* process to create a huge page; BabelFish merges the translations of *different* processes to eliminate unnecessary replication. As a result, BabelFish and huge pages are complementary techniques that can be used together.

If an application uses huge pages, BabelFish automatically tries to combine huge-page translations that have identical {VPN, PPN} pairs and permission bits. Specifically, if the application uses 2MB huge pages, BabelFish automatically tries to merge PMD tables; if the application uses 1GB huge pages, BabelFish automatically tries to merge PUD tables.

6.4.5 Supporting ASLR in BabelFish

Address Space Layout Randomization (ASLR) is a security mechanism that randomizes the positions of the segments of a process in virtual memory [314, 315, 316]. When a process is created, the kernel generates a random virtual address (VA) offset for each segment of the process, and adds it to the base address of the corresponding segment. Hence, the process obtains a unique segment layout, which remains fixed for the process lifetime. This strategy thwarts attackers that attempt to learn a program's segment layout. In Linux, a process has 7 segments, including code, data, stack, heap, and libraries.

BabelFish supports ASLR, even while sharing translations between processes. We envision two alternative configurations for ASLR, a software-only solution called *ASLR-SW* that requires

minimal OS changes, and a hardware-software solution called *ASLR-HW*, that provides stronger security guarantees.

In the *ASLR-SW* configuration, each CCID group has a private ASLR seed and, therefore, gets its own layout randomization. All processes in the same CCID group get the same layout and, therefore, can share TLB and page table entries among themselves. In all cases, different CCID groups have different ASLR seeds.

This configuration is easy to support in Linux. Specifically, the first container in a CCID gets the offsets for its segments, and subsequent containers in the group reuse them. This configuration is likely sufficient for most deployments, especially in serverless environments where groups of containers are spawned and destroyed quickly.

In the *ASLR-HW* configuration, each process has a private ASLR seed and, therefore, its own layout randomization. In this configuration, when a CCID group is created, the kernel generates a randomized VA offset for each of its segments, as indicated above. We call this set of offsets *CCID_offset[]*. Every time that a process *i* is spawned and joins the CCID group, in addition to getting its own set of random VA offsets for its segments (i_offset[]), it also stores the set of *differences* between the CCID group's offsets and its own offsets (i.e., diff_i_offset[] = CCID_offset[] - i_offset[]).

With this support, when a process is about to access the TLB, its VA goes through a logic module with comparators and one adder. This logic module determines which segment is being accessed, and then adds the corresponding entry in diff_i_offset[] to the VA being accessed. The result is the corresponding VA shared by the CCID group. The TLB is accessed with this address, enabling the sharing of translations between same-CCID processes while retaining per-process ASLR. Similarly, software page walks follow the same steps.

The hardware required by this configuration may affect the critical path of an L1 TLB access. Consequently, BabelFish places the logic module in between the L1 TLB and L2 TLB. The result is that BabelFish's translation sharing is only supported from the L2 TLB down; the L1 TLB does not support TLB entry sharing.

In practice, eliminating translation sharing from the L1 TLB only has a minor performance impact. The reason is that the vast majority of the translations are cached in the L2 TLB and, therefore, page walks are still eliminated. The L1 TLB performs well as long are there is locality of accesses within a process, which *ASLR-HW* does not alter.

To be conservative, in our evaluation of Section 6.7, we model BabelFish with *ASLR-HW* by default.

6.5 SECURITY CONSIDERATIONS

To minimize vulnerabilities, cloud providers limit page sharing to occur only within a single user security domain. For example, VMware only allows page deduplication within a single guest VM [317]. In Kubernetes, the security domain is called a Pod [318], and only containers or processes within a Pod share pages by default. In the recently-proposed X-Containers [319], the security domain is the shared LibOS, within which all processes share pages.

In this chapter, we consider a more conservative container environment, where a security domain contains only the containers of a single user that are running the same application. It is on top of this baseline environment that BabelFish proposes that the containers in the security domain additionally share address translations.

The baseline environment, where all the containers of a single user running the same application share pages is likely vulnerable to side channel attacks. Adding page translation sharing with BabelFish does not significantly change the security considerations over the baseline environment. This is because the attacker could leverage the sharing of pages to attack, without needing to leverage the sharing of translations. Addressing the issue of securing the baseline environment is beyond the scope of this chapter.

6.6 EVALUATION METHODOLOGY

Modeled Architecture. We use cycle-level simulations to model a server architecture with 8 cores and 32GB of main memory. The architecture parameters are shown in Table 6.1. Each core is out-of-order and has private L1 I+D caches, a private unified L2 cache, and a shared L3 cache. Each core has L1 I+D TLBs, a unified L2 TLB, and a page walk cache with a page walker. The table shows that the TLBs can hold pages of different sizes at the same time. With BabelFish, the access times of the L1 TLBs do not change. However, on an L1 TLB miss, BabelFish performs a two-cycle address transformation for ASLR (Section 6.4.5). Moreover, the L2 TLB has two access times: 10 and 12 cycles. The short one occurs when the O and $ORPC$ bits preempt an access to the PC bitmask (Figure 6.3(b)); the long one occurs when the PC bitmask is accessed. Section 6.7.4 provides details. We use Ubuntu Server 16.04 [320] and Docker 17.06 [321].

Modeling Infrastructure. We integrate the Simics [198] full-system simulator with the SST framework [199, 259] and the DRAMSim2 [200] memory simulator. Additionally, we utilize Intel SAE [201] on Simics for OS instrumentation. We use CACTI [100] for energy and access time evaluation. For the address translation, each hardware page walker is connected to the cache hierarchy and issues memory requests following the page walk semantics of x86-64 [41]. The Simics infrastructure provides the actual memory and control register contents for each memory

Processor Parameters	
Multicore chip	8 2-issue OoO cores, 128 ROB; 2GHz
L1 (D, I) cache	32KB, 8 way, WB, 2 cycle AT, 16 MSHRs, 64B line
L2 cache	256KB, 8 way, WB, 8 cycle AT, 16 MSHRs, 64B line
L3 cache	8MB, 16 way, WB, shared, 32 cycle AT, 128 MSHRs, 64B line
Per Core MMU Parameters	
L1 (D, I)TLB (4KB pages)	64 entries, 4 way, 1 cycle AT
L1 (D)TLB (2MB pages)	32 entries, 4 way, 1 cycle AT
L1 (D)TLB (1GB pages)	4 entries, FA, 1 cycle AT
ASLR Transformation	2 cycles on L1 TLB miss
L2 TLB (4KB pages)	1536 entries, 12 way, 10 or 12 cyc AT
L2 TLB (2MB pages)	1536 entries, 12 way, 10 or 12 cyc AT
L2 TLB (1GB pages)	16 entries, 4 way, 10 or 12 cycle AT
Page walk cache	16 entries/level, 4 way, 1 cycle AT
Main Memory Parameters	
Capacity; Channels	32GB; 2
Ranks/Channel; Banks/Rank	8; 8
Frequency; Data rate	1GHz; DDR
Host and Docker Parameters	
Scheduling quantum	10ms
PC bitmask; PCID; CCID	32 bits; 12 bits; 12 bits

Table 6.1: Architectural parameters. AT is Access Time.

access of the page walk. We use the page tables maintained by the Linux kernel during full-system simulation. To evaluate the hardware structures of BabelFish, we model them in detail in SST. To evaluate the software structures, we modify the Linux kernel and instrument the page fault handler.

Workloads. We use three types of workloads: three Data Serving applications, two Compute applications, and three Functions representing Function-as-a-Service (FaaS).

The *Data Serving* applications are the containerized *ArangoDB* [322], *MongoDB* [323], and *HTTPd* [324]. *ArangoDB* represents a key-value store NoSQL database with RocksDB as the storage engine. *MongoDB* is a scalable document-model NoSQL database with a memory mapped engine, useful as a backend for data analytics. *HTTPd* is an efficient open source HTTP server with multiprocess scaling, used for websites and online services. Each application is driven by the Yahoo Cloud Serving Benchmark [187] with a 500MB dataset.

The *Compute* applications are the containerized *GraphChi* [325] and *FIO* [326]. *GraphChi* is a graph processing framework with memory caching. We use the PageRank algorithm which traverses a 500MB graph from SNAP [327]. *FIO* is a flexible I/O benchmarking application that performs in-memory operations on a randomly generated 500MB dataset.

We developed three C/C++ containerized *Functions*: *Parse*, which parses an input string into tokens, a *Hash* function based on the djb2 algorithm[328], and a *Marshal* function that transforms an input string to an integer. All functions are based on OpenFaaS [310] and use the GCC image from Docker Hub [329]. Each function operates on an input dataset similar to [296]. For these functions, we explore dense and sparse inputs. In both cases, a function performs the same work; we only change the distance between one accessed element and the next. In dense, we access all the data in a page before moving to the next page; in sparse, we access about 10% of a page before moving to the next one.

Configurations Evaluated. We model widely-used container environments that exploit replication of the applications for better load balancing and fault tolerance. We conservatively keep the number of containers per core low. Specifically, in Data Serving and Compute workloads, each core is multiplexed between two containers, which run the same application on different input data. Each Data Serving container is driven by a distinct YCSB client and, therefore, serves different requests. Similarly, each Compute container accesses different random locations. As a result, in both types of workloads, each container accesses different data, but there is partial overlap in the data pages accessed by the two containers.

In the Function workloads, each core is multiplexed between three containers, each running a different function. The three containers access different data, but there is partial overlap in the data pages accessed by the three containers.

In each case, we compare two configurations: a conventional server (*Baseline*), and one augmented with the proposed hardware and software (*BabelFish*). We enable transparent huge pages (THP) for both the *Baseline* and *BabelFish*.

Simulation Methodology. BabelFish proposes software and hardware changes. To model a realistic system, we need to warm-up both the OS and the architecture state. We use two phases. In the first phase, we warm-up the OS by running the containers for Data Serving and Compute for a minute, and all the functions to completion, as they are short.

In the second phase, we bring the applications to steady state, warm-up the architectural state, and measure. Specifically, for Data Serving and Compute, we instrument the applications to track entry to steady state and then execute for 10 seconds. We then warm-up the architectural state by running 500 million instructions, and finally evaluate for four billion instructions. For Functions, there is no warm-up. We run all the three functions from the beginning to completion and measure their execution time.

We also measure container bring-up, as the time to start a Function container from a pre-created image (docker start). We perform full system simulation of all the above steps.

6.7 EVALUATION

The BabelFish performance improvements come from two sources: page table entry sharing and L2 TLB entry sharing. In this section, we first characterize these two sources, and then discuss the overall performance improvements. Finally, we examine the resources that BabelFish needs.

6.7.1 Characterizing Page Table Entry Sharing

Figure 6.9 shows the shareability of PTE entries (*pte_ts*) when running two containers of the same Data Serving and Compute application, or three containers of the functions. The figure includes the steady-state mappings of all pages, including container infrastructure, and program code and data. The data is obtained by native measurements on a server using Linux Pagemap [330], while running each application for 5 minutes.

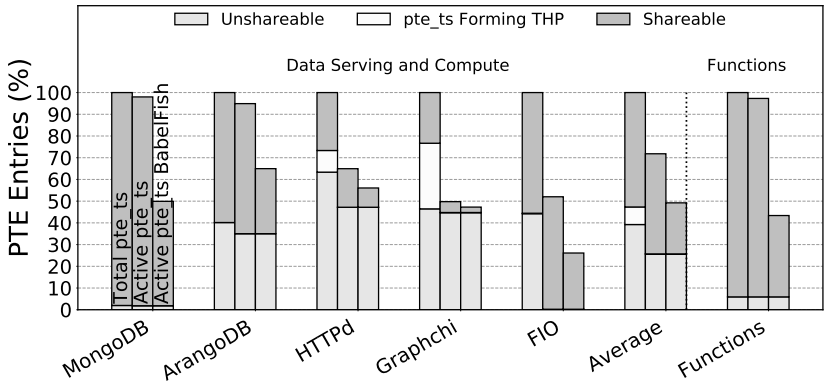


Figure 6.9: Page table sharing characterization.

For each application, there are three bars, which are normalized to the leftmost one. The leftmost one is the total number of *pte_ts* mapped by the containers. The central bar is the number of *Active pte_ts*, namely those that are placed by the kernel in the active LRU list. They are a proxy for the pages that are recently touched. The rightmost bar is the number of Active *pte_ts* after enabling BabelFish.

Each bar is broken down into *pte_ts* that are shareable, *pte_ts* that are unshareable, and *pte_ts* that correspond to the huge pages created by THP. The latter are also unshareable. A shareable *pte_ts* has an identical {VPN, PPN} pair and permission bits as another. Unshareable *pte_ts* are either exclusive to a process or not shareable.

Data Serving and Compute Applications. If we consider the average bars, we see that 53% of the total baseline *pte_ts* are shareable. Further, most of them are active. BabelFish reduces the number

of shareable active *pte_ts* by about half. Since this plot corresponds to only two containers, the reduction in shareable active *pte_ts* is at most half. Sharing *pte_ts* across more containers would linearly increase savings, as only one copy is required for all the sharers. Overall, the average reduction in total active *pte_ts* attained by BabelFish is 30%.

The variation of shareability across applications is primarily due to the differences in usage of shared data versus internal buffering. For example, GraphChi operates on shared vertices, but uses internal buffering for the edges. As a result, most of the active *pte_ts* are unshareable, and we see little gains. In contrast, MongoDB and FIO operate mostly on shared data and see substantial *pte_ts* savings. The other applications have a more balanced mixture of shareable and unshareable *pte_ts*.

Impact of Huge Pages

As shown in Figure 6.9, THP *pte_ts* are on average 8% of the total *pte_ts*, and are in the unshareable portion. Moreover, only a negligible number of these entries are active during execution. To understand this data, note that MongoDB and ArangoDB recommend disabling huge pages [331, 332]. Further, THP supports only anonymous mappings, and not file-backed memory mapping. The anonymous mappings are commonly used for internal buffering, which are unshareable. Therefore, huge pages are rarely active.

Functions

Functions have a very high percentage of shareable *pte_ts*. Combining them reduces the total active *pte_ts* by 57%. The unshareable *pte_ts* correspond to the function code and internal buffering, which are unique for each function. They account for only ≈6% of *pte_ts*.

One can subdivide the shareable category in the bars into application shared data and infrastructure pages. The latter contain the common libraries that are required by all the functions. It can be shown that they are 90% of the total shareable *pte_ts*, and can be shared across functions. Data *pte_ts* are few, but also shareable across functions.

6.7.2 Characterizing TLB Entry Sharing

Figure 6.10 shows the reduction in L2 TLB Misses Per Kilo Instructions (MPKI) attained by BabelFish. Note that in our evaluation, we conservatively do not share translations at the L1 TLB (Section 6.4.5). The figure is organized based on workload, and shows data and instruction entries separately.

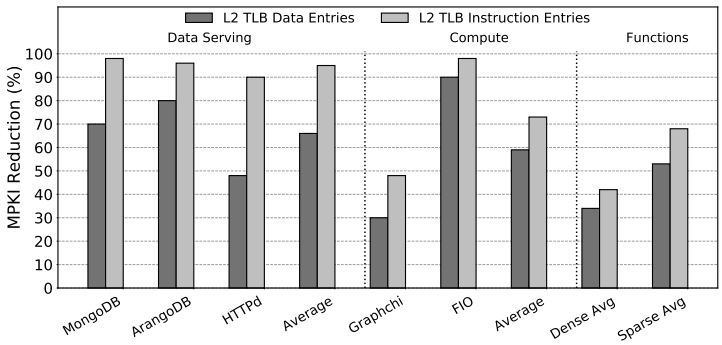


Figure 6.10: L2 TLB MPKI reduction attained by BabelFish.

From the figure, we see that BabelFish reduces the MPKI across the board. For example, for Data Serving, the data MPKI reduces by 66%, and the instruction MPKI reduces even more, by 96%. These are substantial reductions. Good results are also seen for the Compute workloads. Functions see smaller MPKI reductions, as they are short lived and interfered by the docker engine/OS.

The reductions come from various sources. First, a container may reuse the L2 TLB entries of another container and avoid misses. Second, sharing L2 TLB entries effectively increases TLB capacity, which reduces misses. Finally, as a result of these two effects, there is a lower chance that co-scheduled processes evict each other's TLB entries.

