Ghosting the Spectre: fine-grained control over speculative execution [DRAFT COPY]

Allison Randal

allison.randal@cl.cam.ac.uk

Summary

This dissertation considers three approaches that partially or completely eliminate speculative execution from modern hardware architectures, as a finer-grained approach to mitigating the speculative execution vulnerabilities. Current mitigations for the speculative execution vulnerabilities only offer partial protection, have prohibitive performance penalties, and apply globally so mitigations must be chosen during hardware manufacture or data center deployment. Systems software developers have little or no control over which mitigations are deployed, and therefore no choice in whether they endure the risk of speculation or suffer the performance penalty of mitigations.

I begin by tracing the root cause of the speculative execution vulnerabilities to a fundamental design flaw in modern superscalar hardware architectures—that all speculated branches and memory loads leave unprotected microarchitectural state vulnerable to attack. Many mitigations have been proposed and implemented for many variants of the speculative execution vulnerabilities, but none of the mitigations do more than limit some damage caused by some variants. In order to demonstrate that a more comprehensive solution to the vulnerabilties is feasible in modern hardware architectures, I propose three alternative approaches that partially or completely eliminate speculative execution.

Heterogeneous multicore systems that combine both speculative and non-speculative cores (codename Gluon) make it possible to entirely disable speculation for security-critical or untrusted sections of code, by running that code on a non-speculative core. Code running on a speculative core performs as well as it would on an ordinary speculative hardware architecture. The systems software developer has the power to choose which code runs with the performance advantage of speculation, and which code runs with the security advantage of no speculation. However, Gluon-type multicores only offer the ability to disable speculation at the process or thread level. A finer-grained approach is desirable, to limit the performance penalty of disabled speculation to the smallest possible region of code.

Superscalar processors without speculation features (codename Tachyon) keep the performance advantages of most common features in modern hardware architectures—such as dynamic multiple issue, dynamic pipeline scheduling, out-of-order execution, and register renaming—while avoiding the risk of speculative execution. Such processors do not perform as well as equivalent speculative processors, but the results of this work indicate that they can perform better than equivalent speculative processors with all relevant mitigations for the speculative execution vulnerabilities applied. The performance penalty of eliminating speculation can also be partially offset by increasing the size of fetch and issue stage components in the pipeline. Tachyon-type cores are always non-speculative, so they do not give systems software developers the option to choose between performance and security. However, these cores may be desirable for multitenant infrastructure deployments that exclusively serve privacy-centered workloads, such as processing hospital patient data.

Superscalar processors that include both speculative and non-speculative variants of branch and memory load instructions (codename Dyon) make it possible to disable speculation at the level of a single instruction, by selecting a non-speculative instruction instead of a speculative instruction. Making Dyon-type features accessible to systems software developers requires modest changes to higher-level languages and compiler toolchains—to annotate which regions of code should be non-speculative, and compile that code down to non-speculative instructions. The performance of Dyon-type cores is proportional to the use of speculative and non-speculative instructions, so only regions of code that disable speculation pay a performance penalty. Out of the three approaches considered in this dissertation, Dyon-type cores are best for large-scale general-purpose multitenant infrastructure deployments, because they simplify resource allocation by keeping all cores identical, have no performance penalty for code compiled as entirely speculative, and give systems software developers the most precise control over speculation.

Acknowledgements

Thanks to my supervisor Richard Mortier for respecting me as an equal and shielding me from the distractions of administrative details. Thanks to Ravi Nair for help locating and scanning copies of several pivotal IBM papers on virtual machines from the 1960s that were no longer (or perhaps never) available in libraries or online. Thanks also to the reviewers for various stages of this work (alphabetically): Clint Adams, Matthew Allen, Ross Anderson, Alastair Beresford, James Bottomley, Peter Capek, Damian Conway, Kees Cook, Jon Crowcroft, Mike Dodson, Tony Finch, Greg Kroah-Hartman, Anil Madhavapeddy, Ronald Minnich, Davanum Srinivas, Tom Sutcliffe, Zahra Tarkhani, and Daniel Thomas. Their feedback was greatly appreciated. Thanks to Robert Watson and the CHERI research group for inspiration and renewed motivation in the final months of writing up.

Contents

1 Introduction **8**

2 Background **10**
2.1 Terminology . . . 11
2.2 Time-sharing on mainframes . . . 12
2.3 General-purpose hardware and general-purpose operating systems . . . 13
2.4 Speculative execution . . . 14

3 Heterogeneous multicores **18**
3.1 Related work . . . 18
3.2 Feasibility considerations . . . 19
3.2.1 Production hardware . . . 19
3.2.2 Prototyping . . . 20
3.2.3 Kernel . . . 20
3.2.4 Workloads . . . 21
3.3 Evaluation . . . 21
3.3.1 Performance . . . 22

4 Out-of-order cores without speculation **24**
4.1 Related work . . . 24
4.2 Feasibility considerations . . . 25
4.2.1 Speculative branch instructions . . . 25
4.2.2 Non-speculative branch instructions . . . 28
4.2.3 Speculative memory load instructions . . . 32
4.2.4 Non-speculative memory load instructions . . . 33
4.2.5 Speculation buffers for memory load instructions . . . 35
4.2.6 Thread-level parallelism . . . 35
4.3 Evaluation . . . 36
4.3.1 Performance . . . 37

5 Demi-speculative features within a single core **44**
5.1 Related work . . . 44
5.2 Feasibility considerations . . . 45
5.2.1 RISC-V ISA extensions . . . 45
5.2.2 Microarchitecture . . . 47
5.2.3 High-level language modifications . . . 50
5.2.4 Compiler toolchain modifications . . . 52
5.3 Evaluation . . . 53

5.3.1 Performance 53

6 Conclusions **57**

6.1 Future work 58

6.1.1 Heterogeneous multicores 59

6.1.2 Non-speculative cores 59

6.1.3 Demi-speculative cores 60

Bibliography **61**

Published work

In the course of this research, I published the following article that contributed directly to the contents of this dissertation:

The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers, published in ACM Computing Surveys in February 2020 [99].

Chapter 1

Introduction

A series of vulnerabilities related to speculative execution and side-channel attacks rose to attention in 2018. The techniques behind the speculative execution vulnerabilities were not new, but the combined application of the techniques was more sophisticated, and the security impact more severe, than previously considered possible. The models of secure isolation employed by multitenant infrastructures such as virtual machines and containers offer little protection from the speculative execution vulnerabilities. While mitigation patches have typically been applied quickly for the known variants of these vulnerabilities, these patches work around the vulnerabilities but do not eliminate them, and the probability of further variants being discovered in the coming years is high. Not all modern processors use speculative execution as a technique for instruction-level parallelism, but the processors that do use it provide no mechanism to disable the feature entirely.

Completely eliminating speculative execution from modern hardware architectures significantly degrades performance [66, 76, 110, 128, 19, 48, 104]. However, current approaches to mitigating the speculative execution vulnerabilties are largely global in nature, working on the premise that speculative execution is a hidden microarchitectural optimization, and so the mitigations must also be hidden in the microarchitecture. These global mitigations have severe performance penalties, so large-scale multitenant infrastructure providers are forced to choose between enabling mitigations globally on thousands or hundreds of thousands of machines, or disabling the mitigations for a substantial performance boost, on the assumption that successful exploits of the speculative execution vulnerabilities will be rare enough to have a statistically low impact. The not-very-well-kept secret of the industry is that public cloud providers generally run with many speculative execution mitigations disabled, because it is not cost effective to enable them.

If we shift the paradigm slightly, and stop viewing speculation as a universal optimization hidden in the microarchitecture, it opens the door to different approaches. Systems software developers are familiar with making trade-offs between security and performance, and with the fact that different technical choices may be appropriate for different software contexts such as host kernels and operating systems, guest kernels and operating systems, application/workload software, and encrypted data. The choice between the performance boost of running with speculation and the security benefits of running without speculation could be made at a much tighter level of granularity than an entire machine or an entire data center, if hardware architectures supported it. A finer-grained approach to the speculative execution vulnerabilities has the potential to both improve the security of security-critical code and data—by entirely disabling speculation when that is needed—at the same time as reducing the system-wide performance penalty for

large-scale deployments—by limiting disabled speculation to the smallest possible scope.

A significant percentage of the code execution in any general-purpose host or guest operating system can be safely and securely run with speculative execution enabled. As long as speculative execution continues to have a performance advantage over non-speculative execution, it will be desirable to restrict non-speculative execution to the smallest possible region of running code. However, current hardware architectures have no way to identify whether a region of code should be run speculatively or non-speculatively, and no mechanism to switch between speculative and non-speculative modes of execution. We first explore a simplistic approach to designing hardware capable of enabling and disabling speculative execution in response to higher-level demand, by combining speculative and non-speculative cores in a heterogeneous multicore architecture, which gives systems software developers control over speculation at the level of a process or thread. We then explore the logical limits of superscalar processors without speculative execution within the context of modern hardware architectures. We finally explore an alternative approach to designing hardware capable of enabling and disabling speculative execution, by combining speculative and non-speculative instructions on a single core, which gives systems software developers control over speculation at the level of compiler instruction selection. While the scope of this dissertation is limited to preliminary exploration and comparison of possible future directions for hardware and software, we hope that it may help encourage hardware vendors to consider the kind of fundamental hardware architecture changes the industry needs to effectively control speculative execution.

Throughout this dissertation, we use the RISC-V architecture as a base for discussion and implementation. This choice is pragmatic rather than dogmatic—the RISC-V instruction set architecture (ISA) is open source, has established open source implementations of cores available for reuse and modification, and has extensive open source tools for implementing and simulating custom RISC-V cores and SoCs. It is the open nature of RISC-V that makes it so adaptable for design space exploration, and as such, RISC-V has been a target of substantial recent research on hardware architectures, including speculative execution and related vulnerabilities. We do not intend our choice of RISC-V to imply that the architecture is universally superior to other more mature architectures, such as x86 or ARM; we recognize that RISC-V is still in a relatively early stage of architecture design and implementation and has certain limitations. Nor do we expect to see RISC-V servers replace the x86 and ARM servers currently running in large scale public clouds anytime soon. However, the insight gained through this work has microarchitectural implications for all speculative superscalar processors, no matter what ISA they implement.

The next chapter provides necessary background for the work of the dissertation. Chapter 3 explores combining speculative and non-speculative cores in a heterogeneous multicore architecture. Chapter 4 explores the logical limits of superscalar pipelining techniques without speculation in the microarchitecture of a standard ISA. Chapter 5 explores combining speculative and non-speculative instructions in a single ISA, adding instructions for selective speculation as an extension to the RISC-V ISA.

Chapter 2

Background

Many modern computing workloads run in multitenant environments, where each physical machine is split into hundreds or thousands of smaller units of computing, generically called *guests*. Cloud and containers are currently the leading approaches to implementing multitenant infrastructures, but other related technologies, such as unikernels or serverless, are also variations on multitenant infrastructures. The guests in a cloud deployment are commonly called virtual machines or cloud instances, while the guests in a container deployment are commonly called containers. Typically, a single *tenant* (a user or group of users) is granted access to deploy guests in an orchestrated fashion across a cloud or cluster made up of thousands or hundreds of thousands of physical machines located in the same data center or across multiple data centers, to facilitate operational flexibility in areas such as capacity planning, resiliency, and reliable performance under variable load. Each guest runs its own (often minimal) operating system and application workloads, and maintains the illusion of being a physical machine, both to the end users who interact with the services running in the guests, and to developers who are able to build those services using familiar abstractions, such as programming languages, libraries, and operating system features. The illusion, however, is not perfect, because ultimately the guests do share the hardware resources (CPU, memory, cache, devices) of the underlying physical host machine, and consequently also have greater access to the host's privileged software (kernel, operating system) than a physically distinct machine would have.

Ideally, multitenant environments would offer strong isolation of the guest from the host, and between guests on the same host, but reality falls short of the ideal. The approaches that various implementations have taken to isolating guests have different strengths and weaknesses. For example, containers share a kernel with the host, while virtual machines may run as a process in the host operating system or a module in the host kernel, so they expose different attack surfaces through different code paths in the host operating system. Fundamentally, however, all existing implementations of virtual machines and containers are leaky abstractions, exposing more of the underlying software and hardware than is necessary, useful, or desirable. New security research starting in 2018 delivered a further blow to the ideal of isolation in multitenant environments, demonstrating that certain hardware vulnerabilities related to speculative execution—including Spectre, Meltdown, Foreshadow, L1TF, and variants—can easily bypass the software isolation of guests.

Because multitenancy has proven to be useful and profitable for a large sector of the computing industry, it is likely that a significant percentage of computing workloads will continue to run on multitenant infrastructure for the foreseeable future. Randal [99] examined the co-evolution of software and hardware for multitenant infrastructures over

sixty years of history, and how the trade-offs made along the way led to the current tension between the lofty ideals of security versus the flawed reality. This dissertation focuses on the hardware dimension of multitenant infrastructures, and particularly on the impact of speculative execution vulnerabilities.

In the context of this dissertation, it is worth keeping in mind that unlike desktop systems—where a relatively small number of cores run a relatively small number of processes for a small number of closely-related users (perhaps only one human user)—multitenant systems run a massive number of processes/workloads on a massive number of cores for a massive number of unrelated users. At the microarchitecture level, this means that rather than optimizing for executing lengthy sequences of contiguous instructions on a core, the implementation needs to optimize for sharing the core between unrelated workloads and users, with a tendency toward executing shorter sequences of contiguous instructions and rapidly switching between multiple unrelated instruction streams. The inherently fragmented nature of multitenant systems does not dictate a radical departure from established microarchitecture designs, but it is suggestive of directions for exploration.

2.1 Terminology

For the sake of clarity, this dissertation consistently uses certain modern or common terms, even when discussing literature that used various other terms for the same concepts.

- **cloud**: Implementation approaches that adopt the label "cloud" are typically virtual machines with added orchestration features to enhance portability. Cloud implementations also tend to favor lighter-weight guest images, which enhances performance and reduces complexity, though cloud images are generally not quite as minimal as container images.

- **container**: The term "container" does not have a single origin, but some early relevant examples of use are Banga *et al.* [12] in 1999, Lottiaux and Morin [77] in 2001, Morin *et al.* [86] in 2002, and Price and Tucker [97] in 2004. Early literature on containers confusingly referred to them as a kind of virtualization [97, 114, 84, 60, 20, 24], or even called them virtual machines [114]. As containers grew more popular, the confusion shifted to virtual machines being called containers [17, 130]. This dissertation uses the term "container" for multitenant deployment techniques involving process isolation on a shared kernel (in contrast with *virtual machine*, as defined below). However, in practice the distinction between containers and virtual machines is more of a spectrum than a binary divide. Techniques common to one can be effectively applied to the other, such as using system call filtering with containers, or using seccomp sandboxing or user namespaces with virtual machines.