To gain more insight into L2 TLB entry sharing, Figure 6.11 shows the number of hits on L2 TLB entries that were brought into the L2 TLB by processes other than the one performing the accesses. We call them *Shared Hits* and show them as a fraction of all L2 TLB hits. The figure is organized as Figure 6.10. As we can see, the percentage of shared hits is generally sizable, but varies across applications, as it is dependent on the applications' access patterns.

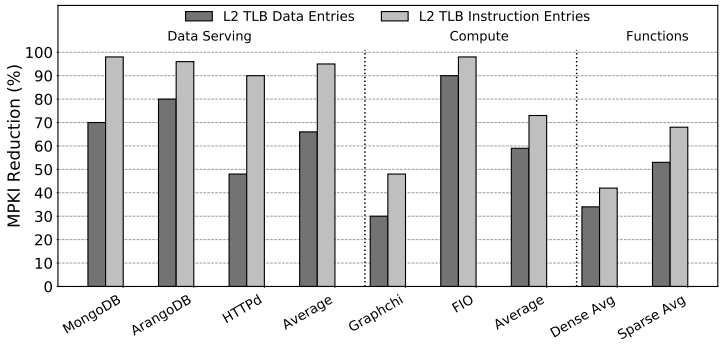


Figure 6.11: Hits on L2 TLB entries that were brought into the TLB by processes other than the one issuing the accesses. We call them Shared Hits and show them as a fraction of all L2 TLB hits.

For example, GraphChi shows 48% shared hits for instructions and 12% for data. This is because PageRank's code is regular, while its data accesses are fairly random, causing variation between the data pages accessed by the two containers. Overall, BabelFish's TLB entry sharing bolsters TLB utilization and reduces page walks.

6.7.3 Latency or Execution Time Reduction

To assess the performance impact of BabelFish, we report different metrics for different applications: reduction in mean and 95th percentile (*Tail*) latency in Data Serving applications; reduction in execution time in Compute applications; and reduction in bring-up time and function execution time in Functions. Figure 6.12 shows the results, all relative to *Baseline*. To gain further insight, Table 6.2 shows what fraction of the performance improvement in Figure 6.12 comes from TLB effects; the rest comes from page table effects. Recall that transparent huge pages are enabled in all applications but MongoDB and ArangoDB [331, 332].

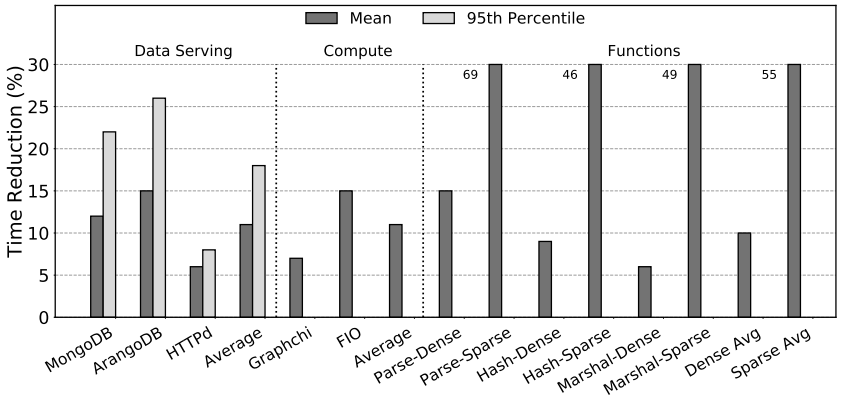


Figure 6.12: Latency/time reduction attained by BabelFish.

Consider first the Data Serving applications. On average, BabelFish reduces their mean and tail latencies by a significant 11% and 18%, respectively. The reductions are higher in MongoDB and ArangoDB than in HTTPd. It can be shown that this is because, in *Baseline*, address translation induces more stress in the MongoDB and ArangoDB database applications than in the stream-based HTTPd application. Hence, BabelFish is more effective in the former. Table 6.2 shows that MongoDB gains more from L2 TLB entry sharing, while ArangoDB more from page table entry sharing. Therefore, both types of entry sharing are helpful.

Compute applications also benefit from BabelFish. On average, their execution time reduces by 11%. GraphChi has lower gains because it performs low-locality accesses in the graph, which makes it hard for one container to bring shared translations that a second container can reuse. On

Data Sharing	Functions
MongoDB: 0.77	Parse-Dense: 0.15
ArangoDB: 0.25	Parse-Sparse: 0.01
HTTPd: 0.81	Hash-Dense: 0.18
Average: 0.61	Hash-Sparse: 0.01
Compute	Marshal-Dense: 0.28
Graphchi: 0.11	Marshal-Sparse: 0.02
FIO: 0.29	Dense Average: 0.20
Average: 0.20	Sparse Average:0.01

Table 6.2: Fraction of time reduction due to L2 TLB effects.

the other hand, FIO has higher gains because its more regular access patterns enable higher shared translation reuse. Table 6.2 shows that these applications benefit more from page table effects.

Recall that we run the Functions in groups of three at a time. The leading function behaves similarly in both BabelFish and *Baseline* due to cold start effects. Hence, Figure 6.12 shows the reduction in execution time for only the other two functions in the group. We see that the reductions are heavily dependent on the access patterns. Functions with dense access patterns access only a few pages, and spend little time in page faults. Hence, their execution time decreases by only 10% on average. In contrast, functions with sparse access patterns access more pages and spend more time servicing page faults. Hence, BabelFish reduces their execution time by 55% on average. In all cases, as shown in Table 6.2, most gains come from page table entry sharing.

Finally, although not shown in any figure, BabelFish speeds-up function bring-up by 8%. Most of the remaining overheads in bring-up are due to the runtime of the Docker engine and the interaction with the kernel. Overall, BabelFish speeds-up applications across the board substantially, even in our conservative environment where we co-locate only 2-3 containers per core.

BabelFish vs Larger TLB.

BabelFish's main hardware requirements are additional bits in the L2 TLB for the CCID and O-PC fields. It could be argued that this extra hardware could be used instead to make a conventional TLB larger. Hence, we have re-run the workloads with a conventional architecture with this larger L2 TLB. On average, the resulting architecture reduces the mean request latency of Data Serving applications by 2.1%, the execution time of Compute applications by 0.6%, and the execution time of Functions by 1.1% (dense) and 0.3% (sparse).

These reductions are much smaller than those attained by BabelFish (Figure 6.12). One reason is that BabelFish benefits from both L2 TLB and page table effects. A second reason is that BabelFish also benefits from processes prefetching TLB entries for other processes into the L2

TLB and caches. Overall, this larger L2 TLB is not a match for BabelFish.

6.7.4 BabelFish Resource Analysis

We analyze the hardware and software resources needed by BabelFish. We also analyze the resources of a design where, as soon as a write occurs on a CoW page, sharing for the corresponding PMD table set immediately stops, and all sharers get private page table entries. This design does not need a PC bitmask.

Hardware Resources. BabelFish's main hardware overhead is the CCID and O-PC fields in the TLB, and the associated comparison logic (Figure 6.1). We estimate that this extra hardware adds 0.4% to the area of a baseline core (without L2). If we eliminate the need for the PC bitmask bits, the area overhead falls to 0.07%. These are very small numbers.

Table 6.3 shows several parameters of the L2 TLB, both for *Baseline* and BabelFish: area, access time, dynamic energy of a read access, and leakage power. The data corresponds to 22nm, and is obtained with CACTI [100]. The table shows that the difference in TLB access time between *Baseline* and BabelFish is a fraction of a cycle. To be conservative, we add two extra cycles to the access time of the BabelFish L2 TLB when the PC bitmask has to be accessed.

Configuration	Area	Access Time	Dyn. Energy	Leak. Power
Baseline	0.030 mm2	327 ps	10.22 pJ	4.16 mW
BabelFish	0.062 mm2	456 ps	21.97 pJ	6.22 mW

Table 6.3: Parameters of the L2 TLB at 22nm.

Memory Space. The memory space of BabelFish is minimal. It includes one MaskPage with PC bitmasks and pid_list (Figure 6.8) for each 512 pages of *pte_ts*. This is 0.19% space overhead. In addition, it includes one 16-bit counter per 512 *pte_ts* to determine when to de-allocate a page of *pte_ts* (Section 6.4.3). This is 0.048% space overhead. Overall, BabelFish only adds 0.238% space overhead. If we eliminate the need for the PC bitmask bits, the first item goes away, and the total space overhead is 0.048%.

Software Complexity. We implement BabelFish's page table sharing mechanism in the Linux kernel and in the Simics shadow page tables. We require about 300 Lines of Code (LoC) in the MMU module, 200 LoC in the page fault handler, and 800 LoC for page table management operations.

Huge Pages. BabelFish transparently supports huge pages, which are in fact a complementary way to reduce TLB and cache pressure. Recent work has tackled huge-page bloating and fragmentation issues [275, 277].

Translation Management and TLB Design. Recent work [333, 334, 335] aims to reduce the translation coherence overheads with software and hardware techniques. Other work provides very fine grain protection domains and enables sub-page sharing, but does not support translation sharing [336]. CoLT and its extension [337, 338] propose the orthogonal idea of coalesced and clustered TLB entries within the same process.

MIX-TLB [339] supports both huge page and regular page translations in a single structure, increasing efficiency. In [340], self-invalidating TLB entries are proposed to avoid TLB shoot-downs. Shared last-level TLB [341] aims to reduce translation overhead in multi-threaded applications. Recent work [342] proposes to prefetch page table entries on TLB misses. Other work [343, 344] shows optimizations for CPU-GPU environments. These approaches tackle a set of different problems in the translation process and can co-exist with BabelFish.

Elastic cuckoo page tables [4] propose a hashed page table design based on elastic cuckoo hashing. Such scheme can be augmented with an additional hashed page table where containers in a CCID group share page table entries. Auxiliary structures called Cuckoo Walk Tables could be enhanced to indicate whether a translation is shared. Since the TLB is not affected by elastic cuckoo page tables, BabelFish's TLB design remains the same.

Other work has focused on virtualized environments. POM-TLB and CSALT [312, 345] propose large in-memory TLBs and cache partitioning for translations. DVMT [346] proposes to reduce 2D page walks in virtual machines by enabling application-managed translations. RMM [238] proposes redundant memory mappings and [287] aims to reduce the dimensionality of nested page walks. PageForge [6] proposes near-memory extensions for content-aware page merging. This deduplication process shares pages in virtualized environments, generating an opportunity to further share translations. These solutions are orthogonal to BabelFish in a scenario with containers inside VMs.

Khalidi and Talluri [347] propose a scheme to share TLB entries between processes. The idea is to tag each TLB entry with a PCID or a bitvector. The bitvector uses one-hot encoding to identify one of 10-16 different collections of shared translations. A TLB is accessed twice: with the PCID and with the bitvector. If the bitvector matches, the shared translation is retrieved. The authors also suggest tagging global hashed page tables with PCIDs or bitvectors. While this scheme allows translation sharing, compared to BabelFish, it has several limitations. First, unlike BabelFish, the scheme is not scalable because the number of different collections of translations that can be shared

is limited to the number of bits in the bitvector. Second, to find a shared translation, the TLB is accessed twice. Third, unlike BabelFish, the scheme does not support CoW or selective sharing of a translation. In addition, it does not support ASLR. Finally, BabelFish also proposes the sharing of multi-level page table entries.

Native Execution Environment. A software-only approach that improves zygote fork and application performance in Android by sharing page translations across applications is presented in [348]. This scheme only shares code translations of 4KB pages in a 2-level page table design. This solution requires disabling TLB tagging and marking TLB entries as global, which leads to TLB flushes at context switches. Moreover, this solution does not support the sharing of data translations. In contrast, BabelFish shares both code and data translations in both TLB and page tables, for multiple page sizes, while supporting tagged TLBs and CoW.

6.9 CONCLUSION

Container environments create replicated translations that cause high TLB pressure and redundant kernel work during page table management. To address this problem, we proposed *BabelFish*, a novel architecture to share address translations across containers in the L2 TLB and in the page tables. We evaluated BabelFish with simulations of an 8-core processor running a set of Docker containers. On average, BabelFish reduced the mean and tail latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowered the execution time of containerized compute workloads by 11%. Finally, it reduced serverless function bring-up time by 8% and execution time by 10%–55%. Overall, BabelFish sped-up applications across the board substantially, even in our conservative environment where we co-located only 2-3 containers per core.

Chapter 7: Near-Memory Content-Aware Page-Merging

7.1 INTRODUCTION

Cloud computing is based on virtualization technology. Public clouds such as Google's Compute Engine [290], Amazon's EC2 [349], IBM's Cloud [292], and Microsoft's Azure [293], as well as private ones based on OpenStack [350] and Mesos [351], rely heavily on virtualization to provide their services. Virtual Machines (VMs) enable server consolidation by allowing multiple service instances to run on a single physical machine, while providing strong failure isolation guarantees, independent configuration capabilities among instances, and protection against malicious attacks. In such environments, VMs can be easily multiplexed over CPU resources, by sharing processor time. However, each VM requires its own private memory resources, resulting in a large total memory footprint. This is especially concerning as more cores are integrated on chip and, as a result, main memory sizes are increasingly insufficient [352, 353].

Indeed, server consolidation and the emergence of memory intensive workloads in the datacenter has prompted cloud providers to equip machines with hundreds of GBs of memory. Such machines are expensive for several reasons. First, the cost of DRAM is more than an order of magnitude higher than flash memory, and more than two orders of magnitude higher than disk [354]. In addition, adding more memory chips to a machine is limited by the number of available slots in the motherboard, which often leads to the replacement of existing DRAM modules with denser ones, increasing cost. Finally, the additional memory consumes more power, which increases the cost of ownership. In spite of these higher costs, we expect users to continue to request more memory over time.

An effective way to decrease memory requirements in virtualized environments is same-page merging or page deduplication. The idea is to identify virtual pages from different VMs that have the same data contents, and map them to a single physical page. The result is a reduced main memory footprint. VMware adopts this technique with the ESX server [353], and attains memory footprint reductions of 10–40%. Similar approaches, like the Difference Engine [352], further extend page merging to include subpage-level sharing and memory compression, and attain over 65% memory footprint reductions. RedHat's implementation of page-merging, called Kernel Same-page Merging (KSM) [355], targets both virtualized and scientific environments. It has been integrated into the current Linux kernel and the KVM hypervisor. A recent study showed that KSM can reduce the memory footprint by about 50% [356].

Unfortunately, same-page merging is a high-overhead operation. Processors scan the memory space of the VMs and compare the contents of their pages exhaustively. When two identical pages

are found, the hypervisor updates the page table mappings, and frees one physical page. To reduce the overhead of same-page merging, numerous optimizations have been adopted. One of them is the assignment of hash keys to pages, based on the contents of a portion of the page. For example, in KSM, a per-page hash key is generated based on 1KB of the page's contents, and is used to detect whether the page's contents change. Still, in modern server-grade systems with hundreds of GBs of memory, the performance overhead of same-page merging can be significant. It is especially harmful in user-facing services, where service-level-agreements (SLAs) mandate response times of a few milliseconds or even microseconds [357]. Unluckily, this effect will only become worse with the increase in memory size induced by emerging non-volatile memories.

In this chapter, we eliminate the processor cycles and cache pollution induced by same-page merging by performing it with hardware near memory. To the best of our knowledge, this is the first solution for hardware-assisted same-page merging that is *general*, *effective*, and requires *modest hardware modifications and hypervisor involvement*. We call our proposal *PageForge*. It augments the memory controller with a small hardware module that efficiently performs page scanning and comparison semi-autonomously and transparently to the VMs. In addition, it repurposes the ECC engine in the memory controller to generate accurate and cheap ECC-based hash keys.

We evaluate PageForge with simulations of a 10-core processor with a VM on each core, running a set of applications from the TailBench suite. When compared with RedHat's KSM, a state-of-the-art software implementation of page merging, PageForge attains identical savings in memory footprint while substantially reducing the overhead. Compared to a system without same-page merging, PageForge reduces the memory footprint by an average of 48%, enabling the deployment of twice as many VMs for the same physical memory. Importantly, it keeps the average latency overhead to 10%, and the 95^{th} percentile tail latency to 11%. In contrast, in KSM, these latency overheads are 68% and 136%, respectively.

7.2 BACKGROUND

Same-page merging consists of utilizing a single physical page for two or more distinct virtual pages that happen to contain the same data. The goal is to reduce the consumption of physical memory. This process is also called page deduplication or content-based sharing. In same-page merging, there are two main operations. First, each page is associated with a hash value obtained by hashing some of the page's data. Second, potentially identical pages are exhaustively compared; if two pages contain the same data, they are merged into a single physical page.