- **guest**: The term "guest" had some early usage in the 1980s for the operating system image running inside a virtual machine [87], but was not common until the early 2000s [121, 13]. This dissertation uses "guest" as a general term for operating system images hosted on multitenant infrastructures, but occasionally distinguishes between virtual machine guests and container guests.

- **kernel**: A variety of different terms appear in the early literature, including "supervisory program" [25], "supervisor program" [5], "control program" [89, 93, 2],

"coordinating program" [93], "nucleus" [18, 26], "monitor" [125], and ultimately "kernel" around the mid-1970s [74, 96]. This dissertation uses the modern term "kernel".

- **process**: The early literature tended to use the terms "job" [103] or "program" [25, 93, 5], and "process" only appeared around the mid-1960s [30, 1]. This dissertation uses the modern term "process". The early use of "multiprogramming" meaning "multiprocessing" was derived from the early use of "program" meaning "process".
- **serverless**: Implementation approaches that adopt the label "serverless" tend to emphasize portability and minimizing complexity. They rely on the underlying infrastructure—typically some combination of bare metal, virtual machines, and/or containers—for whatever secure isolation and performance they provide.
- **unikernel**: Implementation approaches that adopt the label "unikernel" take minimalist guest images to an extreme, by replacing the kernel and operating system of the guest with a set of highly-optimized libraries that provide the same functionality. The code for an application workload is compiled together with the small subset of unikernel libraries required by the application, resulting in a very small binary that runs directly as a guest image. Historically, unikernels have sacrificed portability of guest images, by targeting only a limited set of virtual machine implementations as their host, but recent work has explored running unikernels as containers [126]. The unikernel approach also reduces the portability of application code, since unikernel frameworks tend to require the application code to be written in a specific way to integrate with the unikernel libraries.
- **virtual machine**: This dissertation uses the term "virtual machine" for multi-tenant deployment techniques involving the replication/emulation of real hardware architectures in software (in contrast with *container*, as defined above). The code responsible for managing virtual machine guests on a physical host machine is often called a "hypervisor" or "virtual machine monitor", both derived from early terms for the kernel, "supervisor" and "monitor". In many early implementations of virtual machines, the host kernel managed both guests and ordinary processes.

2.2 Time-sharing on mainframes

The earliest form of hardware for multitenant infrastructures was time-sharing systems on mainframes. While the hardware of today is radically different than the hardware of the 1950s-1970s, several key concepts continue to be relevant.

The first key concept was simply the ability to run more than one process on a machine at the same time. This concept was originally called *multiprogramming* and evolved through a series of hardware architectures in the 1950s, notably the IBM 705 [103], EDSAC [124], UNIVAC LARC [34], STRETCH (IBM 7030) [33, 25], and TX-2 [41]. Multiprogramming involved both multitasking (as simple context-switching) and multiprocessing (as multiple CPUs and dedicated I/O processors), which introduced a risk of processes disrupting the operation of other processes on the same machine.

So, the first key concept led naturally to the second key concept: isolating processes to prevent them disrupting each other. Initially, this work revolved around the now familiar

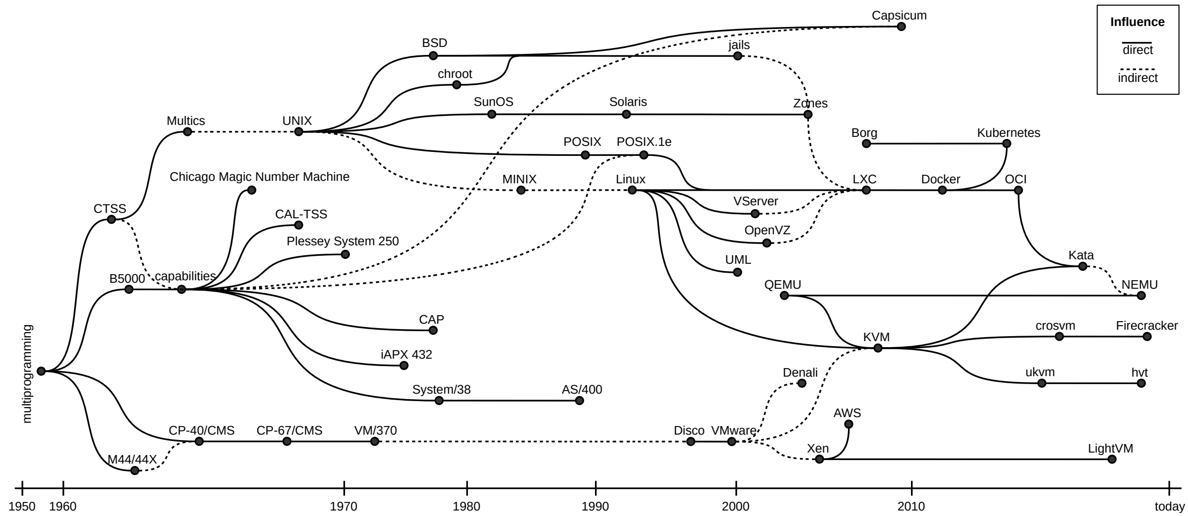

Figure 2.1: The evolution of multitenant infrastructures. Reprinted from Randal [99].

approach of a small privileged kernel with unrestricted access to all hardware resources and running processes, as well as responsibility for potentially disruptive operations such as memory and storage allocation, process scheduling, and interrupt handling, combined with restrictions on any software outside the kernel to limit access to these risky features. STRETCH [25] in the 1950s and IBM System/360 [5] in the 1960s were significant early examples of hardware architectures designed to provide hardware support for kernel process isolation.

The concept of process isolation led to two major divergent schools of thought on hardware security for the multitenant systems of the time—*capabilities* and *virtual machines*—both initially focused on strengthening process isolation by adding memory isolation features. Capabilities viewed secure isolation as an essential feature of the hardware and operating system, which should be available to every process [99, pp. 6-8]. Virtual machines approached secure isolation at a different level of granularity, emphasizing the ability to run an entire operating system in an isolated environment by closely replicating the behavior of physical hardware [99, pp. 8-11]. Both capabilities and virtual machines depended heavily on custom hardware implementations of their security features—the noteworthy early hardware examples for capabilities were the Chicago Magic Number Machine [37, 38], CAL-TSS [72, pp. 52-57], CAP [88, 123], IBM System/38 [14, 54], and Intel iAPX 432 [55, 49, 79, 91], and for virtual machines were the IBM M44/44X [89], CP-40/CMS and CP-67/CMS on the IBM System/360 [2, 18, 29], and VM/370 on the IBM System/370 [47, 29]. The fundamental principles of these two major divergent schools of thought continue today, as a dichotomy between modern containers [99, pp. 15-20] and modern virtual machines [99, pp. 12-15].

2.3 General-purpose hardware and general-purpose operating systems

As early as the 1960s, hardware vendors recognized that designing complete custom hardware, custom operating systems, and custom application software for each generation of their products was an expensive way to approach systems development. In 1964, Amdahl

et al. [5] discussed the philosophy of "general-purpose CPU design for communications-oriented systems" as a driving design principle for the IBM System/360. The idea led to a third key concept that survives in modern multitenant infrastructures—portability made possible by architectural stratification and standardization.

The 1970s and 1980s saw the rise of general-purpose hardware capable of running multiple different operating systems, general-purpose operating systems capable of running on multiple different hardware architectures, and application/workload software capable of running on multiple different operating systems and hardware architectures. On the hardware side, companies like DEC, Honeywell, HP, Intel, and Xerox shifted their product lines toward simpler general-purpose hardware architectures that no longer supported the security features of capabilities [90, 79, 115] or virtual machines [31, 42]. On the operating system side, MIT's Compatible Time-Sharing System (CTSS) [28, 125] laid the foundation for Multics [27], which later inspired UNIX [102] and its robust mutation, the Berkeley Software Distribution (BSD) [83, 82], followed by Solaris, Linux, and their many variants.

On one hand, stratification and standardization were a substantial benefit to the hardware and software industries, as both hardware and software architectures grew so much more complex over the decades, that the only sustainable approach to ongoing development of the full hardware and software stack was to break it into modular and recombinable hardware components—such as CPUs, memory, and storage—together with modular and recombinable software components—such as kernels, system utilities, operating systems, and applications. On the other hand, stratification and standardization were also a source of risk, as researchers and engineers working at one architectural level tended to have less and less exposure over time to how other levels actually functioned, across the boundaries of microarchitecture, instruction set architecture, peripherals, kernel and user space features, and application/workload software. This fundamental disconnect has played a part in the speculative execution vulnerabilities. Modern software security research relies on critical assumptions about the behavior of the hardware that have been false for decades, but software security research is so far removed from modern microarchitecture research that few researchers saw the risk, and even those who did [95] radically underestimated the impact.

2.4 Speculative execution

A collection of papers in the early 1970s, including Tjaden and Flynn [118], Flynn [39], Flynn and Podvin [40], and Riseman and Foster [101], explored the logical limits of instruction-level parallelism for the hardware of the time, identifying branches and memory loads as significant obstacles. Within a decade, the tone of publications shifted from assessing these obstacles as insurmountable, to assessing them as straightforwardly solved by combining several techniques that remain in common use today, particularly the speculative techniques of branch prediction and hardware prefetching. Other techniques for instruction-level parallelism developed around the same time—such as multiple instruction issue, register renaming, dynamic pipeline scheduling, and out-of-order execution—are commonly combined with speculation today, but are not inherently speculative.

Lee and Smith [71] and McFarling and Hennessy [80] captured historical perspectives on branch prediction from the point of view of the mid-1980s. Both surveyed the state of the art in branch prediction techniques at the time—such as dynamic prediction and branch target buffers—and critically reviewed previous techniques to speed up conditional

branches without speculation—such as delayed branches, look-ahead resolution, branch target prefetching, and multiple instruction streams. Smith [113, 112] captured similar early perspectives on hardware prefetching in the late 1970s, surveying the impact of memory access latency for both instruction fetching and memory load instructions, the limitations of existing implementations of instruction and data prefetching at the time, and the potential for future performance improvements. One noteworthy characteristic shared by these papers—and by much of the substantial work on speculative techniques in the decades that followed—was a focus on metrics of performance with little or no consideration given to metrics of security.

In 2005, Percival [95] explored the risks inherent in combining speculative execution with simultaneous multithreading, dynamic pipeline scheduling, multilevel memory caches, and hardware prefetching, but did not recognize the full extent of the security impact. Early in 2018, Kocher *et al.* [66] and Lipp *et al.* [76] published a set of vulnerabilities involving speculative execution techniques, respectively called Spectre and Meltdown. A number of variants on the first reported vulnerabilities soon followed, including Schwarz *et al.* [110] on NetSpectre, Van Bulck *et al.* [119] on Foreshadow or more broadly "L1 Terminal Fault" (L1TF), Weissse *et al.* [122] on Foreshadow-NG, Stecklina and Prescher [116] on LazyFP, Chen *et al.* [23] on SGXPectre, and Islam *et al.* [59] on Spoiler. A second round of speculative execution vulnerabilities specifically targeted the mitigations for the first round of vulnerabilities, including Schwarz *et al.* [109] on ZombieLoad, van Schaik *et al.* [105] on RIDL, Minkin *et al.* [85] on Fallout, van Schaik *et al.* [106] on CacheOut, Van Bulck *et al.* [120] on Load Value Injection (LVI), and Ragab *et al.* [98] on CrossTalk. The overall trend in the variants reported over the years is one of increasingly broad impact.

Studies by Canella *et al.* [19], He *et al* [50], and Genkin and Yarom [46] have systematically reviewed the broad range of speculative execution vulnerabilities, tracing them to microarchitectural root causes. All speculative execution vulnerabilities take advantage of the transient nature of speculated instructions, exploiting race conditions between creating microarchitectural state for a transient instruction and the retirement actions that would discard that microarchitectural state [19, 50]. Concretely, in the dynamic pipeline of a modern microarchitecture, the zone of risk for speculative execution vulnerabilities lies between the point where a functional unit executes an instruction speculatively, and the point where the commit unit finally retires that instruction as correctly or incorrectly speculated, as illustrated in Figure 2.2. Spectre-type attacks use techniques to mistrain the branch predictor or memory disambiguator by feeding it false history, thus poisoning the microarchitectural state used to make speculative predictions for branch/return instructions or memory loads, and steering execution toward code paths and memory locations targeted by the attacker [19, 50]. Meltdown-type attacks exploit the fact that exceptions are only raised after a faulting instruction is finally committed, so when a transient instruction triggers an exception, that exception is suspended long enough for the attack to access microarchitectural state through covert channels [19, 50]. The mechanisms manipulated as part of successful attacks have ranged as broadly as the L1 data cache [66, 75, 119, 122, 59, 120], uncached memory [75], the Line Fill Buffer (LFB) [75, 105, 109, 120], the store buffer [85, 120], the Branch Target Buffer (BTB) [66], the Branch History Buffer (BHB) [36, 52], the Pattern History Table (PHT) [66, 65, 36, 110], the Return Stack Buffer (RSB) [68, 78], and the memory disambiguator [53, 59]. The targets of successful attacks have included simple out-of-bounds memory reads [66, 19, 32] and writes [65], breaking type and memory safety guarantees [52], arbitrary code execution

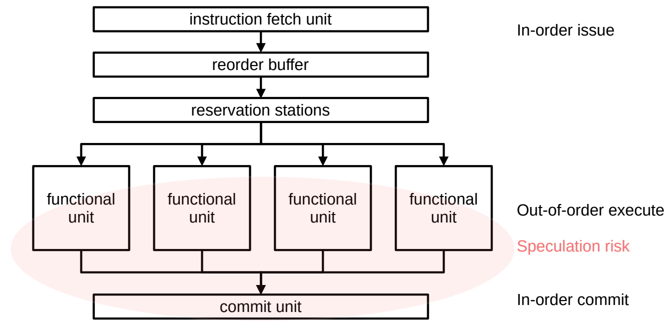

Figure 2.2: Zone of risk for speculative execution in a dynamically scheduled pipeline. Adapted from Patterson and Hennessy [94], Figure 4.69, p. 330.

[65, 66, 68, 78], reading privileged system registers [58], reading FPU and SIMD registers across processes or virtual machines [116], writing over read-only memory [65], bypassing memory-protection keys [19, 57], exposing secret data from SGX enclaves [119, 92, 23, 19, 56], and full access to host kernel memory from unprivileged host user space [75, 52, 56], from an unprivileged guest [122], or remotely over the network [110]. Essentially, the speculative execution vulnerabilities violate the principles of isolation that are the intrinsic purpose of multitenant infrastructures.