Same-page merging is highly successful in datacenter environments. In such environments, a significant amount of duplication occurs across VMs. The reason is that multiple VMs co-located

in the same machine often run the same libraries, packages, drivers, kernels, or even datasets. This pattern can be exploited by the hypervisor to reduce the memory footprint of the VMs.This allows the deployment of additional VMs without additional hardware.

Figure 7.1(a) shows an example where two VMs have two pages of data each. Page 2 in VM-0 and Page 3 in VM-1 have the same contents. Same-page merging changes the Guest Physical Address to Host Physical Address mapping of Page 3 as shown in Figure 7.1(b), and frees up one physical page. This operation is performed by the hypervisor transparently to the VMs. From now on, both VMs will share the same physical page in read-only mode. If one of the VMs writes to the page, a new physical page is allocated, reverting the system back to Figure 7.1(a).

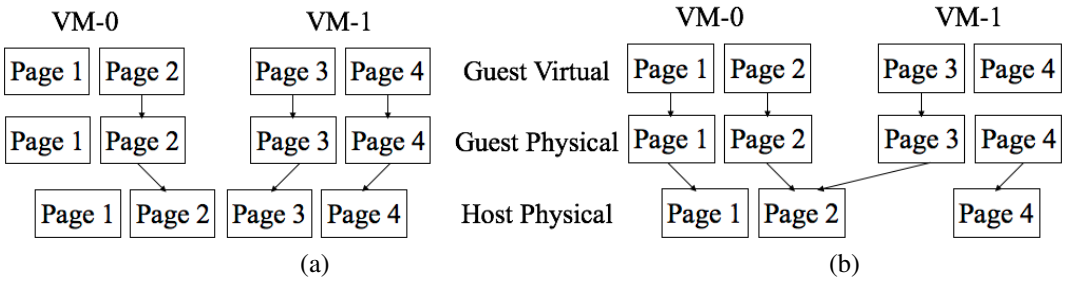


Figure 7.1: Example of page mapping without (a) and with (b) page merging. Pages 2 and 3 are identical.

There are software approaches to optimize same-page merging in virtualized environments [352, 353, 355, 358, 359, 360, 361, 362, 363]. However, software solutions can introduce substantial execution overhead, especially in latency-sensitive cloud applications. The overhead is due to processor cycles consumed by the page-merging process, and the resulting cache pollution. There have been some proposals for hardware to reduce data redundancy, but they either address only part of the problem (e.g., data redundancy in the caches [364]), or require a major redesign of the system [365]. All of these approaches are discussed in detail in Section 7.7.

7.2.1 RedHat's Kernel Same-page Merging

Kernel Same-page Merging (KSM) [355] is a state-of-the-art open-source software implementation of same-page merging by RedHat [366]. It is incorporated in the Linux kernel and targets RedHat's KVM-enabled VMs [367, 368]. Furthermore, KSM is one of the key components of Intel's recently proposed Clear Containers [369, 370]. When a VM is deployed, it provides a hint to KSM with the range of pages that should be considered for merging. In the current implementation, this is done with the *madvise* system call [371] and the *MADV_MERGEABLE* flag.

KSM continuously scans all the pages that have been marked as mergeable, discovers pages with identical content, and merges them. KSM places pages in two red-black binary trees: the *Stable* and the *Unstable* trees. The stable tree stores pages that have been successfully merged, and are marked as Copy-on-Write (CoW); the unstable tree tracks unmerged pages that have been scanned in the previous pass, and may have remained unchanged since then. Each tree is indexed by the contents of the page.

Algorithm 7.1 shows pseudo code for KSM. KSM runs all the time while there are mergeable pages (Line 3). It is organized in passes. In each pass, it goes over all the mergeable pages (Line 5), picking one page at a time (called *candidate page*), trying to merge it. During a pass, KSM accesses the stable tree and creates the unstable tree. At the end of each pass, it destroys the unstable tree.

Algorithm 7.1: RedHat's Kernel Same-page Merging.

```
tree initialization
/* Running all the time */
while mergeable pages > 0 do
    /* Go over all mergeable pages */
    while pages for this pass > 0 do
        candidate_pg = next page in pass
        if search(stable_tree,candidate_pg) then
            merge(stable_tree, candidate_pg)
        else
            /* Not found in stable tree */
            new_hash = compute_hash(candidate_pg)
            if new_hash == previous_hash(candidate_pg) then
                if search(unstable_tree, candidate_pg) then
                    merged_pg=merge(unstable_tree,candidate_pg)
                    cow_protect(merged_pg)
                    remove(unstable_tree, merged_pg)
                    insert(stable_tree, merged_pg)
                else
                    /* Not found in unstable tree */
                    insert(unstable_tree, merged_pg)
                end
            end
            /* Drop the page */
        end
    end
    /* Throw away and regenerate */
    reset(unstable_tree)
end
```

After KSM picks a candidate page, it performs several operations. First, it uses the page to search the stable tree (Line 7). To search the tree, it starts at the root, comparing the candidate

page to the root page byte-by-byte. If the data in the candidate page is smaller or larger than in the root page, KSM moves left or right in the tree, respectively. It then compares the candidate page to the page at that position in the tree, and moves left or right depending on the result. If KSM finds that the pages are identical, it merges the candidate page with the page at that position in the tree (Line 8). This involves updating the mapping of the candidate page to point to that of the page in the tree, and reclaiming the memory of the candidate page.

Figure 7.2(a) shows an example of a red-black tree with pages as nodes. Assume that the candidate page contents are identical to Page 4. KSM starts by comparing the candidate page to Page 3. As soon as the data diverges, KSM moves to compare the candidate page to Page 5, and then to Page 4. After that, it merges it to Page 4.

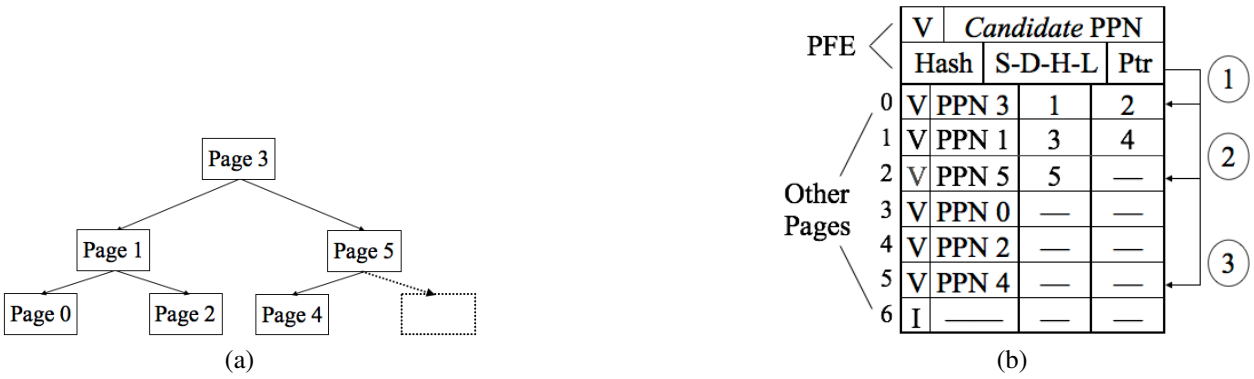


Figure 7.2: Example page tree (a) and corresponding PageForge Scan Table (b).

If a match is not found in the stable tree (line 10), KSM checks if the candidate page was modified since the last pass. To perform this check, it uses the data in the page to generate a hash key (Line 11). The hash key is generated with the jhash2 function [372] and 1KB of the page's data. KSM then compares the newly generated hash key to the hash key generated in the previous pass. If the keys are not the same, it means that the page has been written. In this case (or if this is the first time that this page is scanned), KSM does not consider this page any more (Line 22) and picks a new candidate page. Otherwise, KSM proceeds to compare the candidate page to those in the unstable tree (Line 13).

The search in the unstable tree proceeds as in the stable tree. There are two possible outcomes. If a match is not found, KSM inserts the candidate page in the unstable tree (Line 20). Note that the pages in the unstable tree are not write-protected and hence may change over time. If a match is found, KSM merges the two pages by updating the mappings of the candidate page (Line 14). Then, it protects the merged page with CoW (Line 15). Finally, it removes it from the unstable tree and inserts it into the stable tree (Lines 16-17). Note that, in practice, after a match is found, KSM immediately sets the two pages to CoW, and performs a second comparison of the pages. This is

done to protect against racing writes during the first comparison.

Overall, the stable tree contains pages that are merged and are under CoW, while the unstable tree contains pages that may change without notification. If a write to an already merged page occurs, then the Operating System (OS) enforces the CoW policy by creating a copy of the page and providing it to the process that performed the write. The status and mapping of the other page(s) mapped to the original physical page remain intact.

Two parameters are used to tune the aggressiveness of the algorithm. First, *sleep_millisecs* is the amount of time the KSM process sleeps between work intervals, and is usually a few milliseconds. Second, *pages_to_scan* is the number of pages to be scanned at each work interval, and is usually a few hundred to a few thousand pages. In the current version of the Linux kernel, KSM utilizes a single worker thread that is scheduled as a background kernel task on any core in the system. For big servers with several VMs and hundreds of GBs of memory, a whole core can be dedicated to this process [355].

7.2.2 Memory Controller and ECC

Memory ECC provides single error correction and double error detection (*SECDED*), and is usually based on Hamming or similar codes [373, 374]. Previous work [375, 376, 377] has extensively studied and optimized ECC and its applicability in the datacenter. DRAM devices are commonly protected through 8-16 bits of ECC for every 64-128 data bits. In commercial architectures, an ECC encode/decode engine is placed at the memory controller.

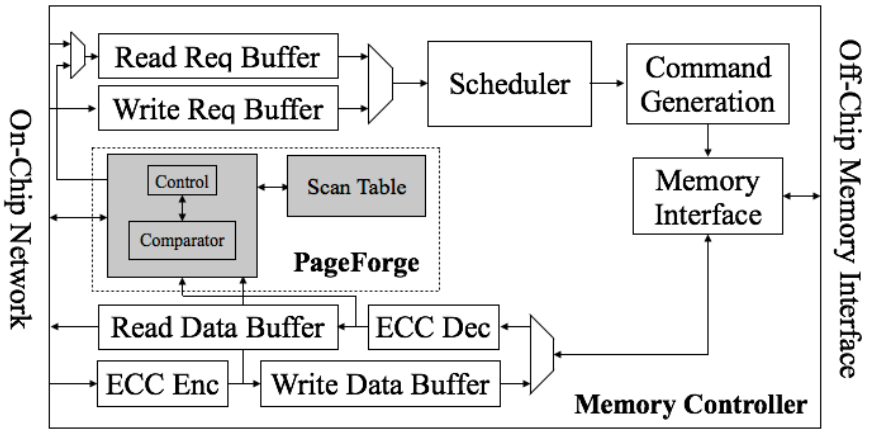


Figure 7.3: PageForge memory controller architecture.

Figure 7.3 shows a block diagram of a memory controller with ECC support. When a write request arrives at the memory controller, it is placed in the write request buffer, and the data block being written goes through the ECC encoder. Then, the generated ECC code and the data block

are placed in the memory controller's write data buffer. The data block eventually gets written to the DRAM, and the ECC code is stored in a spare chip. Figure 7.4 shows one side of a DRAM DIMM with eight 8-bit data chips and one 8-bit ECC chip.

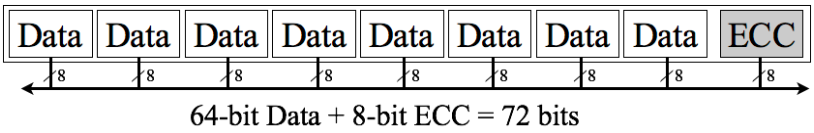


Figure 7.4: A single side of a DRAM DIMM architecture.

When a read request arrives at the memory controller, it is placed in the read request buffer. Eventually, the request is scheduled and the command generation engine issues the necessary sequence of commands to the memory interface. When the DRAM response arrives at the memory controller, the ECC code for the requested block arrives along with the data block. The ECC code is decoded, and the data block is analyzed for errors by checking the bit values in the data block and in the ECC code. If there is no error, the block is placed in the read data buffer to be delivered to the network. Otherwise, the ECC repairs the data block or notifies the software.

7.3 PAGEFORGE DESIGN

7.3.1 Main Idea

Existing proposals to support same-page merging in software [352, 353, 355, 358, 359, 360, 361, 362, 363] induce significant performance overhead — caused by both processor cycles required to perform the work, and the resulting cache pollution. This overhead is especially harmful in latency-sensitive cloud applications. There have been some proposals for hardware to reduce data redundancy, but they either address only part of the problem (e.g., data redundancy in the caches [364]), or require a major redesign of the system [365].

Our goal is to devise a solution for hardware-assisted same-page merging that is *general*, *effective*, and requires *modest hardware modifications and hypervisor involvement*. Our idea is to identify the fundamental operations in same-page merging, and implement them in hardware. Moreover, such hardware should be close to memory and not pollute caches. The remaining operations should be done in software, which can be changed based on the particular same-page merging algorithm implemented.

We identify three operations that should be implemented in hardware: (i) pairwise page comparison, (ii) generation of the hash key for a page, and (iii) ordered access to the set of pages that

need to be compared to the candidate page. The software decides what is this set of pages that need to be compared. This hardware support is not tied to any particular algorithm.

Our design is called *PageForge*. It includes three components. The first one is a state machine that performs pairwise page comparison. The second one is a novel and inexpensive way to generate a hash key for a page by reusing the ECC of the page. The last one is a hardware table called *Scan Table*, where the software periodically uploads information on a candidate page and a set of pages to be compared with it. The hardware then accesses these pages and performs the comparisons. These three components are placed in the memory controller. With this design, accesses to memory are cheap and, because the processor is not involved, caches are not polluted.

Figure 7.3 shows a memory controller with the PageForge hardware shadowed. It has a Scan Table, a page comparator, and some control logic. Figure 7.5 shows a multicore with two memory controllers. One of them has the PageForge hardware. The L3 slices and the memory controllers connect to the interconnect.

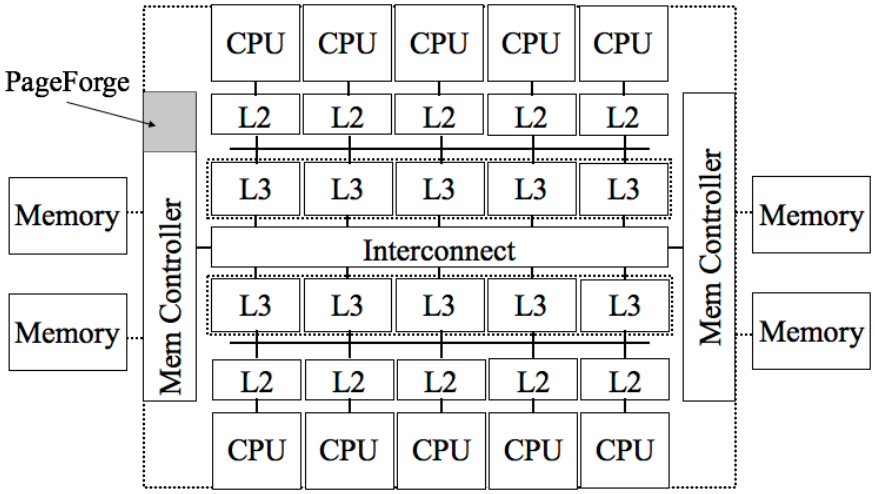


Figure 7.5: Multicore with PageForge in one of the memory controllers.

In the following, we discuss the process of page scanning and the Scan Table (Section 7.3.2), exploiting ECC codes for hash keys (Section 7.3.3), implementing the KSM algorithm (Section 7.3.4), interaction with the cache-coherence protocol (Section 7.3.5), and the software interface (Section 7.3.6). PageForge is not tied to any particular same-page merging algorithm.