The low-level nature of the speculative execution vulnerabilities can make it challenging for the average software developer or security practitioner to reason about them. The diverse assortment of mechanisms and targets in the reported variants appear illogically disconnected, until you grasp the fundamental principle that all speculated branches and memory loads leave unprotected microarchitectural state vulnerable to attack. The reported variants are all practical applications of the same fundamental principle, and the greatest limitation on future variants is only the prospect that not all leaked microarchitectural state contains interesting secrets worth targeting in an attack. It is highly likely further variants will be discovered in the years ahead, because a vulnerability touching every speculated branch and memory load is so broadly applicable that researching new variants to report becomes an exercise in exploring which of the many viable targets have interesting security consequences.

As we will discuss in more detail in Chapter 4, the speculation features that Spectre-type attacks and Meltdown-type attacks exploit are common to modern major hardware architectures, such as x86 and ARM, and had already begun to be replicated in RISC-V implementations before the vulnerabilities were reported. These vulnerabilites are not bugs in the traditional sense, because the speculation features are functioning as they were designed. However, they are flaws in the fundamental design of the speculation features as optimizations to improve instruction-level parallelism, because optimized code running with speculation does not preserve the semantics and security properties of un-optimized code running without speculation. The range of proposed and implemented mitigations limit some of the damage caused by some variants, but the meager protections they offer carry prohibitive performance penalties, and they do not resolve the inherent logic flaws of speculative execution as a microarchitectural optimization, which are the true root cause of the entire class of vulerabilities [19, 46, 81].

We can never know what might have happened if the security trade-offs of speculative execution had been fully considered at the same time the performance advantages were discovered—whether the vulnerabilities might have been exposed earlier, or whether

modern computer architectures might have evolved down a slightly different path. What we do know is that we have the ability to re-consider the security trade-offs today, and make different architectural choices for the future. This dissertation explores the security and performance trade-offs between three different architectural approaches that partially or completely eliminate speculative execution, serving as an illustration that fine-grained control over speculation is both feasible and reasonable to consider in the near-term future of mainstream modern computer architectures. While it may not be fair to judge past work by lessons we learned later, it will be fair to judge future work on whether it applies those lessons or ignores them.

Chapter 3

Heterogeneous multicores

In Chapter 2, we discussed the fundamental problem that all speculated branches and memory load instructions leak microarchitectural state, but also observed that only some of that leaked microarchitectural state has interesting security consequenses. Current approaches to mitigating the speculative execution vulnerabilties are largely global in nature, working on the premise that speculative execution is a hidden microarchitectural optimization, and so the mitigations must also be hidden in the microarchitecture. These microarchitectural mitigations have no mechanism to differentiate between microarchitectural state that needs to be protected (such as passwords, decrypted secrets, read-only or protected memory, privileged system registers) and microarchtectural state that doesn't need to be protected (vast quantities of numeric, text, or other data with no particular security significance). The level of detailed semantic information required to identify security-critical code and data only exists in higher-level languages (and in the minds of the developers), and is lost through the compilation process down to the lower levels of ISA instructions and machine code that actually runs on the hardware. If we shift the paradigm slightly, and view speculation not as a universal optimization hidden in the microarchitecture, but instead as a high-level trade-off between security and performance, it opens the door to different approaches.

This chapter explores combining speculative and non-speculative cores in a heterogeneous multicore system, as a relatively straightforward way to put fine-grained control over speculation into the hands of systems software developers. Adding non-speculative cores makes it possible to run limited sections of code non-speculatively—when the code is either critical to security or untrusted—while allowing most code to run speculatively for performance gains. Chapters 4 and 5 explore more experimental approaches to fine-grained control over speculation, but heterogeneous multicore architectures are in active use today, and so have the advantage of being more immediately amenable to implementation and analysis. This chapter explores the feasibility of heterogeneous multicore processors in the context of a modern cloud/container hardware and software stack and evaluates the performance of a heterogeneous multicore SoC in RTL simulation.

3.1 Related work

Heterogeneous multicore systems have been an area of increasing interest in systems research, as well as increasing plausibility in practical applications, since the early 2000s. While earlier work tended to focus on performance advantages or energy efficiency advan-

tages of the approach, more recent work has brought attention to security advantages. As early as 2003, Kumar *et al.* [69] simulated a model combining cores with the same instruction set architecture (ISA) but different microarchitectures—some in-order and some out-of-order—on a single heterogeneous multicore chip, dynamically migrating a thread between cores to improve performance (on a more powerful core) or energy efficiency (on a less powerful core). In 2010, Li *et al.* [73] modified the Linux Kernel to simultaneously support sets of cores with different performance characteristics and slightly different instruction set architectures (ISAs), dynamically migrating threads between different kinds of cores to improve performance and throughput. In 2014, Aminot *et al.* [7] explored combining cores with different ISAs, some with a minimal set of instructions and others with added instruction extensions for less frequently used and more energy hungry features, dynamically migrating code between cores to improve energy efficiency. In 2017, Birhanu *et al.* [15] built on the ARM big.LITTLE architecture—a heterogeneous multicore architecture combining cores with the same ISA but different power and performance characteristics—and demonstrated that replacing the Linux Kernel's Completely Fair Scheduler (CFS) with their Fastest-Thread-Fastest-Core (FTFC) scheduler, which dynamically migrated threads between "big" and "little" cores based on CPU utilization and capacity, improved power efficiency by 2.22% and improved performance by 52.62%. In 2019, the mainline Linux Kernel accepted the Energy Aware Scheduler (EAS) [63], specifically for heterogeneous multicore architectures like ARM big.LITTLE, which similarly takes CPU utilization and capacity into account to improve energy efficiency while minimizing negative impact on performance. In 2020, Ainsworth and Jones [4] proposed adding a set of non-speculative special-purpose cores with different ISAs running specialized kernels, alongside the main speculative general-purpose cores, to improve the security of small regions of code that are either offloaded to the non-speculative cores or run as independent validation of results on the main cores. Also in 2020, Le *et al.* [70] briefly suggested a heterogeneous multicore system combining non-speculative Rocket cores [8] with speculative BOOM cores [22, 21] as a mitigation approach for speculative execution vulnerabilities, without any implementation details.

3.2 Feasibility considerations

Beyond academic research, several factors make the heterogeneous multicore approach worthwhile to consider as a possibility for the near-term future of mainstream hardware.

3.2.1 Production hardware

In mainstream production hardware, the ARM big.LITTLE architecture has been shipping in smartphones (such as Samsung and Apple) for several years, and Apple's M1 family of SoCs with their own heterogeneous ARM architecture have been shipping in tablets and laptops for a little over a year. Intel has also been trying out a heterogeneous multicore approach for x86 architectures, with the Lakefield mobile processors and Alder Lake desktop processors. These architectures work well in a mobile devices or laptops—where peak performance is only required sporadically, cores are frequently idle, and switching low priority workloads to lower power cores can significantly extend battery life. It is less clear that heterogeneous multicores for server processors are desireable in multitenant infrastructures—which require consistent peak performance for all workloads, and aim to

minimise idle cores by sharing the same hardware resources between many workloads for many tenants—as we will discuss further in Section 3.3.

3.2.2 Prototyping

The University of Berkeley's Chipyard framework [6] for design, simulation, and implementation of RISC-V SoCs—originally called the "Rocket Chip Generator" [8]—includes a collection of default configurations to build SoCs that combine varying numbers of heterogeneous cores, including non-speculative Rocket cores and speculative BOOM cores. Balkind *et al.* [10] developed the BYOC ("bring your own core") framework—extending the earlier OpenPiton framework [11]—explicitly for the purpose of supporting implementations of heterogeneous multicore systems.

Both Chipyard and BYOC provide open hardware implentations of common shared components, including memory with coherent caches, accelerators, and standard peripherals such as UART, block devices, and NICs. Both frameworks support software simulation, FPGA emulation, and tapeout to silicon as output targets. Chipyard is implemented as a parameterized hardware generator based on Chisel, with support for integrating Verilog components directly, while BYOC is implemented in Verilog and SystemVerilog. Both frameworks have a strong emphasis on enabling verification and validation of designs. Both frameworks support heterogeneous cores as general-purpose first-class citizens, but Chipyard focuses on cores running versions of the RISC-V ISA plus extensions, while BYOC includes implementations of cores using RISC-V, x86, and SPARC ISAs, and specifically targets combining different ISAs into unified general-purpose heterogeneous multicore systems.

In addition to Chipyard and BYOC, a number of other more specialized frameworks build custom multicore systems with varying degrees of heterogeneity. OpenPiton [11] was originally focused exclusively on the SPARC ISA, but later added the RISC-V ISA in the form of the 64-bit Ariane core [9]. lowRISC [16] combines RISC-V cores with different ISA extensions, however the small cores only serve as subordinate "minions".

The existence of frameworks like Chipyard and BYOC mean that it is currently relatively straightforward to build custom heterogeneous multicore systems for development and testing, using either the same ISA with different microarchitectures or different ISAs.

3.2.3 Kernel

The Linux Kernel already supports heterogeneous multicore systems like ARM's big.LITTLE architecture with the Energy Aware Scheduler [63] or Capacity Aware Scheduler [62]. A speculation-centric heterogeneous architecture could use those existing schedulers, with the non-speculative cores serving the purpose of the lower energy, less powerful "little" cores, and the speculative cores serving the purpose of the higher energy, more powerful "big" cores. However, the existing schedulers would only take into account CPU utilization, capacity, and energy usage, and would not give any consideration to the security implications of the differences between the big and little cores.

A new scheduler would be necessary to take full advantage of speculation-centric heterogeneous multicore systems, perhaps named the "Speculation Aware Scheduler". Rather than prioritizing energy efficiency or performance, such a scheduler would prioritize consistent allocation of tasks within an appropriate "security domain"—a group of CPUs with the same security characteristics—directly parallel to the "performance domains" that

the Energy Aware Scheduler uses to group CPUs by performance characteristics [64]. A region of code that is not safe for speculation must only be scheduled on a non-speculative core. A region of code that is safe for speculation could always be scheduled on either a speculative or non-speculative core, so the processing resources of the non-speculative cores would still be useful for the purpose of energy efficiency, even when they are not being utilized for the purpose of security.

Developing a new scheduler for a heterogeneous multicore system—with a bifurcation of cores for security rather than performance—is not a trivial task, but the desired features are a relatively minor departure from existing schedulers, and not disruptive to the overall stack of systems software.

3.2.4 Workloads

One level up the system stack from the kernel, modern implementations of virtual machines and containers make use of kernel security features to improve their own security. Randal [99] reviewed the evolution of standard features in the Linux Kernel used by modern virtual machines and containers, such as filesystem, process, IPC, and network namespaces, resource usage limits, access controls, and system call filtering. These security features are applied at the process-level, where each virtual machine or container is a process on the host kernel, and may contain additional processes on the same host kernel or a guest kernel. To take full advantage of speculation-centric heterogeneous multicore systems, the container runtime or virtual machine manager need the ability to declare the security domain of the processes it launches as virtual machines or containers, so the kernel scheduler can appropriately choose a speculative or non-speculative core for the process. One simple way to integrate such a feature into the Linux Kernel would be to add a speculation capability (perhaps `CAP_SYS_SPEC`) that grants permission to run on a speculative core. The Linux Kernel scheduler would then take the speculation capability into account when choosing where to schedule tasks.

Another alternative, which would not require any modification to the Linux Kernel scheduler or capabilities, would be to use the Linux Kernel's existing features for processor affinity and CPU pinning, to restrict a process or thread so that it will only run on a specific core or set of cores. The `taskset` command and `sched_setaffinity` system call set the CPU affinity of a process, while the glibc functions `pthread_setaffinity_np` and `pthread_attr_setaffinity_np` set the CPU affinity of a thread. The existing features for CPU affinity are more manual, and would require defining CPU sets for speculative and non-speculative cores. But, using existing features would make it easier to evaluate an unmodified Linux distribution on a heterogeneous multicore system generated by a default configuration of a framework like Chipyard.

3.3 Evaluation

The greatest advantage of the heterogeneous multicore approach is that it requires less extensive and less disruptive changes to the hardware and software stack, which makes production hardware realistically achievable in the short-term future. By comparison, the demi-speculative ISA approach in Chapter 5 requires substantial changes at the microarchitecture level and throughout the systems software stack. The non-speculative standard ISA approach in Chapter 4 only requires changes at the microarchitecture level,

but developing that non-speculative microarchitecture to production-ready silicon chips for multitenant infrastructures would be a multi-year effort.

One significant disadvantage of the heterogenous multicore approach is the level of granularity in control over speculation. The demi-speculative ISA approach in Chapter 5 has instruction-level granularity—speculation can be enabled and disabled at the level of a single instruction in the instruction stream. The heterogenous multicore approach has process-level or thread-level granularity—speculation can only be enabled and disabled at the process or thread level. So, an entire virtual machine or container—or a process or thread inside a virtual machine or container—may be speculative or non-speculative, but controling speculation down to the level of a single instruction would be impossible. For some use cases the granularity of control may not be important, but as long as speculative pipelining continues to be substantially faster than non-speculative pipelining, there will be a performance advantage to keeping speculation enabled by default, and only disabling it for the smallest possible regions of security critical code.

3.3.1 Performance

For performance evaluation, we use a heterogeneous multicore configuration, called Gluon,[1] combining a non-speculative Rocket core with a speculative BOOM core as a dual-core SoC. The Rocket [8] core is an extensible RISC-V in-order scalar core, that uses branch prediction in the fetch stage, but does not execute instructions speculatively. The BOOM [131] core is a RISC-V out-of-order superscalar core, based on the Rocket core, with a high-performance TAGE [111] branch predictor.

The Rocket and BOOM cores and Gluon multicore were built and executed within the Chipyard framework using the Verlator RTL simulator for a cycle-accurate behavioral model. Evaluating these cores on a baseline set of benchmarks from the RISC-V project [100], the results in Figure 3.1 show that the worst-case performance by IPC for a Gluon multicore is the same as a stand-alone Rocket core. These results are not surprising; when combining heterogeneous cores we can reasonably expect the performance of each workload to depend on which core runs it. Workloads allocated to big cores will run faster and workloads allocated to small cores will run slower.

However, the results do highlight the greatest disadvantage of the heterogenous multicore approach in the context of multitenant infrastructures, which is inflexible hardware resource allocation. When building a large-scale data center, the heterogenous multicore approach would require deciding in advance exactly how many speculative and non-speculative cores to manufacture in every machine in the data center. Getting the upfront allocation decision right involves an impossible level of precision in predicting the kinds of workloads that customers will want to run, and exactly what percentage of speculative versus non-speculative execution they will want. Getting the upfront allocation decision wrong means a massive waste of capital spent on cores that lie idle in production due to a lack of customer demand. A private cloud might get away with downgrading some workloads from big cores to small cores, but a public cloud provider selling CPU time in performance tiers by the hour would quickly lose customers if they arbitrarily downgraded workloads that the customer paid to deploy on big speculative cores, substituting unused

[1]There is no particular significance to "Gluon", we have only adopted names for clarity of exposition. In the domain of physics, a gluon is an elementary particle that binds quarks together. The name also sounds vaguely glue-like, which helps make it memorable for "gluing together" heterogenous cores.

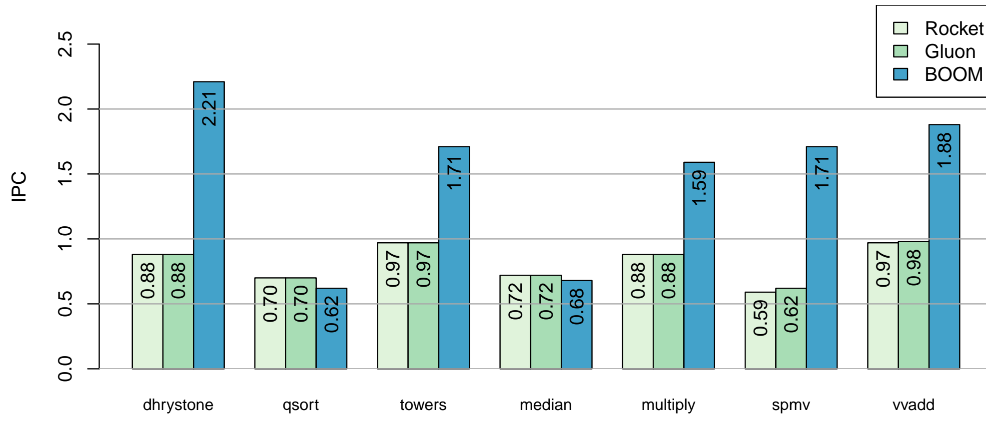

Figure 3.1: Comparing Rocket and BOOM single cores to the Gluon multicore.

small non-speculative cores.

Chapter 4

Out-of-order cores without speculation

Since the speculative execution vulnerabilities were first reported in 2018, it has often been said that we must continue producing superscalar processors based on speculative execution, despite the security risks, because the speculation features are critical for performance [66, 76, 110, 128, 19, 48, 104, 50]. It is true that stripping away all the modern features of superscalar processors and stepping back in time to the exact features of old simple scalar processors would certainly mean returning to the performance levels of those scalar processors. However, the defining characteristics of modern superscalar processors are not speculative execution, they are dynamic multiple issue and dynamic pipeline scheduling [94, p. 328]. In light of the speculative execution vulnerabilities discovered in recent years, it is a worthwhile exercise to consider the logical limits of performance for a modern superscalar processor without speculative execution features. In the contex of multitenant infrastructures, it is also worth considering whether maximizing instruction-level parallelism for a single stream of instructions—which has always been the performance goal of speculative execution—is still the right performance goal for systems that are massively multi-core, multi-threaded, multi-workload, and multi-user.

4.1 Related work

In the early 1970s, before speculative execution became the norm, Riseman and Foster [101] published a thought experiement on the theoretical limits of performance for conditional branches. This work posited a machine with an infinite reorder buffer,[1] infinite functional units, infinite registers with register renaming, and the ability to hold an infinite number of "tentative computational paths" simultaneously for both code paths from each conditional branch (taken or not taken) and discard incorrect paths when the branch conditions were resolved. They observed that the maximum speed on such a machine is attained through the potential to hold an infinite number of conditional branches, which meant that the maximum possible performance improvement was limited by the size of the reorder buffer. They calculated that holding j conditional branches in the pipeline required a reorder buffer large enough to hold least 2^j simultaneous code paths. On the hardware of the time,

[1]They did not call it a "reorder buffer", or settle on any consistent way to refer to it, but the most common were "instruction stack" or "dispatch stack".

they considered a reorder buffer with two slots to be a reasonable size, and ten slots as unusually large, so they rejected the approach as impractical.

This is the theoretical context in which speculative execution was embraced. At a fundamental level, speculative execution is not primarily a performance optimization, it is primarily a space optimization—using less storage space at various stages of the pipeline to achieve roughly the same performance. Speculation allows the pipeline to only hold the instructions for one path of each conditional branch at a time, while ignoring the other path that it predicts as unlikely. But, the space optimization comes with a heavy performance cost when the speculation turns out to be wrong and the instructions for the ignored path have to be fetched, decoded, issued, and executed after the result of the branch condition is known, as if the speculation never happened.

In 2003, Swanson *et al.* [117] observed that the performance benefit of speculation decreases on multithreaded superscalar processors as they increase parallelism through different elements of the pipeline. In their experiments, a non-speculative pipeline with 8 functional units (combined ALU/LSU) performed 12% better than a speculative pipeline with 4 functional units. Increasing the size of the L1 data and instruction caches decreased the performance benefit of speculation from 33% at 16KB, to 24% at 128KB. Increasing the number of threads on the core reduced the performance benefit of speculation from 300% at 1 thread, to 100% at 4 threads, 24% at 8 threads, and 0% at 16 threads. They observed that only heavily loaded servers with many workloads would keep that many threads continuously busy, which means that their results are uniquely applicable to large scale multitenant infrastructures. The performance impacts they observed are also more relevant to modern superscalar processors than they were 20 years ago.

4.2 Feasibility considerations

There are many possible ways to implement the microarchitecture of any given instruction set architecture (ISA). This chapter explores the feasibility of a small but significant variation on existing microarchitecture implementations, evolving superscalar processor techniques forward while eliminating speculation. Chapters 3 and 5 explore alternative approaches that make it possible to disable speculation for specific regions of code, without eliminating it entirely.

The performance evaluation section of this chapter focuses on the RISC-V architecture, but the microarchitecture techniques discussed are applicable to other existing hardware architectures, including x86 or ARM.

4.2.1 Speculative branch instructions

As a foundation, consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor.[2] Like a modern x86, the pipelining approach in this hypothetical microarchitecture uses both dynamic multiple issue and dynamic pipeline scheduling, with out-of-order execution. Figure 4.1 illustrates the essential stages of such a microarchitecture: instructions are fetched, placed in a reorder buffer, queued to reservation stations,

[2]For a specific example, chapter 4, section 11 of Patterson and Hennessy [94] offers a straightforward overview of an x86 microarchitecture, based on an Intel Core i7 920.

dispatched to functional units[3] for execution, and then the commit unit retires instructions, either writing or discarding their results. This illustration abstracts away some details of specific microarchitectures to focus on common elements. For example, microarchitectures for the complex x86 instruction set typically decode the fetched instructions into micro-operations, to simplify the issue, execution, and commit stages, but this extra decoding step is rarely added to microarchitectures for the RISC-V ISA,[4] which is fundamentally designed to be more like the simple micro-operations of x86.

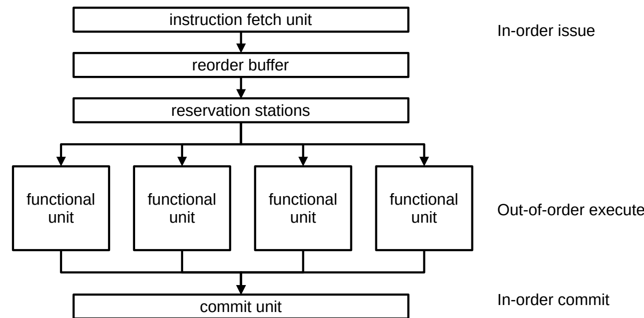

Figure 4.1: Essential components of a dynamically scheduled pipeline. Adapted from Patterson and Hennessy [94], Figure 4.69, p. 330.

First, consider how speculative branch instructions flow through these essential components of a dynamically scheduled pipeline.

4.2.1.1 Fetch

Instruction fetching uses a branch target buffer (BTB) to cache the predicted destination that is most likely for each conditional branch, based on a recent history of executed branches. Various microarchitectures may also use a branch history buffer (BHB), branch history table (BHT), or pattern history table (PHT) in predicting conditional branches, or a return stack buffer (RSB) in predicting return addresses. Rather than waiting for the result of evaluating the branch instruction condition, the instruction fetch unit will continue to fetch instructions along the predicted branch path.

4.2.1.2 Issue

Instruction issue places an entry for each instruction into a reorder buffer, in the order the instructions were fetched. This stage also performs register renaming, mapping the architectural registers (the ones visible in the ISA) onto a larger set of physical registers. Register renaming makes it possible for a sequence of speculatively executed instructions (or unrolled loop instructions) to effectively operate on a temporary virtual register set, which may be discarded if the speculatively predicted branch is later determined to be incorrect. Finally, instruction issue sends instructions to the reservation stations, either copying the operands immediately (if they are available) or copying them later (if the operands depend on the results of other instructions).

[3] Superscalar processors have multiples of each kind of functional unit, including arithmetic logic units (ALU), floating-point units (FPU), load-store units (LSU), and more.

[4] The BOOM [131] RISC-V core does decode some instructions into micro-operations, as an optimization for a special case of data-dependent branches.

4.2.1.3 Execute

The reservation stations queue up instructions and their operands for multiple functional units. The reservation stations buffer each instruction until all its operands are ready and the necessary functional unit is available. The reservation stations dispatch multiple instructions in parallel to multiple functional units in each clock cycle (known as "multiple issue"). The functional unit calculates the result of the operation and sends it to the reorder buffer, as well as to any other reservation stations whose operands depend on the result. Buffering in the reservation stations means that instructions may not be executed in the order they were fetched (known as "out-of-order execution"), because the pipeline tries to avoid hazards and stalls by reordering the instructions (known as "dynamic pipeline scheduling") while maintaining the data flow structure of the program.

4.2.1.4 Commit

The commit unit uses the instruction entry in the reorder buffer to hold the results of the instruction execution until it determines that any speculated results were speculated correctly, and then marks the instruction entry as complete. The commit unit processes the reorder buffer in the order the instructions were fetched, so when the instruction at the head of the reorder buffer is marked as complete, it will perform any pending register writes or memory stores, and then remove the instruction entry from the reorder buffer. Alternatively, if the commit unit determines that the speculation was incorrect, it will discard the result in the reorder buffer, and remove the instruction entry. This approach of preserving the instruction fetch order in the commit process, called "in-order commit", allows the out-of-order pipeline to preserve the appearance of operating like a simple in-order pipeline. The commit unit only stores results to memory after any speculatively predicted branches that the store instruction depends on have been determined as correct.

4.2.1.5 Discussion

A number of different real-world microarchitectures follow this general model of speculative pipelining. For example, Figures 4.2, 4.3, and 4.4 are modern examples of superscalar, speculative x86, ARM, and RISC-V microarchitectures, and though implementation details differ, even a cursory examination reveals that they are all designed using the same fundamental pipeline components from Figure 4.1. The third generation of the Berkeley RISC-V Out-of-Order Machine (BOOM) in Figure 4.2, shows instruction fetch near the top center of the figure, branch predictor near the top left, reorder buffer near the center, feeding into reservation stations labeled "Distributed Scheduler", which feed into functional units labeled "EUs", and finally retirement/commit is annotated on the reorder buffer near the center. The ARM Neoverse N1 microarchitecture in Figure 4.3,[5] shows instruction fetch near the top center of the figure, branch predictor at the top left, reorder buffer near the center, feeding into reservation stations labled "Issue" and "Queue", which feed into functional units labeled "EUs", and finally retirement/commit is annotated on the reorder buffer near the center. The Intel Sunny Cove microarchitecture in Figure 4.4 is a single core within Intel's Ice Lake server processors,[6] and shows instruction

[5] The AWS Graviton processor custom-built for Amazon EC2 is based on the ARM Neoverse microarchitecture.

[6] The Ice Lake server processors are also branded as 10th generation Xeon processors, while the Ice Lake client processors are also branded as 10th Generation Core i3, i5, and i7 processors.

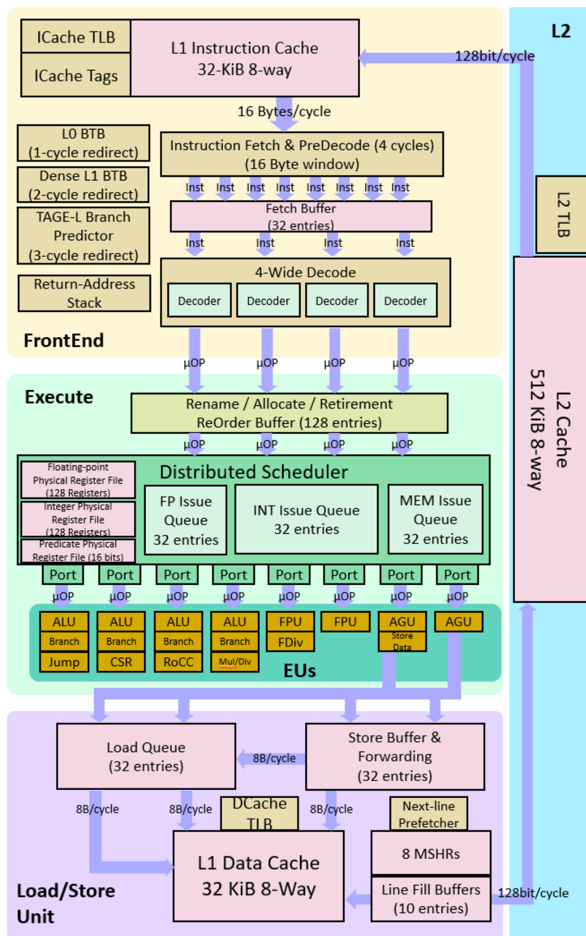

Figure 4.2: RISC-V BOOM microarchitecture. Reprinted from Zhao *et al.* [131], Figure 1.

fetch near the top center of the figure, branch predictor near the top left, reorder buffer near the center, with register alias tables near the center left, feeding into reservation stations labeled "Scheduler", which feed into the functional units labeled "EUs", and finally retirement/commit operates on the reorder buffer near the center right.

The critical security risk in this speculative implementation of branch instructions—and in any microarchitectures that follow a similar pattern—lies in the first component listed above, when instruction fetching predicts a particular result for a conditional branch, and then proceeds to speculatively fetch, issue, and execute instructions on that branch. Spectre-type attacks use techniques to mistrain the branch prediction by feeding it false history, thus "poisoning" the branch predictor state of the microarchitecture [19, pp. 4-6]. Meltdown-type attacks exploit the fact that exceptions raised while executing speculatively are suspended until the faulting instruction is retired by the commit unit, which leaves a window of opportunity open for access to unauthorized results that should have been intercepted by the exception [19, pp. 7-10].