7.3.2 Page Scanning and Scan Table

Figure 7.2(b) shows the structure of the Scan Table. The *PageForge (PFE)* entry contains information on the candidate page, while the *Other Pages* entries have information on a set of pages

that should be compared with the candidate page. The PFE entry holds a Valid bit (V), the Physical Page Number (PPN) of the candidate page, the hash key of the page, a few control bits, and a pointer (Ptr) to one of the Other Pages in the Scan Table (i.e., the page which it is currently being compared with). The control bits are: Scanned (S), Duplicate (D), Hash Key Ready (H), and Last Refill (L). We will see their functionality later. Each entry in Other Pages contains a Valid bit (V), the PPN of the page, and two pointers to Other Pages in the Scan Table (called Less and More). Such pointers point to the next page to compare with the candidate page after the current-page comparison completes. If the current-page comparison finds that the candidate page's data is smaller than the current page's, the hardware sets Ptr to point where Less points; if it is higher, the hardware sets Ptr to point where More points. The pointed page is the next page to compare with the candidate page.

The Search for Identical Pages

The scanning process begins with the OS selecting a candidate page and a set of pages to compare with it. The OS inserts the candidate's information in the PFE entry, and the other pages' information in the Other Pages entries. The OS then sets the Less and More fields of each of the Other Pages entries based on the actual same-page merging algorithm that it wants to implement. It also sets the Ptr field in the PFE entry to point to the first entry in the Other Pages. Finally, it triggers the PageForge hardware.

The PageForge hardware initiates the scanning by comparing the data in the candidate page with that in the page pointed to by Ptr. PageForge issues requests for the contents of the two pages. To generate a memory request, the hardware only needs to compute the offset within the page and concatenate it with the PPN of the page. Since a single data line from each page is compared at a time in lockstep, PageForge reuses the offset for the two pages.

The outcome of the page comparison determines the next steps. If the two pages are found to be identical after the exhaustive comparison of their contents, the Duplicate and Scanned bits in the PFE are set, and comparison stops. Otherwise, the hardware updates Ptr based on the result of the comparison of the last line. If the line of the candidate page was smaller, it sets Ptr to the Less pointer; if was larger, it sets Pts to the More pointer. The page comparison restarts.

If Ptr points to an invalid entry, PageForge completed the search without finding a match. In this case, only the Scanned bit is set. The OS periodically checks the progress of PageForge. If it finds the Scanned bit set and the Duplicate bit clear, it reloads the Scan Table with the next set of pages to compare with the candidate page. This process stops when the candidate page has been compared to all the desired pages. Then, a new candidate page is selected.

Interaction with the Memory System

During the scanning process, a required line can either be residing in the cache subsystem or in main memory. In a software-only implementation of same-page merging, all the operations are performed at a core and, hence, are guaranteed to use the most up-to-date data values thanks to cache coherence. However, two downsides of this approach are that it utilizes a core to perform the scanning process, and that it pollutes the cache hierarchy with unnecessary data.

The goal of PageForge is to alleviate both downsides. To achieve this, the control logic issues each request to the on-chip network first. If the request is serviced from the network, no other action is taken. Otherwise, it places the request in the memory controller's Read Request Buffer, and the request is eventually serviced from the DRAM. If, before the DRAM satisfies the request, another request for the same line arrives at the memory controller, then the incoming request is coalesced with the pending request issued from PageForge. Similarly, coalescing also occurs if a request was pending when PageForge issues a request to memory for the same line.

7.3.3 Exploiting ECC Codes for Hash Keys

An integral part of any same-page merging algorithm is the generation of hash keys for pages. Hash keys are used in different ways. For example, the KSM algorithm generates a new hash key for a page and then compares it to the previous key for the same page, to estimate if the page remains unchanged. Other algorithms compare the hash keys of two pages to estimate if the two pages contain the same data and can be merged.

Fundamentally, the key is used to avoid unnecessary work: if two keys are different, we know that the data in the two pages is different. However, false positives exist, where one claims that the pages have the same data but they do not. The reason is two-fold. First, a key hashes only a fraction of the page contents and, second, hashes have collisions. However, the probability of false positives does not affect the correctness of the algorithm. This is because, before PageForge merges two pages, it first compares them exhaustively.

Designing ECC-Based Hash Keys

PageForge introduces a novel and inexpensive approach to generate hash keys using memory ECC codes. PageForge generates the hash key of a page by concatenating the ECC codes of several, fixed-location lines within the page. This approach has two main advantages. First, the key is very simple to generate, as it simply requires reading ECC bits. Second, the PageForge hardware generates the key of a candidate page in the background, as it compares the page with a

set of pages in the scanning algorithm. This is because, to compare the page, PageForge brings the lines of the page to the memory controller. If a line comes from the DRAM, the access also brings the line's ECC code; if the line comes from a cache, the circuitry in the memory controller quickly generates the line's ECC code.

Figure 7.6 shows a possible implementation. It assumes a 4 KB page with 64B lines and, for each line, an 8B ECC code. PageForge logically divides a 4KB page into four 1KB sections, and picks a different (but fixed) offset within each section. It then requests the lines at each of these four offsets, and picks the least-significant 8-bits of the ECC codes of these lines (called minikeys). These minikeys are concatenated together to form a 4B hash key. Hence, PageForge only brings 256B from memory to generate a hash key.

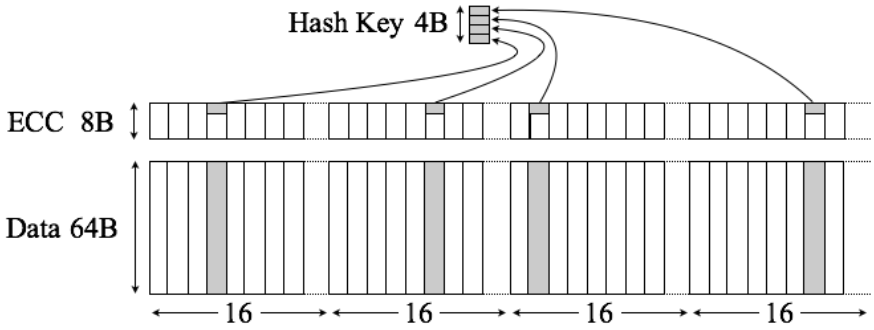


Figure 7.6: Example implementation of ECC-based hash keys.

Compare this to KSM. To generate a hash key, KSM requires 1KB of consecutive data from the page. In addition, its *jhash* hash function is serial, meaning that it traverses the data from the beginning to the end in order. If we were to implement a hash function similar to jhash in hardware, we would require to buffer up to 1KB of data. This is because some requests could be serviced from the caches, while other requests would go to main memory, and hence the responses could arrive at the memory controller out-of-order. PageForge reduces the memory footprint required for key generation by 75%. Moreover, it can read the lines to produce the hash key out-of-order. We evaluate the impact on accuracy in Section 7.6.2.

As indicated above, PageForge generates the hash key of the candidate page during the process of comparing the page to the other pages in the Scan table. As soon as the hash key is fully generated, the hardware stores the hash key in the PFE entry, and sets the Hash Key Ready (H) bit. It could be possible, however, that the hardware would be unable to complete the hash key by the time all the comparisons of the candidate page are done. To avoid this case, when the OS reloads the Scan table with the last set of pages to compare to the candidate page, it sets the Last Refill (L) bit. This forces the hardware to complete the generation of the hash key during this last processing of the entries in the Scan table.

Interaction with the Memory Controller

PageForge follows a series of steps to generate ECC-based hash keys. Consider again Figure 7.3, which shows PageForge's basic components and their connections to the ECC engine in the memory controller. The PageForge control logic generates memory requests as described in Section 7.3.2. A PageForge request can be serviced either from the on-chip network or from the DRAM, depending on where the requested line resides. In case it is serviced from the on-chip network, the response enters the memory controller and goes through the ECC engine. The control logic of PageForge snatches the generated ECC code. If it is serviced from the DRAM, PageForge grabs the ECC code from the ECC decoder.

7.3.4 Example: Implementing the KSM Algorithm

We now describe how PageForge supports the KSM algorithm. The Scan Table is used first for the stable tree, and then for the unstable tree. In both cases, the process is similar. As PageForge picks a candidate page, it loads the page information in the PFE entry. Then, it takes the root of the red-black tree currently being searched, and a few subsequent levels of the tree in breadth-first order, and loads their information in the Other Pages entries. The Less and More fields are set accordingly. Then, the OS triggers the hardware.

Figure 7.2(b) shows the Scan Table for the example described in Section 7.2.1 and shown in Figure 7.2(a). We have inserted 6 Other Pages entries in the Scan Table, corresponding to all the nodes in the tree. Recall that we assume that the candidate page contents are identical to Page 4. In step ①, Ptr points to the root of the tree, which is Entry 0 in the Table. After the comparison, the result shows that the candidate page is greater than Page 3. Hence, in step ②, PageForge updates Ptr to point to Entry 2, which is the right child of Page 3. PageForge then compares the candidate page to Page 5. This time the result is that the candidate page is smaller. Hence, in step ③, PageForge updates Ptr to point to Entry 5. After these steps, PageForge finds out that the candidate page and Page 4 are duplicates, and sets the Duplicate and Scanned bits of the PFE entry.

If, instead, a match was not found, the OS reloads the Scan table and triggers the PageForge hardware again to continue the search. This time, the pages loaded into the Other Pages entries of the Scan Table are those in the first few levels of the subtree on the right or on the left of Page 4 (depending on the outcome of the previous comparison to Page 4).

During the search of the stable tree, PageForge generates the hash key of the candidate page in the background, and stores it in the Hash field of the PFE entry. If a match in the stable tree was not found, the OS compares the newly-generated hash key of the candidate page with the page's old hash key (saved by the OS in advance). Based on the outcome of the comparison, PageForge

either proceeds to search the unstable tree, or picks a new candidate page.

7.3.5 Interaction with the Cache Coherence

The PageForge module in the memory controller accesses and operates on data that may be cached in the processor's caches. Consequently, it has to participate in the cache coherence protocol to some extent. However, in our design, we have tried to keep the hardware as simple as possible.

When the memory controller issues a request on the on-chip network, it has to obtain the latest value of the requested cache line. Such request is equivalent to a request from a core. If the chip supports a snoopy protocol, all the caches are checked, and one may respond; if the chip supports a directory protocol, the request is routed to the directory, which will obtain the latest copy of the line and provide it.

However, the PageForge module does not have a cache and, therefore, does not participate as a supplier of coherent data. Instead, PageForge uses read and write data buffers in the memory controller to temporarily store its requests. If the chip supports a snoopy protocol, PageForge does not participate in the snooping process for incoming requests; if the chip supports a directory protocol, PageForge is not included in the bit vector of sharers.

This design may not produce optimal performance; however, it simplifies the hardware substantially. One could envision PageForge having a cache to cache the candidate page, or even multiple candidate pages at the same time. Caching would eliminate the need to re-read the candidate page into the PageForge module multiple times, as the candidate page is compared to multiple pages. This would reduce memory bandwidth and latency. However, this design would have to interact with the cache coherence protocol and would be substantially more complex.

It is possible that, while a candidate page is being compared to another page, either page is written to. In this case, PageForge, as well as the original software implementation of KSM, may be making decisions based on a line with a stale value. In practice, this does not affect the semantics of the merging process. The reason is that, before the actual page merging, a final comparison of the two pages is always performed under write protection, which guarantees that the page merging is safe.

7.3.6 Software Interface

PageForge provides a five-function interface for the OS to interact with the hardware. The interface is shown in Table 7.1. The main functions are *insert_PPN* and *insert_PFE*, which enable the OS to fill Other Pages and PFE entries, respectively, in the Scan table.

Function	Operands	Semantics
insert_PPN	Index, PPN, Less, More	Fill an Other Pages entry at the specified index of the Scan Table
insert_PFE	PPN, L, Ptr	Fill the PFE entry in the Scan Table
update_PFE	L, Ptr	Update the PFE entry in the Scan Table
get_PFE_info		Get the hash key, Ptr, and the S, D, and H bits from the Scan Table
update_ECC_offset	Page offsets	Update the offsets used to generate the ECC-based hash keys

Table 7.1: API used by the OS to access PageForge.

insert_PPN fills one of the Other Pages entries in the Scan table. It takes as operands the index of the entry to fill, the PPN of the page, and the Less and More indices. Recall that Less is the index of the next page to compare if the data in the candidate page is smaller than that in the current page; More is the index of the next page to compare if data in the candidate page is larger than that in the current page. The OS fills the whole Scan table by calling *insert_PPN* with all the indices, correctly setting the Less and More indices. The entry with index 0 is the one that will be processed first.

insert_PFE fills the PFE entry. The call takes as operands the PPN of the candidate page, the Last Refill (L) flag, and the Ptr pointer set to point to the first entry in the Other Pages array. Recall that L is set to 1 if the scanning will complete after the current batch of pages in the Scan table is processed; otherwise, it is set to 0.

As soon as the PageForge hardware is triggered, it starts the comparison of the candidate page to the pages in the Other Pages array. Once all the pages have been compared (or a duplicate page has been found and the Duplicate (D) bit has been set), PageForge sets the Scanned (S) bit, and idles.

Typically, the OS fills the Scan table multiple times, until all the relevant pages have been compared to the candidate one. Specifically, the OS periodically calls *get_PFE_info* to get S and D. If S is set and D reset, it refills the Scan table with another batch of *insert_PPN* calls, and then calls function *update_PFE*. The latter sets L to 0 or 1, and Ptr to point to the first entry in the Other Pages array. PageForge then restarts the comparison.

The last batch of comparisons is the one that either started with L set to 1, or terminates with D set because a duplicate page was found. Either of these conditions triggers PageForge to complete the generation of the hash key. Consequently, when the OS calls *get_PFE_info* after either of these two conditions, it sees that the Hash Key Ready (H) bit is set, and reads the new hash key. If D is set, the value of Ptr tells which entry matched.

The last function in Table 7.1 is *update_ECC_offset*. It defines the page offsets that should be used to generate the ECC-based hash key. Such offsets are rarely changed. They are set after

profiling the workloads that typically run on the hardware platform. The goal is to attain a good hash key.

7.4 IMPLEMENTATION TRADE-OFFS

7.4.1 Discussion of Alternative Designs

State-of-the-art server architectures usually have 1–4 memory controllers, and interleave pages across memory controllers, channels, ranks, and banks to achieve higher memory-level parallelism. As a result, the decision of where to place PageForge on the chip and the total number of PageForge modules is not trivial. With respect to the placement of PageForge, we discuss the trade-offs between placing it inside the memory controller, and placing it outside the memory controller, directly connected to the on-chip interconnect. As for the number of PageForge modules, we discuss the trade-offs between having the PageForge module in one of the memory controllers, and having one PageForge module per memory controller.

The main benefit of placing PageForge outside of the memory controller, and directly attaching it to the on-chip network is that it leads to a more modular design. This is where accelerator modules are often placed in a processor chip. On the other hand, such approach would require all responses from the main memory to be placed on the on-chip interconnect, significantly increasing the on-chip traffic. By placing PageForge in the memory controller, we avoid the generation of unnecessary interconnect traffic when PageForge requests are serviced from the local memory module. Further, PageForge leverages the existing ECC engine for the hash key generation, which resides in the memory controller. Hence, this approach eliminates the hardware overhead of additional hash-key engines.

By increasing the number of PageForge modules in the system, we linearly increase the number of pages being scanned concurrently. As a result, the upside of having a PageForge module per memory controller is that we can improve the rate at which pages are scanned. However, this approach is accompanied by several drawbacks. First, the memory pressure increases linearly with the number of PageForge modules. Considering that the page-merging process is a very expensive background task, this approach would lead to increased memory access penalties to the workloads running on top of the VMs. In addition, while at a first glance we would expect that having one PageForge module in each memory controller will avoid cross memory controller communication, this is not the case. In practice, the page merging process searches pages across VM instances with multiple MB of memory allocated for each one. So, the common case is to compare pages that are spread out across the memory address space, and hence across memory controllers. Finally, with

multiple PageForge modules, we would have to co-ordinate the scanning process among them, which would increase complexity.

We choose a simple and efficient design, namely a single PageForge module for the system that is placed in one of the memory controllers (Figure 7.5). Memory pressure due to page comparison remains low, since we are only comparing two pages at a time. Moreover, PageForge requests that can be serviced from the local memory module cause no on-chip interconnect traffic.

7.4.2 Generality of PageForge

The division between hardware and software in the PageForge design, as outlined in Sections 7.3.1 and 7.3.6, makes PageForge general and flexible. Given a candidate page, the software decides which pages should be compared to it (i.e., those that the software places in the Scan table), and in what order they should be compared (i.e., starting from the entry pointed to by Ptr, and following the Less and More fields that the software has set). The software also decides how to use the page hash key generated by the hardware.