4.2.2 Non-speculative branch instructions

One possible way to entirely avoid the security risk of speculative branch instructions is to replace them with non-speculative branch instructions, which do not participate in branch prediction. In a traditional, scalar, in-order microarchitecture without branch prediction,

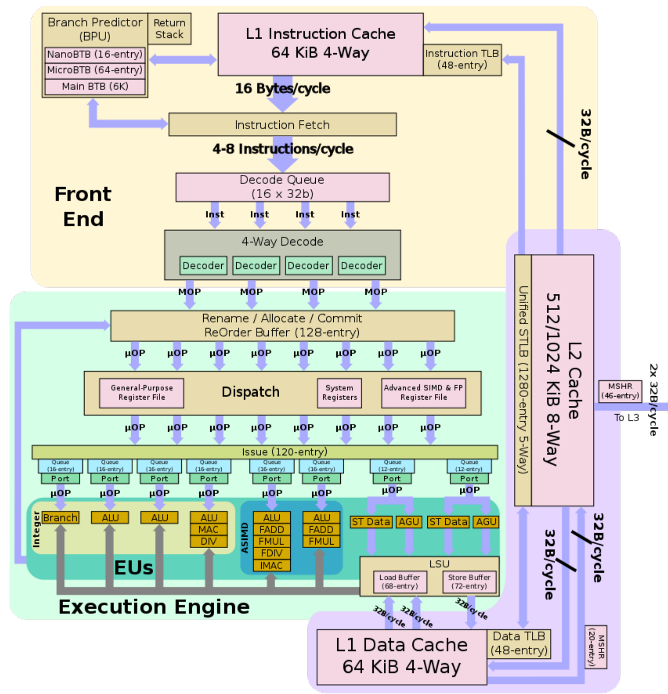

Figure 4.3: ARM Neoverse N1 microarchitecture. Reprinted from Schor [107].

non-speculative branch instructions stall the pipeline, so the pipeline does not fetch or issue any instructions from either path of the branch until the branch condition is resolved. A more nuanced approach to non-speculative branch instructions, in the context of dynamic pipeline scheduling, is to design the system so that it fetches and issues instructions from both paths of the branch. The approach is not new or particularly radical—beyond the theoretical work by Riseman and Foster [101] in the 1970s, the IBM 370/168 and IBM 3033 used similar techniques [71], though the pipelining techniques of the time were a poor fit for fetching multiple independent instruction streams. In this dissertation, we call this more nuanced approach to non-speculative branch instructions *superpositional* pipelining, by analogy to quantum superpositions, which hold multiple simultaneous states of reality as potentially true until they reduce to one true state.[7]

4.2.2.1 Fetch

Instruction fetching for superpositional, non-speculative branch instructions does not read from or update a branch target buffer, branch history buffer, branch history table, pattern history table, or return stack buffer. This means both that branch instructions in malicious code cannot mistrain a branch predictor, and that branch instructions in secure code are not vulunerable to mistraining attacks.

[7]The idea is not in any way related to the field of quantum computing, which uses actual physical superposition and entanglement (as in, quantum mechanics) instead of binary bits to perform computations. However, the idea is similar to the concept of phi nodes from LLVM's intermediate representation, as a resolution point of two variant branches for static single assignment (SSA), and the name "phi" is also a concept from quantum mechanics.

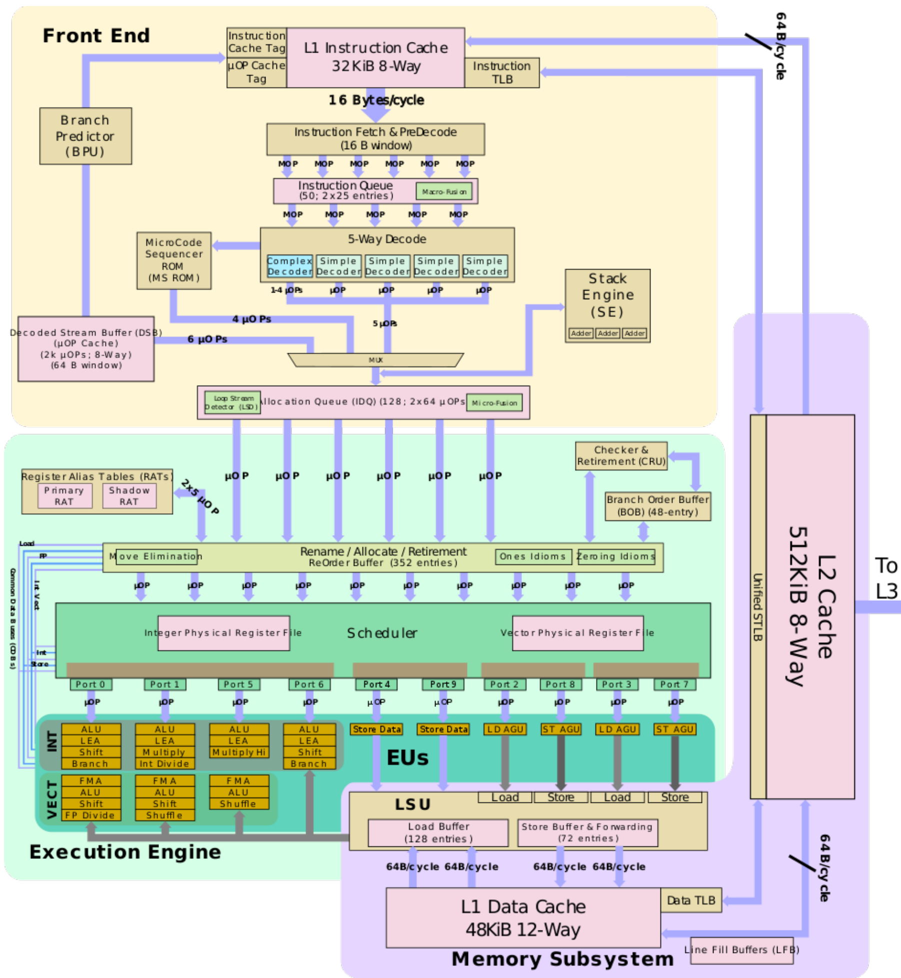

Figure 4.4: Intel Sunny Cove microarchitecture. Reprinted from Schor [108].

4.2.2.2 Issue

The superpositional pipeline places instruction entries for both code paths from a conditional branch in the reorder buffer, tagging the entries with a control dependency on the result of the conditional branch instruction, and renaming registers in each branch path so the two alternative sets of instructions do not use the same physical registers.[8] The pipeline sends the instructions from both branch paths to the reservation stations, preserving the control dependency tag.

4.2.2.3 Execute

The reservation stations buffer the instructions from both branch paths in the usual way, however they treat the control dependency tag similarly to a data dependency of waiting for operands. No instruction from either branch path is dispatched to a functional unit until the result of the non-speculative conditional branch instruction is known, so there is no “speculative execution” of any instruction. However, since the instructions from both conditional branch paths have already been fetched and issued, the instructions on the

[8]This step is similar enough to the microarchitecture technique of loop unrolling that it might deserve to be called “branch unrolling”.

selected branch path can be dispatched to the functional unit on the next clock cycle after the result of the conditional branch has been calculated. This is far more rapid than if the entire fetch and issue process was delayed until after the branch condition is evaluated. The instructions on the rejected branch path are discarded by the reservation stations, and tagged for discard in the reorder buffer (the instruction entries are effectively re-written as completed no-op entries).

4.2.2.4 Commit

The commit unit treats instructions from the rejected conditional branch path in the same way as incorrectly speculated instructions, freeing up the renamed registers and removing the instruction entry from the reorder buffer. The commit unit still processes the reorder buffer in the order the instructions were fetched. The instructions from either path on the conditional branch might have been fetched first, but it ultimately does not matter which branch path is first in the reorder buffer, since the instructions from the rejected branch path will all be discarded (in the order they were fetched), and the instructions from the selected branch path will all be committed (in the order they were fetched). No instructions on either branch path will ever produce a result until after the result of the conditional branch is known.

4.2.2.5 Discussion

If we altered Figures 4.2, 4.3, and 4.4 for a superpositional branch pipeline, the primary visible change would be removing the branch predictor components. The size of some components of the pipeline—such as the L1 instruction cache, fetch width, fetch buffer, decode width, reorder buffer, and the reservation stations—would likely need to be increased to maintain the same instruction throughput despite discarding instructions on superpositional branch code paths. On the whole, however, superpositional branch pipelines preserve the microarchitectural state and functionality of the existing superscalar core design, requiring only minimal changes to the internal behavior of instruction fetch, issue, execute, and commit.

A superpositional branch pipeline does more work than a speculative branch pipeline, because it has to fetch and issue instructions from both branch paths, instead of fetching and issuing instructions from the predicted branch path and ignoring the other branch path. However, the extra work of the superpositional pipeline does not necessarily imply slower throughput of instructions with dynamic pipeline scheduling, since the instructions on the two branch paths are independent, and can be fetched and issued in parallel. A superpositional branch will always be faster than a mispredicted speculative branch, since the pipeline can immediately proceed with executing the already issued instructions from the correct branch path, instead of doing all the work of executing the misspeculated branch path instructions, discarding them, and then starting the fetch for the correct branch path instructions after the result of the conditional branch is known. So, it is likely that a superpositional branch pipeline will tend to perform no better than the best case of all correctly predicted speculative branches, but will tend to perform better than the worst case of all mispredicted speculative branches. In practical terms, the relative performance of a superpositional branch pipeline compared to a speculative branch pipeline will also depend on the actual nature of the code being run—specifically on whether the code is dominated by conditional branches that are consistently predictable or change results

frequently, on whether branches have long or short sequences of instructions on their code paths, and on how extensively the surrounding instructions outside conditional branch code paths have a data dependency on the results of the instructions inside conditional branch code paths.

In Section 4.3.1, we will explore the performance potential of superpositional branch pipelines further with RTL and FPGA simulation.

4.2.3 Speculative memory load instructions

As we observed in Section 2.4, speculative branch instructions are one microarchitectural root cause of the speculative execution vulnerabilities, but the other root cause is speculative memory load instructions. Continuing with the foundation in Section 4.2.1, we consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor. Speculative memory load instructions flow through the same essential components of a dynamically scheduled pipeline as speculative branch instructions.

4.2.3.1 Fetch

Instruction fetching may encounter an ordinary memory load instruction inside a predicted branch, in which case it will fetch (as well as issue and execute) that memory load instruction in exactly the same way it speculatively fetches all other instructions on the predicted branch. Outside of any branch code path, another way that memory load instructions may be speculatively executed is through a memory disambiguator that predicts which memory loads do not have a Store To Load (STL) dependency on any prior store instructions, so a memory load may be executed speculatively and out-of-order, before prior stores to the same address have been completed.

4.2.3.2 Issue

Instruction issue places an entry for each memory load instruction into the reorder buffer, performs register renaming, and dispatches the instruction to the reservation stations, the same as in Section 4.2.1.2.

4.2.3.3 Execute

The reservation stations buffer memory load instructions until their operands are ready and the necessary functional unit (a load-store unit) is available. The functional unit executes the memory load operation, and sends the result (the value loaded from memory) to the instruction entry in the reorder buffer, and also to any other reservation stations whose operands depend on the result. If the memory load was executed speculatively, other instructions with a data dependency on the memory load may also be executed speculatively.

4.2.3.4 Commit

The commit unit uses the instruction entry in the reorder buffer to hold the results of the memory load instruction (the value loaded from memory), until it determines that the speculated memory load was speculated correctly. If the commit unit determines that the speculation was correct, it marks the instruction entry in the reorder buffer

as complete, performs any pending register writes or memory stores, and removes the instruction entry from the reorder buffer. If the memory load instruction was speculated because of branch prediction, the determination of correctness is based on whether the branch path was predicted correctly, and a misprediction will remove the instruction entry from the reorder buffer. If the memory load instruction was speculated because of the memory disambiguator, the determination of correctness is based on whether the prediction that all prior stores were complete was correct, and a misprediction will discard the result in the reorder buffer, but will have to execute the memory load operation all over again to get the correct result, and also re-execute any other instructions that had a data dependency on the result of the memory load.

4.2.3.5 Discussion

The critical security risk in this speculative implementation of memory load instructions, and in any microarchitectures that use similar techniques, lies in the third component listed above, when the load-store unit executes the memory load operation. In modern architectures, a memory load operation is not a passive read directly from physical memory, it is a complex operation with cascading side-effects through multiple levels of memory cache, DRAM buffers, and translation lookaside buffers (TLB), and also has the potential to trigger exceptions. Some Spectre-type attacks—notably, the speculative store bypass variant reported by Horn [53]—use techniques to mistrain the memory disambiguator, so it incorrectly speculates a memory load operation, leaving behind microarchitectural traces of a stale value that should have been overwritten by a prior store, and then access the stale value through cache-based side-channels [19, p. 6]. Meltdown-type attacks exploit the fact that exceptions are only raised after a faulting instruction is finally committed, so exceptions raised by a memory load instruction[9] are suspended long enough for the attack to access the speculatively loaded value through microarchitectural covert channels [19, pp. 6-9].

4.2.4 Non-speculative memory load instructions

One possible way to entirely avoid the security risk of speculative memory load instructions is to replace them with non-speculative memory load instructions. However, while the limiting factor for branches is parallelism in the pipeline itself, the limiting factor for memory loads is unavoidable memory latency through the data cache hierarchy and DRAM, which means that the performance impact of eliminating speculation will be greater for memory loads than for branches. We also discuss an alternative approach to memory load instructions in Section 4.2.5, which are partially speculative, but in a restricted way to avoid leaking microarchitectural traces.

4.2.4.1 Fetch

Instruction fetching for non-speculative memory load instructions does not use speculative predictions from the memory disambiguator, and does not participate in training the memory disambiguator for any future speculative predictions. It does, however, use the memory disambiguator for tracking when all prior stores to the same address have

[9] Such as a page fault, a general protection fault, or a device-not-available exception.

been completed—resolving all Store To Load (STL) dependencies for the memory load instruction—to determine when the memory load is safe to execute non-speculatively.

4.2.4.2 Issue

Non-speculative memory load instructions are fetched, issued with an entry in the reorder buffer, and dispatched to the reservation stations, the same as in Section 4.2.2.2. Memory load instructions are tagged with a control dependency by the memory disambiguator, so they cannot execute before all prior stores to the same address have been completed. If a memory load instruction is fetched and issued as part of a superpositional branch code path, it is also tagged with a control dependency on the result of the conditional branch instruction.

4.2.4.3 Execute

The reservation stations preserve control dependency tags, treating them similarly to the data dependencies of waiting for operands. They will not dispatch any memory load instruction to the load-store unit until all prior stores to the same address have been completed. When the memory load is on a conditional branch code path, the reservation stations also will not dispatch the memory load instruction to the load-store unit until the result of the conditional branch is known. Since memory loads are only executed non-speculatively, any exceptions raised by the memory load instruction are raised immediately, so there is no period of transient execution to allow access to unauthorized memory load results through covert side channels.

4.2.4.4 Commit

Since non-speculative memory load instructions are never speculatively executed, the memory load instruction entry in the reorder buffer is always marked as complete after it receives a result from the load-store unit.

4.2.4.5 Discussion

Non-speculative memory loads with no hardware prefetching will always be slower than the best case where a speculative memory load is correctly predicted through the memory disambiguator. In the worst case of branch misprediction, non-speculative memory loads avoid the cost of loading a value through multiple layers of cache that will only be discarded, while still giving the pipeline flexibility to dynamically schedule the memory load instruction out-of-order (without executing it speculatively). In the worst case of the memory disambiguator mispredicting, non-speculative memory loads avoid the cost of re-executing the memory load operation and any other instructions that depended on the value it loaded.

In the case where a non-speculative memory load is an L1 data cache hit, simple out-of-order execution without speculation may be able to hide any performance penalty of the memory load, because it can execute the memory load instruction as soon as any control dependencies on non-speculative branch instructions or prior stores are completed. These control dependencies may be resolved and the memory load executed long before any instructions with a data dependency on the memory load are ready to execute, even without speculation. In the case where the non-speculative memory load is an L1 data

cache miss, however, the performance penalty of the memory load may be prohibitive. Zhao *et al.* [131] estimate that on the BOOM RISC-V microarchitecture, the performance penalty for an L1 fetch is only 10 cycles, while the performance penalty for an L3 fetch is on the order of 50 cycles, which would require a lookahead of 200 instructions on a 4-wide BOOM pipeline, exceeding the capacity they designed for the reorder buffer.

4.2.5 Speculation buffers for memory load instructions

A hardware prefetching technique prototyped by Gonzalez *et al.* [48] in 2019 and independently by Ainsworth and Jones [3] in 2020 could be combined with superpositional branch pipelining, to speed up memory loads without leaving microarchitectural traces. The technique is partially speculative, in that it does read from L1-L3 data cache or DRAM before any control dependencies (on branch instructions or prior stores) are resolved, however it holds the result of these memory loads in a special cache (the "L0 speculation buffer" or "L0 filter cache"), which is only accessible to the memory load instruction, and does not update the L1-L3 cache in any way until after the memory load instruction is marked complete by the commit unit. Ainsworth and Jones [3] demonstrated the performance impact for applying the technique to an 8-wide ARM core in the gem5 simulator was a 4% performance penalty in the SPEC CPU2006 benchmarks and a 5% performance gain in the Parsec benchmarks.

Further modifying the technique proposed by Gonzalez *et al.* and Ainsworth and Jones to more closely fit superpositional pipelines, the speculation buffer could cache multiple variants of the value, with some variants fetched from data cache or DRAM, and other variants set by stores to the same memory address. A memory load would only be granted access to the most recent variant—effectively serving as a tightly restricted form of data forwarding[10]—and the memory disambiguator could discard earlier variants as soon as it determines that all memory loads which should have access to a particular variant (because they are not blocked by a subsequent store to the same address) have already received the value. Adding a speculation buffer to a core does require more storage space in the implementation, but the trade-off is a substantial improvement in isolation within the microarchitecture.

At a conceptual level—continuing the analogy of superpositions holding multiple states of reality as potentially true until it can be resolved which one is true—this approach to memory load instructions using speculation buffers is more "superpositional" than the non-speculative memory load instructions in Section 4.2.4.

We did not implement either non-speculative memory loads or speculation buffers as part of the work of this dissertation, but may do so in future work.[11]

4.2.6 Thread-level parallelism

Modern hardware architectures like the x86 do not limit a single hardware core to running a single process, instead they use the abstraction of *threads* to share a core between multiple tasks. Multithreading is important in the context of desktop and mobile hardware—which tend to have a relatively small number of cores—but it is absolutely essential in the context

[10] The technique is similar enough to register renaming that it might even be called "cache renaming".

[11] In fact, we started working on an implementation in the gem5 simulator, but gem5's support for the RISC-V architecture is new and proved too unstable to deliver reliable results, even without any modifications to the cache hierarchy or core implementations.

of multitenant server architectures—where the cloud/container business model depends on the ability to overcommit CPU resources, running more (mostly inactive) virtual machines or containers than the machine has cores. Sharing cores yields a substantial performance gain, because idle compute resources from one thread can be used for another thread. Even combining all the most advanced techniques of instruction-level parallelism,[12] a single stream of instructions for a single thread will still regularly block on data hazards or control hazards when there are not enough instructions ready to be dispatched to a functional unit in a clock cycle to fill all the available functional units, leaving compute resources idle.

Unfortunately, the performance gain of multithreading compounds the security risk of speculative execution, because it means that the microarchitectural state exposed by branches and memory loads is shared between multiple threads, which may be running completely unrelated code from unrelated virtual machines or containers for unrelated users. Percival [95] provides an early but comprehensive exploration of the risks inherent in combining simultaneous multithreading with speculative superscalar pipelining, multilevel memory caches, and hardware prefetching. Ge *et al.* [43] more broadly survey the microarchitecural side-channel attacks enabled by multithreading, multilevel memory caches, and co-resident virtual machines. Escouteloup *et al.* [35] propose a collection of fundamental security principles and recommendations for the design of future hardware in light of these microarchitectural risks.

Existing techniques to improve the security of multithreading with speculative superscalar pipelines—such as tracking a unique thread ID for every instruction through all stages of the pipeline and segmenting caches and other microarchitectural state per-thread—are equally effective with non-speculative superpositional pipelines. On each clock cycle, multiple instructions from multiple different threads may be fetched, entered in the reorder buffer, issued to the reservation stations, dispatched to functional units, and marked as complete by the commit unit. Although they share the same microarchitecture hardware, instructions from one thread must not have access to another thread's microarchitectural state. Superpositional pipelines have the potential to provide a stronger guarantee of the required isolation between threads than speculative pipelines, because they never create the lingering traces of mispeculated microarchitechural state that are exploited by the speculative execution vulnerabilities.

4.3 Evaluation

Replacing speculative pipelining with superpositional pipelining makes it possible to eliminate the speculative execution vulnerabilities, in a way that is invisible outside the microarchitecture. The approach does not require any changes to the ISA or system software, so it is not disruptive to existing software stacks, and preserves portability between different processors with the same ISA, even when one processor has a speculative microarchitecture and another has a non-speculative microarchitecture.

From a security perspective, the approach in this chapter—a standard ISA with a non-speculative microarchitecture—has the advantage of entirely eliminating the risk of speculative execution, while the heterogeneous multicore approach in Chapter 3 or the

[12] Such as dynamic multiple issue, dynamic pipline scheduling, hardware prefetching, speculative execution, etc.

demi-speculative ISA approach in Chapter 5 only provide the ability to partially disable speculation.

From a performance perspective, the standard ISA approach in this chapter may have an advantage over the demi-speculative ISA approach in Chapter 5 because it replaces one form of complexity in the pipeline (speculation) with a different but roughly equivalent form of complexity (superposition), rather than combining both forms of complexity into a more complex demi-speculative ISA. The standard ISA approach in this chapter also has a resource allocation advantage over the heterogeneous multicore approach in Chapter 3, because every core on the server runs an identical standard ISA, so the host retains full flexibility to control scheduling between cores in response to demand, rather than restricting certain workloads to specialized non-speculative cores.

One potential disadvantage of the standard ISA approach in this chapter is that the microarchitecture is exclusively dedicated to non-speculative pipelining, and has no option to use speculation even in regions of non-critical code where it might be safe to speculate. For the immediate future, despite the security risks, speculation is a solid bet for improving performance, and as long as that continues to be true, it is worthwhile to explore approaches that only partially disable speculation, as in Chapters 3 and 5.

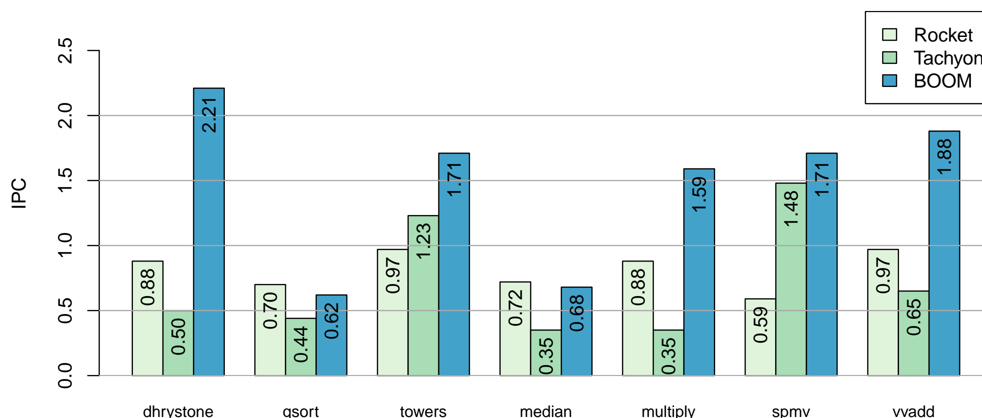

Figure 4.5: Comparing Rocket and BOOM cores to the Tachyon core.

4.3.1 Performance

For performance evaluation, we use a modified version of the BOOM core, called Tachyon,[13] which removes the branch prediction features. The Tachyon core is not a complete implementation of superpositional branches and memory loads, it is only an out-of-order core with non-speculative branches, but even this limited implementation provides some insights into the performance potential of superpositional pipelines. The Rocket, BOOM, and Tachyon cores were built and run within the Chipyard framework using the Verilator RTL simulator for a cycle-accurate behavioral model, and using the FireSim [61] simulation platform for cycle-exact microarchitectural simulation deployed on AWS F1 FPGAs.

[13]Like Gluon, there is no particular significance to "Tachyon", we have only adopted names for clarity of exposition. In the domain of physics, a tachyon is a hypothetical particle that violates the laws of causality by traveling faster than light. Tachyons have been popular in science fiction since they were theorized in the 1960s, but most modern physicists will tell you they cannot exist.

4.3.1.1 RTL simulation

Evaluating the Rocket, BOOM, and Tachyon cores in Verilator on a baseline set of benchmarks from the RISC-V project [100], the results in Figure 4.5 show that for most benchmarks the Tachyon core performs worse by IPC than the Rocket core. This result meets expectations—the Rocket core should perform better than the Tachyon core, because the Rocket core uses branch prediction as an optimization in the fetch stage, while the Tachyon core has removed branch prediction features from the BOOM core. The outliers in Figure 4.5 are the `towers` and `spmv` benchmarks, where the Tachyon core performs better than the Rocket core, though not as well as the BOOM core.

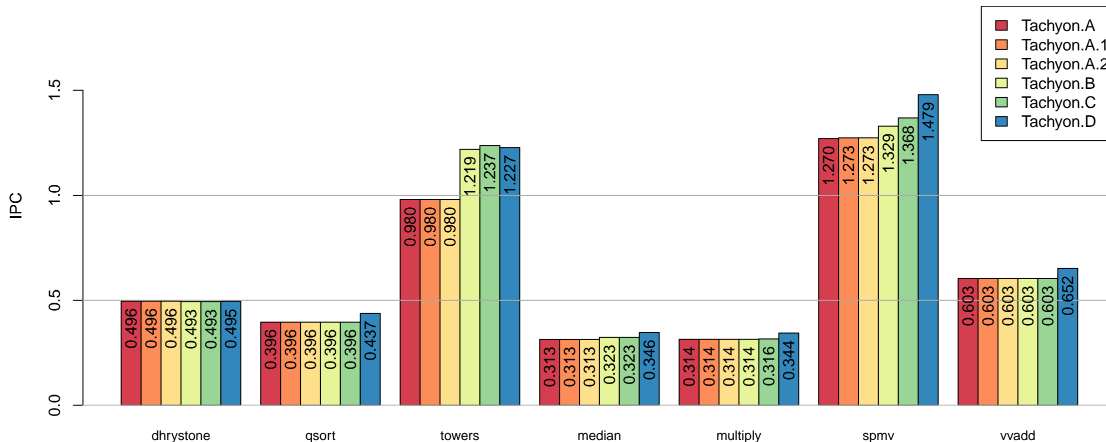

Figure 4.6: Comparing a series of variations on the Tachyon core.

Digging more deeply into the performance results, Figure 4.6 tightens the focus, comparing a series of small variations on the Tachyon core with the characteristics described in Table 4.1. For all benchmarks except `towers` and `spmv`, the performance by IPC remains relatively constant as long as the fetch width is kept at 8, even if the decode/dispatch width is increased from 3 (in Tachyon.A) to 4 (in Tachyon.B) or 5 (in Tachyon.C), and even if the reorder buffer and fetch buffer are doubled in size (in Tachyon.A.1 and Tachyon.A.2). The performance by IPC for most of the benchmarks only begins to improve when the fetch width is increased from 8 to 16 (in Tachyon.D), which again indicates that the performance bottleneck in these benchmarks lies in the fetch stage, because the Tachyon cores do not use branch prediction.

Table 4.1: Characteristics of Tachyon variations.

Variant	Fetch Width	Decode Width	Fetch Buffer	ROB	ALUs	FPUs	LSUs
Tachyon.A/ BOOM.A	8	3	24	96	3	1	1
Tachyon.A.1	8	3	24	255	3	1	1
Tachyon.A.2	8	3	48	255	3	1	1

continued on next page...

Table 4.1 – *continued from previous page*

Variant	Fetch Width	Decode Width	Fetch Buffer	ROB	ALUs	FPUs	LSUs
Tachyon.B/ BOOM.B	8	4	32	128	4	2	2
Tachyon.C/ BOOM.C	8	5	40	130	5	2	2
Tachyon.D	16	8	64	256	8	4	2

However, Figure 4.6 raises a question about why the `towers` and `spmv` benchmarks show such different results. These benchmarks perform better by IPC than the Rocket core (in Figure 4.5), and also generally show a stepwise improvement in performance by IPC as the decode/dispatch width of the Tachyon core is gradually increased (in Figure 4.6), instead of showing a bottleneck on fetch width. The key to the difference is a microarchitecture feature shared by the BOOM and Tachyon cores, but not the Rocket core—an alternative to branch prediction known as short forward branch optimizations. Zhao *et al.* [131, p. 4] describe the feature in more detail, but fundamentally the feature is a simple form of superpositional branching, which is only applied to data-dependent branches within a short basic block. Instead of holding up the fetch stage waiting for the result of evaluating the branch condition, the pipeline fetches and decodes all the instructions that might be executed depending on the result of the branch condition. It transforms the conditional branch instruction into a micro-op that sets a predicate flag, and transforms the instructions that depend on the conditional branch into "conditional execute" micro-ops that determine whether they should execute based on the predicate flag.