In turn, the hardware efficiently supports three operations that are widely used in same-page merging algorithms: pairwise page comparison, generation of the hash key in the background, and in-order traversal of the set of pages to compare.

In Section 7.3.4, we discussed how this framework can be used to support the KSM algorithm in hardware. However, we can apply it to other same-page merging algorithms. For example, consider an algorithm that wants to compare the candidate page to an arbitrary set of pages. In this case, the OS uses *insert_PPN* to insert these pages in the Scan table (possibly in multiple batches). For each page, the OS sets *both* the Less and More fields *to the same value*: that of the *subsequent entry* in the Scan table. In this way, all the pages are selected for comparison.

Alternatively, PageForge can support an algorithm that traverses a *graph* of pages. In this case, the OS needs to put the correct pages in the Scan table with the correct More/Less pointers. Finally, PageForge can also support algorithms that use the hash key of a page in different ways.

7.4.3 In-Order Cores or Uncacheable Accesses

We consider two design alternatives to PageForge. The first one is to run the page-merging algorithm in software in a simple in-order core, potentially shared with other background tasks. The second one is to run the page-merging algorithm in software on a regular core, but use cache-bypassing accesses.

There are several drawbacks to running a software algorithm on an in-order core. First, the core is farther than the PageForge module from the main memory, and from the ECC-generating

circuit. This means that main memory accesses and hash key generation are more costly. Second, a core consumes more power than PageForge. Section 7.6.4 estimates that, in 22nm, PageForge consumes only 0.037W on average, while an ARM-A9 core without an L2 cache consumes 0.37W on average. Third, the in-order core has to fully support the cache coherence protocol, while the PageForge module does not — which makes the system simpler. Finally, it is unrealistic to assume that the in-order core will be shared with other background tasks. Page deduplication is a sizable task that uses a large fraction of a core. In fact, it is typically pinned on an out-of-order core. Moreover, page deduplication will become heavier-weight as machines scale-up their memory size.

Running a software algorithm with cache-bypassing accesses can potentially reduce some of the performance overhead due to cache pollution. However, the CPU cycles required to run the software algorithm still remain. Further, non-cacheable requests occupy MSHR resources in the cache hierarchy, leading to resource pressure within the cache subsystem, and reducing the potential benefits of this approach. On the other hand, PageForge eliminates cache pollution, circumvents unnecessary MSHR pressure, and eliminates CPU cycles needed to run the page-deduplication algorithm.

7.5 EVALUATION METHODOLOGY

7.5.1 Modeled Architecture

We use cycle-level simulations to model a server architecture with a 10-core processor and 16GB of main memory. The architecture parameters are shown in Table 7.2. Each core is an out-of-order core with private L1 and L2 caches, and a shared L3 cache. A snoopy MESI protocol using a wide bus maintains coherence. We use Ubuntu Server 16.04 [320] with KVM [367] as the hypervisor, and create QEMU-KVM VM instances. Each VM is running a Ubuntu Cloud Image [378] based on the 16.04 distribution. During the experiments, each VM is pinned to a core. When the KSM process is enabled, all the cores of the system are included in its scheduling pool. Table 7.2 also shows the parameters of PageForge and KSM.

7.5.2 Modeling Infrastructure

We integrate the Simics [198] full-system simulator with the SST framework [199] and the DRAMSim2 [200] memory simulator. Additionally, we utilize Intel SAE [201] on top of Simics for OS instrumentation. Finally we use McPAT [379] for area and power estimations. We run

Processor Parameters	
Multicore chip; Frequency	10 single-issue out-of-order cores; 2GHz
L1 cache	32KB, 8 way, WB, 2 cycles Round Trip (RT), 16 MSHRs, 64B line
L2 cache	256KB, 8 way, WB, 6 cycles RT, 16 MSHRs, 64B line
L3 cache	32MB, 20 way, WB, shared, 20 cycles RT, 24 MSHRs per slice, 64B line
Network; Coherence	512b bus; Snoopy MESI at L3
Main-Memory Parameters	
Capacity; Channels	16GB; 2
Ranks/Channel; Banks/Rank	8; 8
Frequency; Data rate	1GHz; DDR
Host and Guest Parameters	
Host OS	Ubuntu Server 16.04
Guest OS	Ubuntu Cloud 16.04
Hypervisor	QEMU-KVM
# VMs; Core/VM; Mem/VM	10; 1; 512MB
PageForge and KSM Parameters	
sleep_millisecs = 5ms; *pages_to_scan* = 400; # PageForge modules = 1 # Scan table entries = 31 Other Pages + 1 PFE ECC hash key = 32bits; Scan table size $\approx$ 260B	

Table 7.2: Architectural parameters.

some applications from the Tailbench suite [380]. More specifically, we deploy a total of ten VMs, one for every core in the system, each running the same application in the harness configuration provided by the TailBench suite.

We assess our architecture with a total of five applications from TailBench. *Img-dnn* is a handwriting recognition application based on a deep neural network auto-encoder which covers the general area of image recognition services. *Masstree* represents in-memory key-value store services, and is driven by a modified version of the Yahoo Cloud Serving Benchmarks [187] with 50% get and 50% put queries. *Moses* is a statistical machine translation system similar to services like Google Translate. *Silo* is an in-memory transactional database that represents online transaction processing systems (OLTP) and is driven by TPC-C. Finally, *Sphinx* represents speech recognition systems like Apple Siri and Google Now. Table 7.3 lists the applications and the Queries Per Second (QPS) of the runs.

7.5.3 Configurations Evaluated

We target a cloud scenario where we have 10 homogeneous VMs running the same application on 10 cores. Each VM is pinned to one core. This scenario effectively describes a widely-used

Application	QPS
Img_Dnn	500
Masstree	500
Moses	100
Silo	2000
Sphinx	1

Table 7.3: Applications executed.

environment that exploits replication of the applications to achieve better load balancing and fault tolerance guarantees. We compare three configurations: *Baseline*, *KSM*, and *PageForge*. Baseline is a system where same-page merging is disabled. KSM is a system running RedHat's KSM software algorithm. PageForge is a system with our proposed architecture, using the same tree search algorithm as KSM. KSM and PageForge have the same sleep interval and number of pages to scan per interval, as shown in Table 7.2.

We evaluate three characteristics of same-page merging: memory savings, behavior of ECC-based hash keys, and execution overhead. For the memory savings experiments, we run the application multiple times, until the same-page merging algorithm reaches steady state. At that point, we measure the memory savings attained. Because KSM and PageForge achieve the same results, we only compare two configurations: one with page merging and one without.

For the experiments on the behavior of ECC-based hash keys, we also measure the behavior when the same-page merging algorithm reaches steady state. We compare PageForge's ECC-based hash keys to KSM's jhash-based hash keys. We report the fraction of hash key matches and mismatches.

For the execution overhead experiments, we measure the time it takes to complete a request of the application under each of the three configurations. We report two measures. The first one is the average of all the requests, also called *mean sojourn latency* or mean waiting time. The second one is the latency of the 95^{th} percentile, also called *tail latency*. We report these latencies normalized to those of Baseline. The measurements are taken after a warm-up period of 1 billion instructions executed.

7.6 EVALUATION

7.6.1 Memory Savings

Figure 7.7 measures the savings in memory allocation attained by same-page merging. For each application, the figure shows the number of physical pages allocated without page merging (left)

and with page merging (right). The bars are normalized to without page merging. Each bar is broken down into *Unmergeable*, *Mergeable Zero*, and *Mergeable Non-Zero* pages.

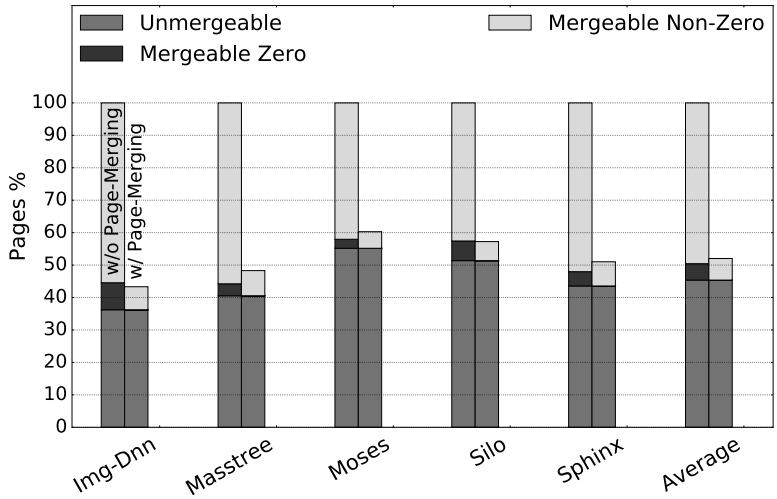


Figure 7.7: Memory allocation without and with page merging.

Unmergeable are the pages in the system that have unique values or whose data changes too frequently to be merged. As a result, they are not merged. On average, they account for 45% of the pages in the original system.

Mergeable Zero are pages whose data is zero. On average, they account for 5% of the pages in the original system. In current hypervisors, when the guest OS tries to allocate a page for the first time, a soft page-fault occurs, which invokes the hypervisor. The hypervisor picks a page, zeroes it out to avoid information leakage, and provides it to the guest OS. Most of the zero pages are eventually modified. However, at any time, a fraction of them remains, and is available for merging. When zero pages are merged, they are all merged into a single page (Figure 7.7).

Mergeable Non-Zero are non-zero pages that can be merged. On average, they account for 50% of the pages in the original system. The large majority of them are OS pages, as opposed to application pages. The figure shows that, on average, these pages are compressed to an equivalent of 6.6% of the pages in the original system.

Overall, page deduplication is very effective for these workloads. On average, it reduces the memory footprint by 48%. Intuitively, the memory savings attained imply that we can deploy about twice as many VMs as in a system without page deduplication, and use about the same amount of physical memory.

7.6.2 ECC Hash Key Characterization

We compare the effectiveness of PageForge's ECC-based hash keys to KSM's jhash-based hash keys. Recall that KSM's hash keys are much more expensive than PageForge's. Specifically, to generate a 32-bit key, KSM uses the jhash function on 1KB of the page contents. PageForge only needs to read four cache lines, which take 256B in our system. Hence, PageForge attains a 75% reduction in memory footprint to generate the hash key of a page.

Figure 7.8 compares the accuracy of the jhash-based and the ECC-based hash keys. For the latter, we use a SECDED encoding function based on the (72,64) Hamming code, which is a truncated version of the (127, 120) Hamming code with the addition of a parity bit. In the figure, we consider the hash key comparison performed by KSM and PageForge when the algorithm considers whether or not to search the unstable tree for a candidate page, and plot the fraction of key mismatches and of key matches.

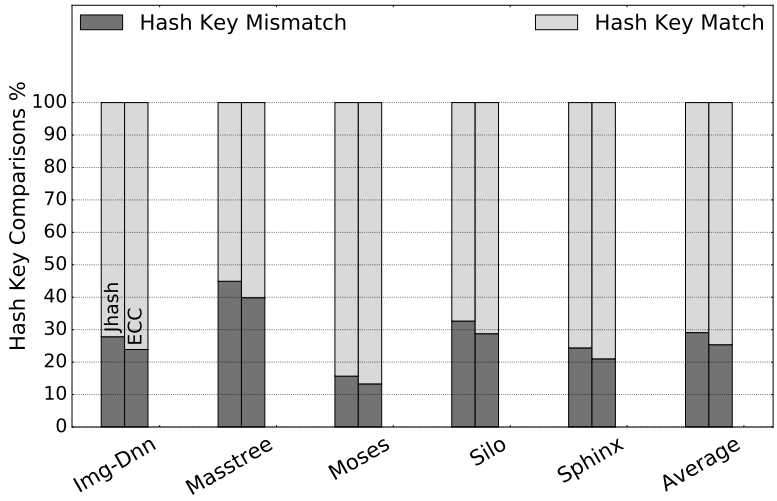


Figure 7.8: Outcome of hash key comparisons.

Recall that a mismatch of keys guarantees that the pages are different, while matches may be false positives (i.e., the pages may actually differ). The figure shows that ECC-based keys have slightly more matches than jhash-based ones, which correspond to false positives. On average, these additional false positives only account for 3.7% of the comparisons. This leads to the initiation of slightly more searches in the unstable tree with ECC-based keys. However, the benefits of ECC-based keys — a 75% reduction in memory footprint for key generation, the elimination of any dedicated hash engine, and the ability to overlap the search of the stable tree with the hash key generation — more than compensate for these extra searches.

7.6.3 Execution Overhead

Table 7.4 presents a detailed characterization of the KSM configuration. The second column shows the number of cycles taken by the execution of the KSM process, as a percentage of total cycles in a core. The column shows both the average across cores, and the maximum of all cores. On average, the KSM process utilizes the cores for 6.8% of the time. Recall that the Linux scheduler keeps migrating the KSM process across all the cores. However, the time that KSM spends in each core is different. As shown in the table, the core that runs the KSM process the most spends on average 33.4% of its cycles in it.

Applic.	KSM				Baseline
	Cycles (%)			L3 Miss Rate (%)	L3 Miss Rate (%)
	Avg, Max KSM Process / Total	Page Comp / KSM Process	Hash Key Gen / KSM Process		
Img_Dnn	6.1, 27	54	13	47.2	44.2
Masstree	6.4, 34	50	13	39.8	26.7
Moses	7.4, 31	49	23	34.5	30.8
Silo	7.1, 39	49	12	31.7	26.5
Sphinx	7.0, 36	57	13	42.9	41.0
Average	6.8, 33.4	51.8	14.8	39.2	33.8

Table 7.4: Characterization of the KSM configuration.

The next two columns show the percentage of cycles in the KSM process taken by page comparison and by hash key generation, respectively. On average, 52% of the time is spent on page comparisons. Page comparisons occur during the search of the stable and unstable trees. An additional 15% of the time is spent on hash key generation. Finally, the last two columns show the local miss rate of the shared L3 for two configurations, namely, KSM and Baseline. In the KSM configuration, the shared L3 gets affected by the migrating KSM process. We can see that, on average, the L3 miss rate increases by over 5%, and goes from 34% to 39%.

To see the execution overhead of page deduplication, Figure 7.9 compares the mean sojourn latency in the Baseline, KSM, and PageForge configurations. For each application, the figure shows a bar for each configuration, normalized to Baseline. Recall that the sojourn latency is the overall time that a request stays in the system, and includes both queuing and service time. In an application, each bar shows the geometric mean across the ten VMs of the system.

Baseline does not perform page deduplication and, therefore, is the shortest bar in all cases. KSM performs page deduplication in software. On average, its mean sojourn latency is 1.68 times longer than Baseline's. This overhead varies across applications, depending on the load of the application (measured in queries per second (QPS)), and on the time granularity of each query.

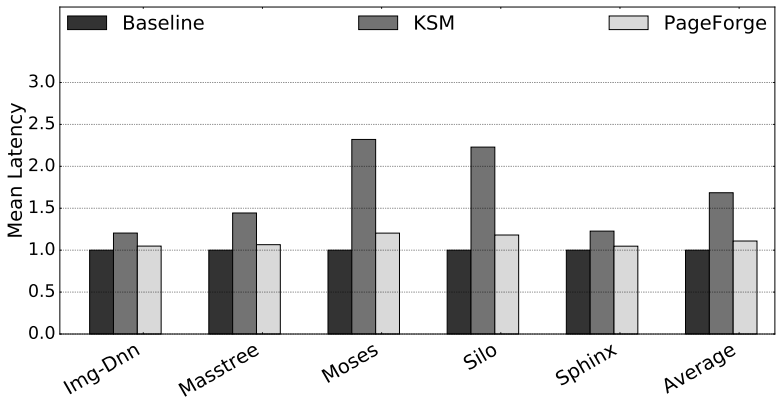


Figure 7.9: Mean sojourn latency normalized to Baseline.

Specifically, for a given application, the overhead caused by the KSM process is higher for higher QPS loads. Furthermore, applications with short queries are more affected by the KSM process than those with long queries. The latter can more easily tolerate the queueing induced by the KSM process before it migrates away. For example, Sphinx queries have second-level granularity, while Moses queries have millisecond-level granularity. As we can see from Figure 7.9, their relative latency increase is very different.

PageForge performs page deduplication in hardware and, as a result, its mean sojourn latency is much lower than KSM for all the applications. On average, PageForge has a mean sojourn latency that is only 10% higher than Baseline. This overhead is tolerable. Unlike KSM, PageForge masks the cost of page comparison and hash key generation by performing them in the memory controller. It avoids taking processor cycles and polluting the cache hierarchy.

We further explore the execution overhead by comparing the latency of the 95^{th} percentile (also called tail) latency of the three configurations. This is shown in Figure 7.10, which is organized as Figure 7.9. The tail latency better highlights the outliers in the system, when compared to the mean sojourn latency. For example, we see that Silo's tail latency in KSM is more than 5 times longer than in Baseline, while the mean sojourn latency is only twice longer. On average across applications, KSM increases the tail latency by 136% over Baseline, while PageForge increases it by only 11%.

Overall, PageForge effectively eliminates the performance overhead of page deduplication. This is because it offloads the core execution with special hardware, and avoids cache pollution. In addition, it performs hash key generation with very low overhead.

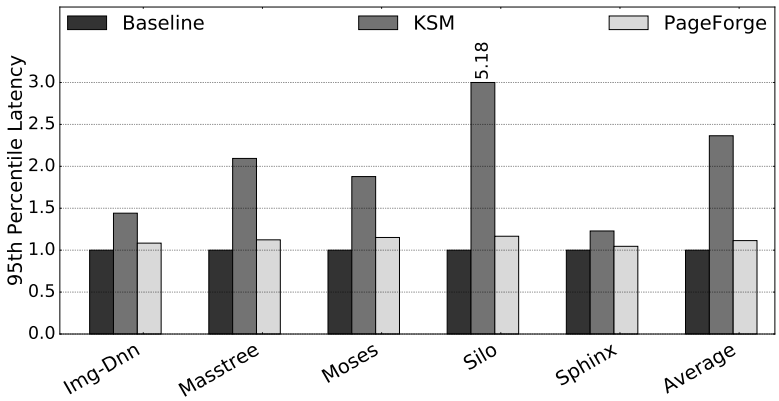


Figure 7.10: 95^{th} percentile latency normalized to Baseline.

7.6.4 PageForge Characterization

Memory Bandwidth Analysis

Figure 7.11 shows the memory bandwidth consumption of the KSM and PageForge configurations during the most memory-intensive phase of the page deduplication process in each application. The bandwidth consumption of the Baseline configuration during the same period of the application is also shown as a reference.

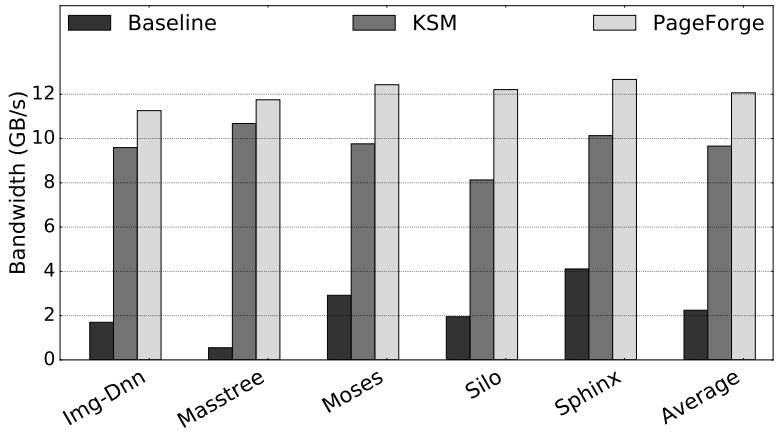


Figure 7.11: Memory bandwidth consumption in the most memory-intensive phase of page deduplication.

From the figure, we see that the bandwidth consumption of KSM and PageForge is higher than in Baseline. On average, Baseline consumes 2GB/s, while KSM and PageForge consume 10 and 12GB/s, respectively. The reason is that, during this active phase of page deduplication, the mem-

ory is heavily utilized with data being streamed for the page comparisons and hash key generation. Further, PageForge consumes more bandwidth than KSM. This is because deduplication in PageForge proceeds independently of the cores. Hence, its bandwidth consumption is additive to the one of the cores. Overall, however, even in these phases of the applications, the absolute memory bandwidth demands of PageForge are very tolerable.

Design Characteristics

Table 7.5 shows some design characteristics of PageForge. As indicated in Table 7.2, the Scan table stores a total of 31 Other Pages entries (which correspond to a root node plus four levels of the tree), and the PFE entry, for a total of 260B. We measure that, to process all the required entries in the table, PageForge takes on average 7,486 cycles. This time is mostly determined by the latency of the page comparison operations, which depend on the page contents. Hence, the table also shows the standard deviation across the applications, which is 1,296. The table also shows that the OS checks the Scan table every 12,000 cycles. Typically, the table has been fully processed by the time the OS checks.

Operation	Avg. Time (Cycles)	Applic. Standard Dev.
Processing the Scan table	7,486	1,296
OS checking	12,000	0
Unit	**Area (mm^2)**	**Power (W)**
Scan table	0.010	0.028
ALU	0.019	0.009
Total PageForge	0.029	0.037

Table 7.5: PageForge design characteristics.

The table also lists PageForge's area and power requirements. For the Scan table, we conservatively use a 512B cache-like structure. For the comparisons and other ALU operations, we use an ALU similar to those found in embedded processors. With 22 *nm* and high performance devices, our tools show that PageForge requires only 0.029 mm^2 and 0.037 W. In contrast, a server-grade architecture like the one presented in Table 7.2 requires a total of 138.6 mm^2 and has a TDP of 164 W. Compared to such architecture, PageForge adds negligible area and power overhead.

We also compare PageForge to a simple in-order core. Our tools show that a core similar to an ARM A9 with 32KB L1 data and instruction caches, and without an L2 cache, requires 0.77 mm^2 and has a TDP of 0.37 W, at 22nm and with low operating power devices. Compared to this very simple core, PageForge uses negligible area and requires an order of magnitude less power.

7.7 RELATED WORK

7.7.1 Hardware-Based Deduplication

There are two proposals for hardware-based deduplication. Both schemes perform merging of memory lines rather than pages as in PageForge. The first one by Tian et al. [364] merges cache lines that have the same contents in the Last Level Cache (LLC). The proposed cache design utilizes a hashing technique to detect identical cache lines, and merges them. The result is an increase of the LLC capacity and, hence, an improvement in application performance. However, deduplication does not propagate to the main memory and, therefore, the scheme does not increase the memory capacity of the system. Hence, this scheme is orthogonal to PageForge and can be used in conjunction with it.

The second proposal is HICAMP [365, 381], a complete redesign of the memory system so that each memory line stores unique data. In HICAMP, the memory is organized in a content addressable manner, where each line has immutable data over its lifetime. The memory is organized in segments, where each segment is a Directed Acyclic Graph (DAG). To see if the memory contains a certain value, the processor hashes the value and performs a memory lookup. The lookup can return multiple lines, which are then compared against the value to eliminate false positives. The authors introduce a programming model similar to an object oriented model.

HICAMP requires a complete redesign of the memory, introduces a complex memory access scheme, and needs a new programming model. PageForge focuses on identifying identical pages and merging them. PageForge's hardware can be easily integrated in current systems. Moreover, it requires little software changes, and no special programming model.

7.7.2 Software-Based Deduplication

Most same-page merging proposals and commercial products are software-based. One of the first implementations, Transparent Page Sharing (TPS), originated from the Disco research project [358]. A limitation of TPS is that it relies on modifications to the guest OS to enable the tracking of identical pages. This limitation was later addressed by VMware's ESX Server [353]. ESX enables the hypervisor to transparently track and merge pages. It generates a hash key for each page and, only if the keys of two pages are the same, it compares the pages. A similar approach is used by IBM's Active Memory Deduplication [362], which generates a signature for each physical page. If two signatures are different, there is no need to compare the pages.

The Difference Engine [352] is the first work that proposes sub-page level sharing. In this case, pages are broken down into smaller pieces in order to enable finer-grain page sharing. In addition,

this work extends page sharing with page compression to achieve even greater memory savings. Memory Buddies [382] proposes the intelligent collocation of multiple VMs in datacenters in order to optimize the memory sharing opportunities. Furthermore, Despande et al. [383] present a deduplication-based approach to perform group migration of co-located VMs.

The Satori system [360] introduces a sharing technique that monitors the read operations from disk to identify identical regions. The authors argue that additional memory sharing opportunities exist within the system, but only last a few seconds. They conclude that the periodic scanning of the memory space is not sufficient to exploit such sharing opportunities. PageForge potentially enables the exploitation of such sharing opportunities since it can perform aggressive memory scanning at a fraction of the overhead of software approaches.

RedHat's open-source version of same-page merging is Kernel Same-page Merging (KSM) [355]. KSM is currently distributed along with the Linux kernel. It targets both KVM-based VMs and regular applications. We describe it in Section 7.2.1. A recent empirical study [356] shows that KSM can achieve memory savings of up to 50%. Since KSM is a state-of-the-art open-source algorithm, we compare PageForge to it. However, PageForge is not limited to supporting KSM.

UKSM [363] is a modification of KSM available as an independent kernel patch. In UKSM, the user defines the amount of CPU utilization that is assigned to same-page merging, while in KSM, the user defines the *sleep_millisecs* between runs of the same-page merging algorithm, and the *pages_to_scan* in each run. In addition, UKSM performs a whole-system memory scan, instead of leveraging the *madvise* system call [371] and *MADV_MERGEABLE* flag that KSM uses. While this approach enables UKSM to scan every anonymous page in the system, it eliminates the tuning opportunities provided through the *madvise* interface, and does not allow a cloud provider to choose which VMs should be prevented from performing same-page merging. UKSM also uses a different hash generation algorithm.

Overall, PageForge takes inspiration from an open-source state-of-the-art software algorithm (i.e., KSM), and implements a general and flexible hardware-based design. It is the first hardware-based design for same-page merging that is effective and can be easily integrated in current systems.

7.7.3 Other Related Work

Previous work in virtual memory management has focused on large pages [384, 385] and their implications on memory consolidation in virtualized environments [386, 387]. These approaches can be transparently integrated with PageForge to boost the page sharing opportunities in the presence of large pages.

DRAM devices are usually protected through 8/16-bits of ECC for every 64/128 data bits. Pre-

vious work [373, 374, 375, 376, 388] explores the trade-offs and methodologies required to efficiently provide single error correction and double error detection.

Previous work proposes modifications to the memory controller to attain better performance (e.g., [389, 390, 391]). Some work focuses on optimizing request scheduling policies [392, 393, 394, 395]. These optimizations are orthogonal to our goal, and can be employed together with PageForge.

7.8 CONCLUSION

This chapter presented PageForge, a lightweight hardware mechanism to perform same-page merging in the memory controller. To the best of our knowledge, this is the first solution for hardware-assisted same-page merging that is general, effective, and requires modest hardware modifications and hypervisor involvement. We evaluated PageForge with simulations of a 10-core processor and a VM on each core, running a set of applications from the TailBench suite. When compared to RedHat's KSM, a state-of-the-art software implementation of page merging, PageForge achieved identical memory savings while substantially reducing the overhead. Compared to a system without same-page merging, PageForge reduced the memory footprint by an average of 48%, enabling the deployment of twice as many VMs for the same physical memory. Importantly, it kept the average latency overhead to 10%, and the 95^{th} percentile tail latency to 11%. In contrast, in KSM, these overheads were 68% and 136%, respectively.

Chapter 8: Conclusions

The unprecedented growth in data and users, together with the emergence of critical security vulnerabilities have highlighted the limitations of the current computing stack. To remedy these limitations, this thesis studied the security, performance, and scalability of OS and hardware abstractions and interfaces.

This thesis started by presenting Microarchitectural Replay Attacks (MRAs), a novel class of attacks based around the premise that a single dynamic instruction may execute more than once on modern out-of-order machines. Attackers can leverage this capability to create new privacy- and integrity-breaking attacks that are able to de-noise nearly arbitrary microarchitecture side-channels. To defend against MRAs, this thesis then introduced Jamais Vu, the first defense against MRAs. Jamais Vu provides different trade-offs between execution overhead, security, and implementation complexity by exploring hardware-only and compiler and OS-assisted solutions. In addition, this thesis presented Draco, a scalable system call defense mechanism that protects the OS in modern multi-tenant cloud environments. Draco consists of a software kernel component or hardware extensions that reduce the attack surface of the kernel. Hardware Draco has nearly no overhead over an insecure system.

To improve system performance, this thesis proposed Elastic Cuckoo Page Tables, a radical page table design that banishes the sequential pointer chasing in the address translation walk of existing virtual memory designs and replaces it with fully parallel lookups. Furthermore, the thesis introduced Elastic Cuckoo Hashing, a novel extension of d-ary cuckoo hashing that requires only d memory accesses to lookup an element during gradual rehashing, instead of $2 \times d$ in previous algorithms. Elastic Cuckoo page tables are shown to outperform existing Radix pages tables.

To improve scalability, this thesis presented BabelFish, which is a set of OS and hardware extensions that enable an unlimited number of processes, containers, and functions to be bundled together in groups and share translation resources across the stack. The result is a significant reduction in process management overheads. In addition, to further improve scalability, this thesis introduced PageForge, a novel content-aware page sharing architecture and hypervisor extensions. PageForge includes a set of hardware primitives that enable the deployment of twice as many VMs for the same physical memory, while at the same time eliminating the performance overhead of software-only solutions.

The combination of the proposed techniques improves the security, performance, and scalability of next-generation cloud systems.

Chapter 9: Future Work

My vision for future research is to radically redesign abstractions and interfaces between OS and hardware for computer systems that are massively shared and have strict security guarantees. This thesis is only the first step towards this direction. Future work will be interdisciplinary, and involve collaborating with researchers in algorithms, programming languages, compilers, systems, and hardware. In the following, I give a few examples of short- and long-term projects.

9.1 SHORT TERM WORK

Nested Elastic Cuckoo Page Tables Virtual memory mechanisms have recently been under scrutiny due to the increased overhead of address translation caused by the inevitable main memory capacity increase. In particular, in virtualized environments, nested address translation based on radix-page tables may require up to twenty-four sequential memory accesses. In this section, we briefly present the first steps towards a fundamentally different approach to address this problem by building on Elastic Cuckoo Page Tables. Our design, called Nested Elastic Cuckoo Page Tables, first replaces the sequential translation lookups of radix-page tables with fully parallel walks within the guest and host page tables. Then, we further accelerate the address translation process by optimizing the guest page table allocation and introduce novel hardware MMU caches that bound the parallelism during lookups.

Directly applying elastic cuckoo hash tables to virtualized environments is a difficult task. This is because a nested page walk can significantly magnify the number of parallel memory accesses required to perform a translation. To address this limitation, we are working on a design that leverages two sets of cuckoo walk tables that provide page size and way information about the guest and host page tables, respectively. Furthermore, we propose a page table allocation technique deployed at the hypervisor layer that enforces the usage of 4KB pages for the backing of page tables. Finally, we introduce a nested Cuckoo Walk cache that efficiently caches guest and host Cuckoo Walk Table entries.

9.2 LONG TERM WORK

Scalable Operating System Virtualization and Security. As cloud computing shifts toward ever finer-grain computing and strict security requirements, we will need to rethink (a) how security privileges are distributed among the computing stack, and (b) what kernel functionalities to provide for massively parallel, short-lived task execution. More to the point, the key question

is how to design an OS for serverless computing and functions as a service (FaaS). To reach this goal, we need to explore new security and isolation primitives, as well as new scheduling mechanisms at the OS and hardware levels. Furthermore, we need to rethink how we perform resource allocation and sharing across tenants. We should build lightweight sandboxing systems that support extremely efficient transitions between user and kernel contexts, while guaranteeing isolation between them. Finally, we need to conceive new abstractions for users to define the performance and security requirements of their applications.

Side-channels in Multi-tenant Environments. My previous work has focused on *noiseless* side-channel attacks and defenses. In the future, my goal is to explore side-channels across the stack at the OS and hardware levels. One such example is Remote Direct Memory Access (RDMA). Primitives such as RDMA allow a machine to access the memory of another machine while bypassing the local and remote OS. While such approach enables low-latency communication, it introduces a new attack surface with potentially critical consequences.

Another example is enclave computing, such as Intel's SGX. Secure enclaves aim to protect user applications from privileged software such as the OS. As my work has highlighted, current designs are fundamentally broken against side-channels. My work will build novel mechanisms across the stack that seal side-channels and provide strong isolation without sacrificing performance.