Zhao *et al.* [131, p. 4] give the following code example to illustrate short forward branches, in a loop to find the maximum value in an array. The C source code:

```
int max = 0;
int maxid = -1;
for (i = 0; i > n; i++) {
  if (x[i] >= max) {
    max = x[i];
    maxid = i;
  }
}
```

Is compiled to the RISC-V assembly code:

```
loop:
  lw   x2, 0(a0)
  bge  x1, x2, skip
  mv   x1, x2
  mv   a1, t0
skip:
  addi a0, a0, 4
  addi t0, t0, 0x1
  j    loop
```

But the decoded micro-ops replace the `bge` (branch if greater than or equal) branch instruction with a `set.ge` micro-op and replace the potentially skipped `mv` instructions before the `skip` label with conditionally executed `p.mv` micro-ops:

```
loop:
  lw      x2, 0(a0)
  set.bge x1, x2
  p.mv    x1, x2
  p.mv    a1, t0

  addi    a0, a0, 4
  addi    t0, t0, 0x1
  j       loop
```

The specific implementation of short forward branch optimizations in the BOOM core—writing the result of the branch micro-op to a renamed predicate register file—is not relevant to the discussion in this chapter. What is relevant, is that the performance advantages the BOOM core gains for branches using this alternative to branch prediction persist in the Tachyon core. The compiled RISC-V assembly from the `towers` and `spmv` benchmarks exhibit the behavior of data-dependent branches in short basic blocks, which triggers the short forward branch optimizations—in nested loops for `spmv`, and in recursion for `towers`.

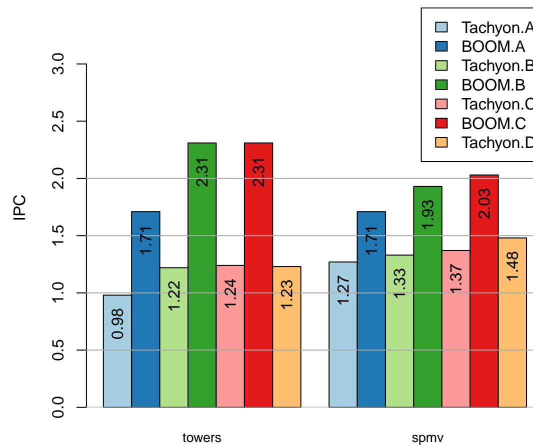

Figure 4.7: Equivalent variations on the Tachyon and BOOM cores.

Comparing the Tachyon core variations to equivalent BOOM core variations in Table 4.1, Figure 4.7 shows that the performance of the Tachyon cores improves in proportion to the performance of the BOOM cores.[14] The `spmv` benchmark consistently shows the BOOM cores performing about 30% better than the equivalently sized Tachyon core, so as Section 4.2.2 anticipated, the trade-off for removing the branch predictor is that an increase in fetch and issue stage features can offset the performance penalty of eliminating

[14] It was not possible to build a BOOM.D core with a fetch width of 16 and a decode/dispatch width of 8 to compare to the Tachyon.D core, because BOOM's TAGE branch predictor configures a hard limit against building such a large core.

speculation. To put this in perspective, applying the mitigations for only the L1TF variant of the speculative execution vulnerabilities has as much as a 31% performance penalty, and each additional mitigation for each variant adds its own performance penalty [19, p. 16].[15] A more complete implementation of superpositional branching has the potential to narrow the performance gap between the Tachyon and BOOM cores, by freeing more branches from the fetch stage bottleneck. The `towers` benchmark shows a performance cap at a decode/dispatch width of 4 in both BOOM.B and Tachyon.B—with no performance improvements for the Boom.C, Tachyon.C, or Tachyon.D variations—which is a sign of some other bottleneck unrelated to branch prediction (though possibly related to recursion). It is an open question is whether this trade-off between branch prediction and fetch and issue stage features is viable from a manufacturing cost perspective—whether the die space saved by dropping the branch predictor adequately compensates for the die space consumed by the increased fetch width, decode/dispatch width, fetch buffer, and reorder buffer.

4.3.1.2 FPGA simulation

We evaluated the Rocket, BOOM, and Tachyon cores simulated with FireSim on AWS F1 FPGAs, using the SPEC CPU2017 intspeed benchmarks. The FireSim simulations ran at a host frequency of 65MHz on the FPGAs, and modeled the system running at 1GHz, configured as single-core systems with 512KB L2, 4MB simulated L3, and 16GB DRAM, and with 32KB L1I, 32KB L1D on the BOOM and Tachyon cores, but 16KB L1I, 16KB L1D on the Rocket core. The SPEC benchmarks were compiled with `gcc`, with `-O3` optimizations. The SPEC CPU benchmark suite scores are a measure of normalized execution time, which depends on CPU performance by IPC, but also depends on the performance of other components in the system such as the cache and memory hierarchy, number of cores, number of threads, and clock speed. This means that the SPEC CPU score is a more comprehensive measure of an overall system than IPC alone, but it also means that direct comparison of the score results is most useful when the systems being compared are similar.

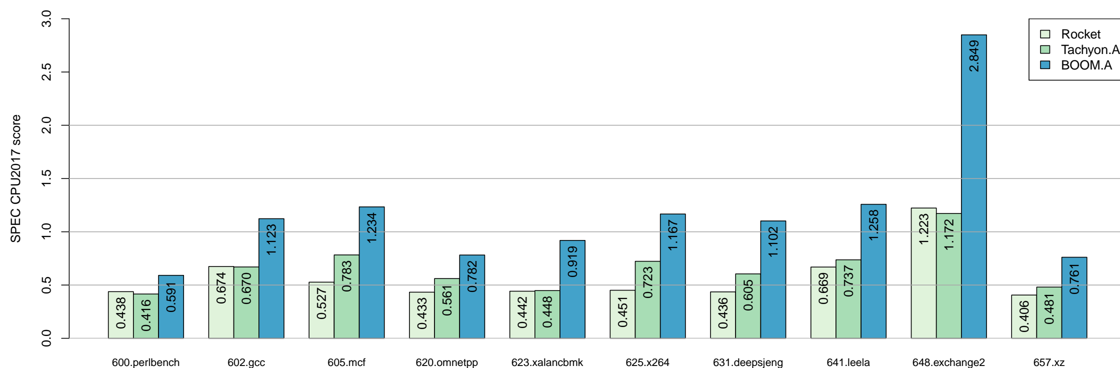

Figure 4.8: Rocket, BOOM, and Tachyon on SPEC CPU 2017 benchmarks.

Evaluating the Rocket, BOOM, and Tachyon cores in FPGA simulation on the SPEC

[15] Large public cloud providers have privately mentioned that the effect of deploying all known and relevant current mitigations for the speculative execution vulnerablities in production is in the range of a 50% performance penalty, and still cannot fully protect against the vulnerabilities.

CPU 2017 intspeed benchmarks, the results in Figure 4.8 show a pattern of relative performance between the cores similar to the RISC-V benchmarks in Figures 4.5 and 4.6, though the Tachyon core tends to perform nearly as well or better than the Rocket core. The Tachyon.A core performs slightly worse than the Rocket core on the `perlbench`, `gcc`, and `exchange2` benchmarks, and better than the Rocket core on the `mcf`, `omnetpp`, `xalancbmk`, `x264`, `deepsjeng`, `leela`, and `xz` benchmarks. Compared to the BOOM.A core, eliminating branch prediction on the Tachyon.A core has about a 30% performance penalty on the `perlbench` and `omnetpp` benchmarks, about a 40% performance penalty on the `gcc`, `mcf`, `x264`, `leela`, and `xz` benchmarks, about a 50% performance penalty on the `xalancbmk`, `deepsjeng` benchmarks, and about a 60% performance penalty on the `exchange2` benchmark.[16]

4.3.1.3 Compared to x86 and ARM

To place these results in the broader context of multitenant infrastructure deployments, Figure 4.9 shows that the performance of the BOOM core by IPC is comparable to production hardware running today. On systems with radically different cache, memory, core, and clock speed configurations, measuring by IPC gives a better sense of the potential throughput of the microarchitecture than measuring by normalized execution time, but it is important to note that the different ISAs affect the IPC results, because the benchmarks compile down to a different number of instructions for RISC-V, ARM, and x86. The Graviton1 in Figure 4.9 is a 1st generation AWS Graviton core (based on ARM Cortex A72), configured as an AWS `a1.metal` instance with 16 vCPUs (16 threads) at 2.3GHz and 32GB memory. The Graviton2 is a 2nd generation AWS Graviton core (based on ARM Neoverse N1), configured as an AWS `m6g.metal` instance with 64 vCPUs (64 threads) at 2.5GHz and 256GB memory. The Xeon is an Intel Xeon Platinum 8175M, configured as an AWS `m5.metal` instance with 96 vCPUs (192 threads) at 3.1GHz and 384GB memory. The Ryzen is an AMD Ryzen Threadripper 3990X, on a local bare metal machine with 64 cores (128 threads) at 2.9GHz and 256GB memory.[17]

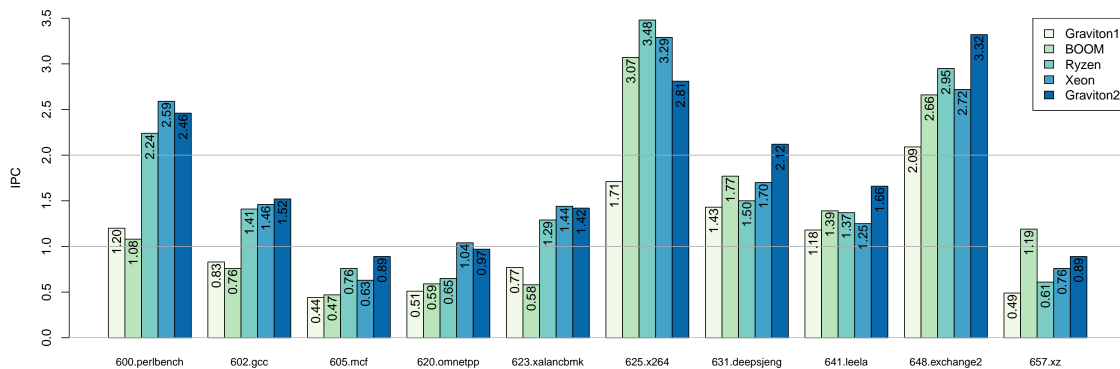

Figure 4.9: BOOM compared to ARM, Intel, and AMD cores.

[16]The Tachyon.B, Tachyon.C, and Tachyon.D variations were too large and complex for FireSim to build them for the AWS F1 FPGAs, so we were not able to measure whether increasing the size of fetch and issue stage components improved performance on the FPGA simulations.

[17]The Ryzen is a desktop processor rather than a server processor, but we included it as a bare metal baseline, because the AWS "bare metal" Xeon and Graviton instances actually run in the lightweight Nitro hypervisor.

The BOOM core consistently performs by IPC nearly as well as or better than the Graviton1, outperforms the Graviton2 in the `x264` and `xz` benchmarks, and outperforms the Ryzen and Xeon in the `deepsjeng`, `leela`, and `xz` benchmarks.

Chapter 5

Demi-speculative features within a single core

The microarchitectural features that make the speculative execution vulnerabilities possible are integral to the performance potential of modern hardware architectures. Entirely eliminating speculative features from an architecture does eliminate the speculative execution vulnerabilities, but it also degrades performance. Applying the known mitigations for the speculative execution vulnerabilities also significantly degrades performance. This performance trade-off is a blunt instrument, all code running on the system is affected.

We question the fundamental assumption in current research and production hardware—and in decades of hardware architecture design—that speculative features must be always off or always on, and so mitigations must also apply in a universal fashion. Early systems like the Intel i860 [67] combined speculative and non-speculative features, and despite decades of evolution in a different direction, the combination is still possible in modern hardware architectures. The advantage of such a hardware architecture would be the ability for systems software to choose between parts of a host or guest operating system that are so sensitive they must not be speculated, and other parts where performance is crucial but leaking information is harmless.

This chapter explores adding instructions to the RISC-V instruction set architecture (ISA), making it possible to selectively disable speculative execution within a single core. We refer to these systems as *demi-speculative* because they combine the features of both speculative and non-speculative microarchitectures.

5.1 Related work

In the late 1980s, the RISC-based Intel i860[1] included both speculative and non-speculative instructions, so the compiler had the power to choose whether to use a primitive form of speculation for branches and memory loads within a particular region of code [67]. The architecture was not commercially successful, so Intel abandoned it in the 1990s. However, in light of the speculative execution vulnerabilities discovered in recent years, it is fascinating to consider where the industry might have ended up today if mainstream superscalar processors had taken the path of the i860 rather than the x86. We can never know what might have been, but we can apply a similar approach to modern hardware architectures and evaluate the results.

[1] Also known as the 80860.

Some research into avoiding speculative execution vulnerabilities has taken the approach of implementing a traditional superscalar architecture with the standard RISC-V ISA and experimenting with minor variations in the microarchitecture. The Berkeley Out-of-Order Machine (BOOM) [22, 21] is a superscalar out-of-order RISC-V core, designed as a compatible substitute for the 5-stage in-order scalar pipelined RISC-V core Rocket [8], within the Chipyard [6] implementation framework for RISC-V custom SoCs. The third version of the BOOM core [131] offers more advanced speculation features, including an instruction fetch unit based on the TAGE branch predictor algorithm and a load-store unit that supports multiple loads per cycle. Gonzalez *et al.* [48] replicated the Spectre [66] bounds check bypass and branch target injection attacks on an extended RISC-V BOOM processor. The further addition of an L0 speculation buffer mitigated the attacks, by holding speculative load data in the buffer, to be flushed if the speculative load resolves as misspeculated, or written to the L1 cache when the speculative load commits.

Some researchers have also proposed minor variations on the RISC-V ISA. Yu *et al.* [129] prototyped a RISC-V extension on the BOOM core, to track confidentiality labels on data against security-guarantees of instructions, as a mitigation for microarchitecural side-channel attacks. Escouteloup *et al.* [35] proposed an extension to the RISC-V ISA (specifically the RV32I base ISA) introducing a concept of confidential registers and hardware security contexts to express security boundaries, as a mitigation for some side-channel attacks. Wistoff *et al.* [127] proposed adding a `fence` instruction to the RISC-V Ariane core, to flush microarchitectural state when switching between security contexts, based on earlier research [43, 51, 45, 44] into time protection mechanisms against side-channel attacks.

5.2 Feasibility considerations

As in Chapters 3 and 4, the ideas discussed in this chapter are applicable to other hardware architectures, such as x86 and ARM. But, the modular nature of the RISC-V ISA—consisting of a base set of integer instructions and a collection of standard and non-standard instruction extensions for more complex features—does make it a particularly good target for experimenting with combining speculative and non-speculative instructions in a single core.