Machine Learning to Improve Operating System Functionality. During my industrial experience, I built systems for machine learning (ML) and explored how ML can be used to strengthen user privacy. It is clear that the predictive capabilities of ML will play an important role towards highly adaptive and responsive systems. A promising area is to research how to leverage ML to augment common OS operations such as process creation, memory management, and other core functionalities.

Appendix A: Other Work

In this Section I briefly describe other research work that was conducted in parallel to this thesis.

Machine Learning as a Service. During my internship at VMware, I built *DoppioML* [396], a novel privacy-aware machine learning as a service (MLaaS) framework. DoppioML cuts the computational dataflow graph of a neural network into two subgraphs: one to run in the edge device, and the other in the cloud. For each subgraph, DoppioML characterizes the capability of an attacker to reconstruct the original data through adversarial neural networks. The goal of DoppioML is to identify the best cut that minimize the communication between the edge and the cloud, and the probability of a successful attack. Together with VMware researchers, I patented DoppioML [396] and implemented it on VMware's datacenters.

Non-Volatile Memory Systems. A key difficulty in hybrid memory systems that combine volatile and non-volatile memory segments is to identify which pages to swap between the two memory segments and when. The goal is to minimize the accesses to the non-volatile memory in the critical path. To address this challenge, we proposed *PageSeer* [397], a hardware mechanism that, as soon as a request suffers a TLB miss, it prefetches the corresponding page from the non-volatile to the volatile memory.

Speculative Execution Defenses. Speculative execution attacks are a major security threat in modern processors. To defend against these attacks, we proposed *InvisiSpec* [67], a hardware mechanism that makes speculative execution invisible in the cache hierarchy. In *InvisiSpec*, speculative loads bring data into a speculative buffer without modifying the cache hierarchy. When the load is deemed safe, it is made visible.

3D Die-stacked Processors. 3D-stacking of processor and memory dies is an attractive technology to increase the integration of computing systems. In such architectures, processor and memory have their separate power delivery networks (PDNs). Each PDN is provisioned for the worst-case power requirements. However, typically, a program executes a compute-intensive section or a memory-intensive section, but not both at the same time. To exploit this behavior, I proposed *Snatch* [398], a power management and delivery mechanism which dynamically and opportunistically diverts some power between the two PDNs on demand. The proposed power-sharing approach results in 3D architectures with significantly improved performance or, alternatively, reduced packaging costs.

Novel Processor Technologies. Since CMOS technology scaling has slowed down, we need new technologies that deliver improved performance and energy efficiency. One approach is to combine CMOS devices with beyond-CMOS ones that have better energy characteristics than CMOS. In *HetCore* [399], we explored using CMOS and T-FET devices in the same die. HetCore presents a methodology of how architects can design a CPU and a GPU that integrate both CMOS and T-FET devices by identifying which units are best suitable to each technology.

References

[1] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019.

[2] D. Skarlatos, Z. N. Zhao, R. Paccagnella, C. Fletcher, and J. Torrellas, "Jamais Vu: Thwarting Microarchitectural Replay Attacks; Submitted for publication," 2021.

[3] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and Operating System Support for System Call Security," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, October 2020.

[4] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, 2020.

[5] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "BabelFish: Fusing Address Translations for Containers," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[6] D. Skarlatos, N. S. Kim, and J. Torrellas, "PageForge: A Near-memory Content-aware Page-merging Architecture," in *the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[7] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017.

[8] Intel, "Intel Software Guard Extensions Programming Reference," https://software.intel.com/sites/default/files/329298-001.pdf, 2013.

[9] Intel, "Intel Software Guard Extensions Software Development Kit," https://software.intel.com/en-us/sgx-sdk, 2013.

[10] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptology ePrint Archive, Report 2016/204, 2016, https://eprint.iacr.org/2016/204.

[11] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[12] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with SGX enclaves," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[13] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 9:1–9:14.

[14] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014.

[15] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless calls made practical in Intel SGX," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, ser. SysTEX '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3268935.3268942 pp. 22–27.

[16] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017.

[17] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

[18] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel SGX," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[19] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 279–296.

[20] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017.

[21] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A data oblivious filesystem for intel SGX," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[22] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, 2017.

[23] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*. Springer Berlin Heidelberg, 2006.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622.

[25] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019.

[26] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014.

[27] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: ACM, 2007, pp. 312–320.

[28] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018.

[29] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*, May 2015.

[30] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl pp. 565–581.

[31] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *CoRR*, vol. abs/1711.08002, 2017. [Online]. Available: http://arxiv.org/abs/1711.08002

[32] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," Cryptology ePrint Archive, Report 2018/1060, 2018, https://eprint.iacr.org/2018/1060.

[33] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras pp. 955–972.

[34] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," *CoRR*, vol. abs/1702.07521, 2017. [Online]. Available: http://arxiv.org/abs/1702.07521

[35] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017.

[36] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, May 2015.

[37] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," *CoRR*, vol. abs/1703.06986, 2017. [Online]. Available: http://arxiv.org/abs/1703.06986

[38] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 171–191, May 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/879

[39] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic pp. 1289–1306.

[40] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre attacks: Leaking enclave secrets via speculative execution," *CoRR*, vol. abs/1802.09085, 2018. [Online]. Available: http://arxiv.org/abs/1802.09085

[41] Intel, "64 and IA-32 Architectures Software Developer's Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf, Feb. 2019.

[42] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[43] M. Johnson, *Superscalar microprocessor design*. Prentice Hall Englewood Cliffs, New Jersey, 1991, vol. 77.

[44] Intel, "Intel Trusted Execution Technology," http://www.intel.com/technology/security, 2007.

[45] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *ARM white paper*, 2004.

[46] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065915 pp. 2:1–2:6.

[47] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *2011 IEEE Symposium on Security and Privacy*, May 2011.

[48] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 191–205.

[49] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 299–312.

[50] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. ACM, 2017.

[51] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Dec 2006, pp. 473–482.

[52] O. Acıiçmez and J. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, Sept 2007, pp. 80–91.

[53] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195686 pp. 40:1–40:13.

[54] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'07. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 225–242.

[55] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[56] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065915 pp. 2:1–2:6.

[57] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel pp. 299–312.

[58] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS'19*, https://eprint.iacr.org/2018/808.

[59] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. Springer-Verlag, 1999, pp. 388–397.

[60] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017.

[61] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[62] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[63] OpenSSL, "Open source cryptography and SSL/TLS toolkit," https://www.openssl.org, 2019.

[64] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." Network and Distributed System Security Symposium 2017 (NDSS'17), February 2017.

[65] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," CoRR'18.

[66] E. Nascimento, L. Chmielewski, D. Oswald, and P. Schwabe, "Attacking embedded ecc implementations through cmov side channels," Cryptology ePrint Archive, Report 2016/923, 2016, https://eprint.iacr.org/2016/923.

[67] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[68] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," *CoRR*, vol. abs/1806.05179, 2018.

[69] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[70] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897885 pp. 317–328.

[71] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018.

[72] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[73] C. Percival, "Cache Missing for Fun and Profit," 2005.

[74] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[75] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[76] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[77] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," 2019.

[78] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[79] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *CCS'12*.

[80] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. of the USENIX Security Symposium (USENIX)*, 2015.

[81] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519," in *CCS'17*.

[82] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision L3 cache attack using intel TSX," in *Proc. of the USENIX Security Symposium (USENIX)*, 2017.

[83] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on the last level cache," in *Proc. of the Design Automation Conference (DAC)*, 2016.

[84] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[85] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Proc. of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2007.

[86] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," Cryptology ePrint Archive, Report 2016/613, 2016, https://eprint.iacr.org/2016/613.

[87] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *Proc. of the International Workshop on Selected Areas in Cryptography (SAC)*, 2006.

[88] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[89] M. Dubois, M. Annavaram, and P. Stenstrm, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[90] A. Ros and S. Kaxiras, "The superfluous load queue," in *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[91] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[92] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2020.

[93] Intel, "Intel VTune profiler," https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html, accessed on 20.08.2020.

[94] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A practical attack framework for precise enclave execution control," in *Proc. of the Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[95] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, July 1970.

[96] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, 2000.

[97] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[98] Intel, "64 and IA-32 Architectures Software Developer's Manual," 2019.

[99] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, no. 2, pp. 1–7, 2011.

[100] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, June 2017.

[101] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Proc. of the Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018.

[102] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[103] "UNIX-like reverse engineering framework and command-line toolset," https://github.com/radareorg/radare2.

[104] "C++ Bloom Filter Library," https://github.com/ArashPartow/bloom.

[105] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: closing controlled channels with self-paging enclaves," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2020.

[106] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *Proc. of the USENIX Annual Technical Conference (ATC)*, 2018.

[107] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[108] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments," in *Proc. of the USENIX Security Symposium (USENIX)*, 2020.

[109] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA'16*.

[110] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proc. of the USENIX Security Symposium (USENIX Security)*, 2012.

[111] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[112] D. Townley and D. Ponomarev, "Smt-cop: Defeating side-channel attacks on execution units in smt processors," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[113] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[114] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Security'17*.

[115] S. Deng, W. Xiong, and J. Szefer, "Secure TLBs," in *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2019.

[116] "CVE-2017-5123," https://nvd.nist.gov/vuln/detail/CVE-2017-5123.

[117] "CVE-2016-0728," https://nvd.nist.gov/vuln/detail/CVE-2016-0728.

[118] "CVE-2014-3153," https://nvd.nist.gov/vuln/detail/CVE-2014-3153.

[119] "CVE-2017-18344," https://nvd.nist.gov/vuln/detail/CVE-2017-18344.

[120] "CVE-2018-18281," https://nvd.nist.gov/vuln/detail/CVE-2018-18281.

[121] "CVE-2015-3290," https://nvd.nist.gov/vuln/detail/CVE-2015-3290.

[122] "CVE-2016-5195," https://nvd.nist.gov/vuln/detail/CVE-2016-5195.

[123] "CVE-2014-9529," https://nvd.nist.gov/vuln/detail/CVE-2014-9529.

[124] "CVE-2014-4699," https://nvd.nist.gov/vuln/detail/CVE-2014-4699.

[125] "CVE-2016-2383," https://nvd.nist.gov/vuln/detail/CVE-2016-2383.

[126] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, Sep. 2015.

[127] "OpenBSD Pledge," https://man.openbsd.org/pledge.

[128] "OpenBSD Tame," https://man.openbsd.org/OpenBSD-5.8/tame.2.

[129] "PROCESS MITIGATION SYSTEM CALL DISABLE POLICY structure," https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy.

[130] P. Lawrence, "Seccomp filter in Android O," https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html, July 2017.

[131] J. Corbet, "Systemd Gets Seccomp Filter Support," https://lwn.net/Articles/507067/.

[132] Docker, "Seccomp security profiles for Docker," https://docs.docker.com/engine/security/seccomp/.

[133] Ubuntu, "LXD," https://help.ubuntu.com/lts/serverguide/lxd.html#lxd-seccomp.

[134] Google, "gVisor: Container Runtime Sandbox," https://github.com/google/gvisor/blob/master/runsc/boot/filter/config.go.

[135] AWS, "Firecracker Design," https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md.

[136] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, 2020, pp. 419–434.

[137] Rtk Documentation, "Seccomp Isolators Guide," https://coreos.com/rkt/docs/latest/seccomp-guide.html.

[138] Singularity, "Security Options in Singularity," https://sylabs.io/guides/3.0/user-guide/security_options.html.

[139] Kubernetes Documentation, "Configure a Security Context for a Pod or Container," https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, July 2019.

[140] Mesos, "Linux Seccomp Support in Mesos Containerizer," http://mesos.apache.org/documentation/latest/isolators/linux-seccomp/.

[141] "Sandboxed API," https://github.com/google/sandboxed-api, 2018.

[142] T. Hromatka, "Seccomp and Libseccomp performance improvements," in *Linux Plumbers Conference 2018*, Vancouver, BC, Canada, Nov. 2018.

[143] T. Hromatka, "Using a cBPF Binary Tree in Libseccomp to Improve Performance," in *Linux Plumbers Conference 2018*, Vancouver, BC, Canada, Nov. 2018.

[144] M. Kerrisk, "Using seccomp to Limit the Kernel Attack Surface," in *Linux Plumbers Conference 2015*, Seattle, WA, USA, Aug. 2015.

[145] A. Grattafiori, "Understanding and Hardening Linux Containers," NCC Group, Tech. Rep., June 2016.

[146] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, June 2013.

[147] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," in *Proceedings of the 8th European Symposium on Research in Computer Security*, Gjøvik, Norway, Oct. 2003.

[148] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[149] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 7, no. 4, pp. 381–395, Oct. 2010.

[150] Linux, "Secure Computing with filters," https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[151] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, San Diego, California, Feb. 2003.

[152] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," in *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)*, Boston, MA, USA, Aug. 2007.

[153] AWS, "Firecracker microVMs," https://github.com/firecracker-microvm/firecracker/blob/master/vmm/src/default_syscalls/filters.rs.

[154] Moby Project, "A collaborative project for the container ecosystem to assemble container-based systems," https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

[155] Singularity, "Singularity: Application Containers for Linux," https://github.com/sylabs/singularity/blob/master/etc/seccomp-profiles/default.json.

[156] Subgraph, "Repository of maintained OZ profiles and seccomp filters," https://github.com/subgraph/subgraph-oz-profiles.

[157] "QEMU," https://github.com/qemu/qemu/blob/master/qemu-seccomp.c.

[158] Julien Tinnes, "A safer playground for your Linux and Chrome OS renderers," https://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, Nov. 2012.

[159] Julien Tinnes, "Introducing Chrome's next-generation Linux sandbox," https://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, Sep. 2012.

[160] "OZ: a sandboxing system targeting everyday workstation applications," https://github.com/subgraph/oz/blob/master/oz-seccomp/tracer.go, 2018.

[161] "binctr: Fully static, unprivileged, self-contained, containers as executable binaries," https://github.com/genuinetools/binctr.

[162] "go2seccomp: Generate seccomp profiles from go binaries," https://github.com/xfernando/go2seccomp.

[163] oci-seccomp-bpf-hook, "OCI hook to trace syscalls and generate a seccomp profile," https://github.com/containers/oci-seccomp-bpf-hook.

[164] "Kubernetes Seccomp Operator," https://github.com/kubernetes-sigs/seccomp-operator.

[165] S. Kerner, "The future of Docker containers," https://lwn.net/Articles/788282/.

[166] Red Hat, "Configuring OpenShift Container Platform for a Custom Seccomp Profile," https://docs.openshift.com/container-platform/3.5/admin_guide/seccomp.html#seccomp-configuring-openshift-with-custom-seccomp.

[167] Rtk Documentation, "Overriding Seccomp Filters," https://coreos.com/rkt/docs/latest/seccomp-guide.html#overriding-seccomp-filters.

[168] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, "Tailored Application-specific System Call Tables," The Pennsylvania State University, Tech. Rep., 2014.

[169] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, London, United Kingdom, Apr. 2016.

[170] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining Sandboxes for Linux Containers," in *Proceedings of 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*, Tokyo, Japan, Mar. 2017.

[171] Moby Project, "An open framework to assemble specialized container systems without reinventing the wheel," https://mobyproject.org/, 2019.

[172] K. McAllister, "Attacking hardened Linux systems with kernel JIT spraying," https://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html, Nov. 2012.

[173] E. Reshetova, F. Bonazzi, and N. Asokan, "Randomization can't stop BPF JIT spray," https://www.blackhat.com/docs/eu-16/materials/eu-16-Reshetova-Randomization-Can't-Stop-BPF-JIT-Spray-wp.pdf, Nov. 2016.

[174] R. Gawlik and T. Holz, "SoK: Make JIT-Spray Great Again," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT'18)*, 2018.

[175] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[176] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, Feb 2005.

[177] ECMA International, "Standard ECMA-182," https://www.ecma-international.org/publications/standards/Ecma-182.htm.

[178] Microsoft, "Windows System Call Disable Policy," https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy.

[179] Akamai, "Akamai Online Retail Performance Report: Milliseconds Are Critical," https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[180] Google, "Find out how you stack up to new industry benchmarks for mobile page speed," https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/.

[181] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing : Securing speculative execution via microcode customization," in *ASPLOS'19*.

[182] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, 2019.

[183] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "Undo" Approach to Safe Speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[184] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *ASPLOS'19*, 2019.

[185] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[186] Elastic, "Elasticsearch: A Distributed RESTful Search Engine," https://github.com/elastic/elasticsearch, 2019.

[187] Yahoo!, "Yahoo! Cloud Serving Benchmark," https://github.com/brianfrankcooper/YCSB, 2019.

[188] Apache, "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[189] SysBench, "A modular, cross-platform and multi-threaded benchmark tool." http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html, 2019.

[190] Redis, "How fast is Redis?" https://redis.io/topics/benchmarks, 2019.

[191] OpenFaaS, "OpenFaaS Sample Functions," https://github.com/openfaas/faas/tree/master/sample-functions.

[192] HPCC, "HPC Challenge Benchmark," https://icl.utk.edu/hpcc/.

[193] UnixBench, "BYTE UNIX benchmark suite," https://github.com/kdlucas/byte-unixbench.

[194] IPC-Bench, "Benchmarks for inter-process-communication techniques," https://github.com/goldsborough/ipc-bench.

[195] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-Phase Execution of Application Containers," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*, Bonn, Germany, 2017.

[196] Heroku Engineering Blog, "Applying Seccomp Filters at Runtime for Go Binaries," https://blog.heroku.com/applying-seccomp-filters-on-go-binaries, Aug. 2018.

[197] Adobe Security, "Better Security Hygiene for Containers," https://blogs.adobe.com/security/2018/08/better-security-hygiene-for-containers.html, 2018.

[198] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, 2002.

[199] A. F. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, "The Structural Simulation Toolkit," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'10)*, 2006.

[200] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[201] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, 2016.

[202] Synopsys, "Design compiler," https://www.synopsys.com, 2019.

[203] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," in *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, July 1996.

[204] D. A. Wagner, "Janus: an Approach for Confinement of Untrusted Applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-99-1056, 2016.

[205] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, Aug. 2003.

[206] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS'00)*, San Diego, California, USA, Feb. 2000.

[207] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, San Diego, California, Feb. 2004.

[208] A. Acharya and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, USA, Aug. 2000.

[209] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: Confined Execution Environment for Internet Computations," The University of California, Santa Barbara, Tech. Rep., 1999.

[210] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," in *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, Aug. 2002.

[211] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[212] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, "Protecting Against Unexpected System Calls," in *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, July 2005.

[213] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitara, "ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code," IBM Research Division, T.J. Watson Research Center, Tech. Rep. RC 20742 (2/20/97), Feb. 1997.

[214] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'98)*, New Orleans, Louisiana, June 1998.

[215] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing Security Checks on Commodity Hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, WA, USA, Mar. 2008.

[216] Y. Oyama, K. Onoue, and A. Yonezawa, "Speculative Security Checks in Sandboxing Systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, Apr. 2005.

[217] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, Beijing, China, June 2008.

[218] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014.

[219] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[220] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera, "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture," *IEEE Micro*, vol. 36, no. 5, pp. 38–49, Jan. 2016.

[221] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: Protecting Software with Code-Centric Memory Domains," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014.

[222] L. Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero, "Direct Inter-Process Communication (DIPC): Repurposing the CODOMs Architecture to Accelerate IPC," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, 2017.

[223] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 487–502, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694383

[224] K. Sinha and S. Sethumadhavan, "Practical Memory Safety with REST," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018.

[225] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical Byte-Granular Memory Blacklisting Using Califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[226] AMD, "Architecture Programmer's Manual (Volume 2)," https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

[227] J. Corbet, "Four-level page tables," https://lwn.net/Articles/117749/, 2005.

[228] Intel, "5-Level Paging and 5-Level EPT (White Paper)," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2015.

[229] Intel, "Sunny Cove Microarchitecture," https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, 2018.

[230] J. Corbet, "Five-level page tables," https://lwn.net/Articles/717293/, 2017.

[231] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, 2010.

[232] 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'17)*, 2017.

[233] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.

[234] 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-45)*, 2012.

[235] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[236] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010.

[237] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*, 2011.

[238] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, 2015.

[239] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, 2013.

[240] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[241] F. T. Hady, A. P. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," *Proceedings of the IEEE*, vol. 105, no. 9, 2017.

[242] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*, 1st ed. Morgan & Claypool Publishers, 2011.

[243] M. Talluri, M. D. Hill, and Y. A. Khalidi, "A New Page Table for 64-bit Address Spaces," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.

[244] B. L. Jacob and T. N. Mudge, "A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, 1998.

[245] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, 1993.

[246] C. Dougan, P. Mackerras, and V. Yodaiken, "Optimizing the Idle Task and Other MMU Tricks," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999.

[247] J. Jann, P. Mackerras, J. Ludden, M. Gschwind, W. Ouren, S. Jacobs, B. F. Veale, and D. Edelsohn, "IBM POWER9 system software," *IBM Journal of Research and Development*, vol. 62, no. 4/5, June 2018.

[248] Intel, "Itanium Architecture Software Developer's Manual (Volume 2)," https://www.intel.com/content/www/us/en/products/docs/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html, 2010.

[249] I. Yaniv and D. Tsafrir, "Hash, Don't Cache (the Page Table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*, 2016.

[250] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008.

[251] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*, 2012.

[252] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, "Itanium — A System Implementor's Tale," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, 2005.

[253] S. Eranian and D. Mosberger, "The Linux/ia64 Project: Kernel Design and Status Update," HP Laboratories Palo Alto, Tech. Rep. HPL-2000-85, 2000.

[254] J.-P. Aumasson, W. Meier, R. Phan, and L. Henzen, *The Hash Function BLAKE*. Springer, 2014.

[255] IBM, "PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors," https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf, 2005.

[256] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, Feb. 2005.

[257] Linux Kernel, "Page table types," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/pgtable_types.h?h=v4.19.1, 2019.

[258] The Linux Kernel Archives, "Transparent hugepage support," https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2019.

[259] A. Awad, S. D. Hammond, G. R. Voskuilen, and R. J. Hoekstra, "Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework," Sandia National Laboratories, Tech. Rep. SAND2017-0002, 2017.

[260] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, 2015.

[261] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, 2005.

[262] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) Benchmark Suite," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, 2006.

[263] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[264] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[265] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, 2014.

[266] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*, 2014.

[267] J. B. Chen, A. Borg, and N. P. Jouppi, "A Simulation Based Study of TLB Performance," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, 1992.

[268] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.

[269] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[270] 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*, ser. ISCA'19, 2019.

[271] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, 2000.

[272] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, 2002.

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

[274] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context Switch Aware Large TLB," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, 2017.

[275] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018.

[276] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-grained OS Support for Huge Pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[277] 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 Conference on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[278] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

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

[280] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes," in *19th International Symposium on Computer Architecture (ISCA'92)*, 1992.

[281] B. Pham, J. Veselŷ, G. H. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?" in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[282] M. Gorman and P. Healy, "Performance Characteristics of Explicit Superpage Support," in *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010.

[283] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*, 2015.

[284] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large Pages May Be Harmful on NUMA Systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.

[285] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, 1995.

[286] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-Based Superpage-Friendly TLB Designs," in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*, 2015.

[287] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, 2014.

[288] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018.

[289] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-It-Yourself Virtual Memory Translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.

[290] G. C. Engine, "https://cloud.google.com/compute."

[291] "Amazon Elastic Container Service," https://aws.amazon.com/ecs/.

[292] I. Cloud, "https://www.ibm.com/cloud-computing."

[293] M. Azure, "https://azure.microsoft.com."

[294] OpenShift Documentantion, "OpenShift Container Platform Cluster Limits," https://docs.openshift.com/container-platform/3.11/scaling_performance/cluster_limits.html, Red Hat, Inc, Tech. Rep., 2019.

[295] N. Savage, "Going serverless," *Commun. ACM*, 2018. [Online]. Available: http://doi.acm.org/10.1145/3171583

[296] Amazon, "AWS Lambda," https://aws.amazon.com/lambda.

[297] Microsoft, "Azure Functions," https://azure.microsoft.com/en-us/services/functions.

[298] Google, "Cloud Functions," https://cloud.google.com/functions/.

[299] IBM Cloud Functions, "Function-as-a-Service on IBM," https://www.ibm.com/cloud/functions.

[300] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic pp. 427–444.

[301] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019.

[302] B. Burns and D. Oppenheimer, "Design Patterns for Container-based Distributed Systems," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, CO, USA, June 2016.

[303] B. Ibryam, "Principles of Container-based Application Design," https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf, Red Hat, Inc, Tech. Rep., 2017.

[304] IBM, "Docker at insane scale on IBM power systems," https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems.

[305] Linux, "Linux Containers," https://en.wikipedia.org/wiki/LXC.

[306] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 1:1–1:13.

[307] Docker, "What is Docker?" https://www.docker.com/what-docker.

[308] Google, "Production Grade Container Orchestration," https://kubernetes.io.

[309] Docker, "Swarm Mode Overview," https://docs.docker.com/engine/swarm.

[310] A. Ellis, "Open Functions-as-a-Service," https://github.com/openfaas/faas.

[311] Docker, "Best practices for writing Dockerfiles," https://docs.docker.com/develop/develop-images/dockerfile_best-practices, Tech. Rep., 2019.

[312] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context Switch Aware Large TLB," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 449–462.

[313] J. Choi, J. Kim, and H. Han, "Efficient Memory Mapped File I/O for In-Memory File Systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[314] Linux Kernel Documentation, "Address Space Layout Randomization," https://www.kernel.org/doc/Documentation/sysctl/kernel.txt.

[315] PaX Team, “Address Space Layout Randomization,” Phrack, 2003.

[316] M. Howard, “Address space layout randomization in Windows Vista,” Microsoft Corporation, vol.26, 2006.

[317] VMware, “Security Considerations and Disallowing Inter-Virtual Machine Transparent Page Sharing,” ”https://kb.vmware.com/s/article/2080735”, Tech. Rep., 2018.

[318] Google, “Kubernetes pods,” https://kubernetes.io/docs/concepts/workloads/pods/pod, Tech. Rep., 2019.

[319] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, “X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: ACM, 2019, pp. 121–135.

[320] U. Server, “https://www.ubuntu.com/server.”

[321] Docker-ce, “Docker Community Edition,” https://github.com/docker/docker-ce.

[322] ArangoDB Inc., “ArangoDB,” https://www.arangodb.com.

[323] MongoDB Inc., “MongoDB,” https://www.mongodb.com.

[324] Apache Software Foundation, “Apache HTTP Server Project,” https://httpd.apache.org.

[325] A. Kyrola, G. Blelloch, and C. Guestrin, “GraphChi: Large-scale Graph Computation on Just a PC,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46.

[326] Jens Axboe, “Flexible I/O Tester,” https://github.com/axboe/fio.

[327] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford Large Network Dataset Collection,” http://snap.stanford.edu/data.

[328] B. J. McKenzie, R. Harries, and T. Bell, “Selecting a hashing algorithm.” *Software: Practice and Experience*, vol. 20, pp. 209 – 224, 02 1990.

[329] Docker, “Docker Hub,” https://hub.docker.com.

[330] Linux Kernel, “Pagemap,” https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[331] ArangoDB, “Transparent Huge Pages Warning,” https://github.com/arangodb/arangodb/blob/2b84348b7789893878ebd0c8b552dc20416c98f0/lib/ApplicationFeatures/EnvironmentFeature.cpp.

[332] MongoDB, "Transparent Huge Pages Warning," https://github.com/mongodb/mongo/blob/eeea7c2f80bdaf49a197a1c8149d7bc6cbc9395e/src/mongo/db/startup_warnings_mongod.cpp.

[333] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[334] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 430–443.

[335] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 651–664.

[336] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316.

[337] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269.

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

[339] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448.

[340] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[341] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 62–63.

[342] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019.

[343] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758.

[344] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling Page Table Walks for Irregular GPU Applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 180–192.

[345] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 469–480.

[346] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 457–468.

[347] Y. A. Khalidi and M. Talluri, "Improving the address translation performance of widely shared pages," Sun Microsystems, Inc., Tech. Rep., 1995.

[348] X. Dong, S. Dwarkadas, and A. L. Cox, "Shared address translation revisited," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:15.

[349] A. EC2, "https://aws.amazon.com/ec2."

[350] Openstack, "https://www.openstack.org."

[351] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488 pp. 295–308.

[352] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855763 pp. 309–322.

[353] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[354] L. A. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[355] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the Linux Symposium*, 2009, pp. 19–28.

[356] C. R. Chang, J. J. Wu, and P. Liu, "An empirical study on memory sharing of virtual machines for server consolidation," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, May 2011, pp. 244–249.

[357] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM,*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[358] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265930

[359] M. Eto and H. Umeno, "Design and implementation of content based page sharing method in xen," in *International Conference on Control, Automation and Systems (ICCAS)*, Oct 2008, pp. 2919–2922.

[360] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855808 pp. 1–1.

[361] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An empirical study of memory sharing in virtual machines," in *USENIX Annual Technical Conference*. Boston, MA: USENIX, 2012, pp. 273–284.

[362] R. Ceron, R. Folco, B. Leitao, and H. Tsubamoto, "Power systems memory deduplication." [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp4827.html.

[363] UKSM, "http://kerneldedup.org/."

[364] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597655 pp. 53–62.

[365] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "Hicamp: Architectural support for efficient concurrency-safe shared structured data access," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151007 pp. 287–300.

[366] RedHat, "https://www.redhat.com."

[367] K. V. Machine, "https://www.linux-kvm.org."

[368] RedHat, "https://www.redhat.com/en/resources/kvm-kernel-based-virtual-machine."

[369] Intel, "Intel clear containers." [Online]. Available: https://clearlinux.org/features/intel-clear-containers

[370] Intel, "Intel clear containers: A breakthrough combination of speed and workload isolation." [Online]. Available: https://clearlinux.org/sites/default/files/vmscontainers_wp_v5.pdf

[371] L. P. M. MADVISE(2), "http://man7.org/linux/man-pages/man2/madvise.2.html."

[372] L. K. J. H. File, "http://lxr.free-electrons.com/source/include/ linux/jhash.h."

[373] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, July 1970.

[374] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[375] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, "Modeling the implications of dram failures and protection techniques on datacenter tco," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830804 pp. 572–584.

[376] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," in *IBM Executive Overview*, 1997.

[377] Y. Sazeides, E. Özer, D. Kershaw, P. Nikolaou, M. Kleanthous, and J. Abella, "Implicit-storing and redundant-encoding-of-attribute information in error-correction-codes," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540723 pp. 160–171.

[378] U. C. Images, "https://cloud-images.ubuntu.com."

[379] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172 pp. 469–480.

[380] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.

[381] J. Stevenson, "Fine-grain in-memory deduplication for large-scale workloads, phd thesis, stanford university, department of electrical engineering," 2013. [Online]. Available: https://books.google.com/books?id=tsjdnQAACAAJ

[382] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09. New York, NY, USA: ACM, 2009.

[383] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996151 pp. 135–146.

[384] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000101 pp. 307–318.

[385] Y. Du, M. Zhou, B. R. Childers, D. Mosse, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 223–234.

[386] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively breaking large pages to improve memory overcommitment performance in vmware esxi," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2731186.2731187 pp. 39–51.

[387] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830773 pp. 1–12.

[388] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked dram caches," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485958 pp. 416–427.

[389] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 1999.

[390] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating dependent cache misses with an enhanced memory controller," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.46 pp. 444–455.

[391] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "Chargecache: Reducing dram latency by exploiting row access locality," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 581–593.

[392] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Jan 2010, pp. 1–12.

[393] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 65–76.

[394] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 128–138.

[395] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *2008 International Symposium on Computer Architecture*, 2008.

[396] L. Vu, D. Skarlatos, H. Sivaraman, U. Kurkure, A. Bappanadu, and R. Soundararajan, "Secure Cloud-based Machine Learning," in *United States Patent and Trademark Office Patent Application No. 16/417,139*, 2019.

[397] A. Kokolis, D. Skarlatos, and J. Torrellas, "PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[398] D. Skarlatos, R. Thomas, A. Agrawal, S. Qin, R. Pilawa-Podgurski, U. R. Karpuzcu, R. Teodorescu, N. S. Kim, and J. Torrellas, "Snatch: Opportunistically reassigning power allocation between processor and memory in 3D stacks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[399] B. Gopireddy, D. Skarlatos, W. Zhu, and J. Torrellas, "HetCore: TFET-CMOS Hetero-Device Architecture for CPUs and GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.