5.2.1 RISC-V ISA extensions

Within the RISC-V 32-bit base integer instruction set (RV32I), only a relatively small number of instructions are relevant for speculative execution: load operations access memory, and conditional branch instructions create decision points in the control flow. An ISA extension to support both speculative and non-speculative features would add a small number of duplicate instructions, so each speculative instruction would have a non-speculative variant. Following the RISC-V naming convention specified for non-standard extensions, we call this hypothetical extension "Xdemispec". Table 5.1 lists the relevant RV32I base instructions and their variants in the extension.[2] Throughout this chapter, we will describe the base instructions as speculative and the Xdemispec extension as non-speculative, for

[2]There is no particular significance in the use of "X" as the first letter in the names of the variant instructions, it was chosen merely because the character is rarely used, relatively distinctive, and akin to the "X" prefix for non-standard extensions.

example the `beq` (branch if equal) instruction is speculative, while the `xbeq` instruction in the Xdemispec extension is non-speculative. The concept could be implemented equally well with non-speculative base instructions, but a speculative base means that unmodified compilers get the performance advantages of the speculative instructions, while modified compilers can access the security benefits of disabling speculation for limited sections of code.

Table 5.1: Demi-speculative extensions for RISC-V 32-bit base integer instruction set

Description	RV32I Base	RV32I_Xdemispec
Load Byte (8-bit)	`LB rd,rs,imm`	`XLB rd,rs,imm`
Load Halfword (16-bit)	`LH rd,rs,imm`	`XLH rd,rs,imm`
Load Word (32-bit)	`LW rd,rs,imm`	`XLW rd,rs,imm`
Load Byte Unsigned	`LBU rd,rs,imm`	`XLBU rd,rs,imm`
Load Halfword Unsigned	`LHU rd,rs,imm`	`XLHU rd,rs,imm`
Branch Equal	`BEQ rs1,rs2,imm`	`XBEQ rs1,rs2,imm`
Branch Not Equal	`BNE rs1,rs2,imm`	`XBNE rs1,rs2,imm`
Branch Less-Than	`BLT rs1,rs2,imm`	`XBLT rs1,rs2,imm`
Branch Greater-Than or Equal	`BGE rs1,rs2,imm`	`XBGE rs1,rs2,imm`
Branch Less-Than Unsigned	`BLTU rs1,rs2,imm`	`XBLTU rs1,rs2,imm`
Branch Greater-Than or Equal Unsigned	`BGEU rs1,rs2,imm`	`XBGEU rs1,rs2,imm`

Memory store operations (`sb`, `sh`, `sw`, and `sd`) participate in speculative execution, but only to the extent that they must wait until any speculative results they depend on have been finally committed. Since the Xdemispec extension makes it possible to mix speculative and non-speculative instructions, it would not be safe to provide the user with alternative store instructions that ignore speculation, since the user might incorrectly use the non-speculative instruction to store results that actually depend on some speculatively executed code. Instead, the microarchitecture implementation of the store instructions must be modified to recognize that the Xdemispec load and branch instructions do not participate in speculative execution, which effectively means they always commit immediately after the execution stage.

In RISC-V, computational instructions—such as math or logic operations—only operate on registers and immediates, so while they might benefit from a speculative memory fetch or might run as part of a speculative branch prediction, they are inherently neutral to speculation, and do not require variants in the Xdemispec extension.[3]

The RISC-V 64-bit base integer instruction set (RV64I) adds two instructions that are relevant to speculation. Table 5.2 shows variants for these instructions in the Xdemispec extension.

[3]It is worth noting that the x86 instruction set defines computational instructions that operate directly on memory locations, so a similar extension for x86 would require many more instruction variants than RISC-V.

Table 5.2: Demi-speculative extensions for RISC-V 64-bit base integer instruction set

Description	RV64I Base	RV64I_Xdemispec
Load Doubleword (64-bit)	`LD rd,rs,imm`	`XLD rd,rs,imm`
Load Word Unsigned	`LWU rd,rs,imm`	`XLWU rd,rs,imm`

Beyond the base integer instruction sets, RISC-V defines a combination of standard extensions for general-purpose computing, including multiplication and division instructions (extension M), atomic instructions (A), single-precision (F) and double-precision (D) floating point instructions, control and status register instructions (Zicsr), and the instruction-fetch fence instruction (Zifencei). This combination of integer base and standard extensions is abbreviated from "IMAFDZicsr_Zifencei" to simply "G", so the 32-bit and 64-bit general-purpose combined instruction sets can be clearly referred to as RV32G and RV64G.

RV64G is the typical target for RISC-V hardware capable of running a full Linux operating system, so it is reasonable for the Xdemispec extension to consider the full set of RV64G instructions. The multiplication and division extension (M) and floating point extensions (F and D) do not add any instructions relevant to speculation, and so do not require variants in the Xdemispec extension. The atomic extension (A) adds load and store operations for memory, however the instructions read, modify, and write memory within a single operation (for syncronization between multiple RISC-V hardware threads), and so will always be non-speculative. The control and status register extension (Zicsr) adds loads and stores of the Control/Status Register (CSR) set, however CSR instructions are also only executed non-speculatively. The instruction-fetch fence extension (Zifencei) may require changes at the microarchitecture level to handle the combination of speculative and non-speculative instructions, but does not require the addition of any variant instructions for the Xdemispec extension.

In summary, the Xdemispec extension adds a total of only 13 new instructions to the RV64G general-purpose instruction set. This increase in the footprint of the ISA is tolerably small, when weighed against the benefit of providing user-level control over speculative execution.

5.2.2 Microarchitecture

One crucial challenge for implementing demi-speculative features at the instruction level lies in the microarchitecture, specifically in implementing a pipeline capable of efficiently executing both speculative and non-speculative instructions. As in Chapter 4, consider a foundation of a superscalar microarchitecture that is roughly analogous to a modern x86 processor, which uses dynamic multiple issue and dynamic pipeline scheduling, with out-of-order execution. There are many possible ways to implement the microarchitecture of both speculative and non-speculative features, but some are more compatible than others. Specifically, combining a modern superscalar microarchitecture with an old scalar microarchitecture on a single core would effectively require including two completely different instruction pipelines each with their own microarchitectural state. On the other hand, the speculative and superpositional microarchitectures described in Chapter

4 are highly compatible—they use identical instruction pipelines and nearly identical microarchitectural state, with only minor differences in which instructions are fetched and when instructions are executed. The next two sections describe a combination of speculative and superpositional pipelining as one reasonable way to implement a microarchitecture supporting the Xdemispec extension.

5.2.2.1 Branch instructions

In the demi-speculative ISA, ordinary branch instructions are speculative, while the non-speculative branch instructions in the Xdemispec extension do not participate in branch prediction.

Fetch: Instruction fetching for the non-speculative branch instructions does not read from or update the branch target buffer, branch history buffer, branch history table, pattern history table, or return stack buffer, even though the microarchitecture has these features in the hardware and uses them for the speculative branch instructions. This means both that non-speculative branch instructions will never use mistrained predictions from other speculative branch instructions (so they might be used in regions of code that are critical to security), and also that non-speculative branch instructions cannot be used to mistrain the branch predictor (so they might be substituted for speculative branch instructions in regions of code that are untrusted).

Issue: As in Section 4.2.2.2, the pipeline for non-speculative branch instructions would fetch instruction entries for both branch paths and place them in the reorder buffer, tagging the entries with a control dependency on the result of the conditional branch instruction, and renaming registers in each branch path so the two branch code paths do not use the same physical registers. Because the reorder buffer entries for non-speculative branch instructions are tagged with a special control dependency, they do not interfere with ordinary speculative branch instructions, which proceed through the pipeline in the normal speculative way. For non-speculative branches, the pipeline sends instructions from both branch paths to the reservation stations, preserving the control dependency tag. For speculative branches, the pipeline only sends instructions from the predicted branch path to the reservation stations, but without the control dependency tag.

Execute: The reservation stations buffer all instructions issued from either speculative or non-speculative branch paths in the usual way, however they treat the control dependency tag on non-speculative branch instructions as similar to a data dependency of waiting for operands. Instructions from a speculated branch path are dispatched to a functional unit to be executed immediately. However, no instructions from either non-speculative branch path are dispatched to a functional unit until the result of the non-speculative conditional branch instruction is known, so there is no speculative execution of any instruction for the non-speculative branches.

Commit: The commit unit treats the instructions from the rejected non-speculative conditional branch path in the same way as incorrectly speculated instructions, it frees up the renamed registers and removes the instruction entries from the reorder buffer. As in Sections 4.2.1.4 and 4.2.2.4, the commit unit processes results from the reorder buffer

for all instructions—speculative or non-speculative—in the order the instructions were fetched.

As discussed in Section 4.2.2, the size of several fetch and issue components need to be increased to maintain the same instruction throughput despite discarding instructions on non-speculative branch paths.

5.2.2.2 Memory load instructions

Continuing to consider a superscalar microarchitecture that is roughly analogous to a modern x86 processor, the demi-speculative RISC-V microarchitecture may speculatively execute ordinary memory load instructions. Ordinary memory load instructions within a non-speculative branch path behave like any other instruction to the extent that they will be fetched and issued, but will not be speculatively executed as part of the branch. However ordinary memory loads could still be set up for speculative execution by the memory disambiguator. There is a viable use case for permitting the memory disambiguator to speculate memory loads within a non-speculative branch path,[4] so in this hypothetical demi-speculative microarchitecture only the non-speculative memory load instructions in the Xdemispec extension fully avoid speculative execution.

Fetch: Instruction fetching for speculative memory load instructions uses speculative predictions from the memory disambiguator, and participates in training the memory disambiguator for future speculative predictions. Non-speculative memory load instructions do not use the memory disambiguator for predictions, however, they do use the memory disambiguator for tracking when all prior stores to the same address have been completed—resolving all Store To Load (STL) dependencies for the memory load instruction—to determine when the memory load is ready to execute non-speculatively.

Issue: Both speculative and non-speculative memory load instructions are fetched, issued with an entry in the reorder buffer, and dispatched to the reservation stations, the same as in Sections 4.2.3.2 and 4.2.4.2. Speculative memory load instructions proceed through the reorder buffer to the reservation stations in the normal speculative way. Non-speculative memory load instructions that are fetched and issued as part of a non-speculative branch path are tagged with a control dependency on the result of the non-speculative conditional branch instruction. Non-speculative memory load instructions outside of a non-speculative branch path are tagged with a control dependency by the memory disambiguator, so they cannot execute before all prior stores to the same address have been completed.

Execute: The reservation stations preserve the control dependency tag on non-speculative memory load instructions, treating it similarly to a data dependency of waiting for operands. They will not dispatch a non-speculative memory load instruction to the load-store unit until all prior stores to the same address have been completed and/or the result of the non-speculative conditional branch is known. Since the result of the non-speculative conditional branch instruction is known before any instructions on the non-speculative branch path execute, any exceptions raised by the non-speculative memory load instruction are

[4] As usual, this is primarily a trade-off between security and performance.

raised immediately, so there is no period of transient execution to allow access to incorrect memory loads through covert side channels. Speculative memory load instructions proceed through the reservation stations, through the functional units, and on to the commit unit in the normal speculative way.

Commit: Since non-speculative memory load instructions are never speculatively executed, the commit unit always marks the memory load instruction entry in the reorder buffer as complete after it receives a result from the load-store unit. For speculative memory load instructions, the commit unit determines whether the speculated memory load instruction was speculated correctly, and if so, marks the instruction entry in the reorder buffer as complete, performs any pending register writes or memory stores, and removes the instruction entry from the reorder buffer. If the commit unit determines the speculation was incorrect, it will discard the result of the speculated memory load, and either remove the memory load instruction entry from the reorder buffer (if the instruction was on a misspeculated branch path), or else execute the memory load all over again together with any instructions that depended on its result (if the instruction was misspeculated by the memory disambiguator).

As discussed in Section 4.2.5, the performance of non-speculative memory loads can substantially benefit from hardware prefetching, but the feature does need to be designed carefully to avoid leaving microarchitectural traces. Speculative memory loads also benefit from hardware prefetching that avoids leaving microarchitectural traces, so choosing hardware prefetching techniques such as those in Section 4.2.5 are a better choice overall than current hardware prefetching approaches that naively update the L1-L3 cache with results while speculating memory loads.

5.2.3 High-level language modifications

If demi-speculative features were implemented at the instruction level, the challenge for high-level languages would be how to expose the concept of choosing between speculative and non-speculative instructions, in a way that is meaningful to programmers and relatively easy to use. An extended ISA would require modifying the compiler to output the added instructions. Such a change involves modifying the definition of the source high-level language, adding a way for programmers to indicate which sections of code should be non-speculative, modifying the parser to recognize the new syntax, modifying the semantic analysis phase to retain information about regions of code tagged as non-speculative, and modifying the generator to select different instructions within non-speculative regions of code. There are many different ways such changes could be implemented in a compiler, this section and Section 5.2.4 explore one possible example using Rust as the high-level language and LLVM as the compiler toolchain, to demonstrate the feasibility of the approach.

In Rust, the `unsafe` keyword instructs the Rust compiler to alter a few small but significant low-level behaviors related to memory safety. Without delving into the details of how `unsafe` functions in Rust, what is relevant here is that Rust programmers have become familiar with the concept of `unsafe` as a strategically placed keyword that alters the output of the compiler.[5] The `unsafe` keyword is allowed in three positions in the

[5]Other programming languages have keywords to alter compiler behavior, such as `volatile` in C,

syntax of Rust: as a block, on a function or method definition, and on a trait declaration or implementation. We posit a new keyword named `nospec`, allowed in all the same syntactic positions as the `unsafe` keyword.

When used as a block, the `nospec` keyword indicates that all code within the body of the block should be compiled using non-speculative instructions:

```
nospec {
    // non-speculative code here
}
```

Conditional branches within the `nospec` block will not run speculatively, which means no code that depends on the branch instruction will execute until the condition is evaluated, no trace will be left in the cache of falsely predicted results, and no trace of the branch will be left in the branch predictor, so the code cannot influence any future branch predictions. Memory accesses within the `nospec` block will not perform speculative memory fetching, which means they will leave no trace of misspeculated loads in the cache to be exposed by side-channel attacks.

When used on a function or method definition, the `nospec` keyword indicates that all code within the body of the function should be compiled using non-speculative instructions. A function defined as a `nospec` function can only be called from within a `nospec` block, as a way of requiring the programmer to explicitly take responsibility for the altered behavior. In the code example below, the `leave_no_trace` function is defined as non-speculative, and can only be called from within a `nospec` block:

```
nospec fn leave_no_trace() {}

nospec {
    leave_no_trace();
}
```

When used on a trait definition or implementation, the `nospec` keyword indicates that all code within the body of the trait should be compiled using non-speculative instructions. In the code example below, the `LeaveNoTrace` trait is defined as non-speculative, and later implemented for the `Password` type:

```
nospec trait LeaveNoTrace {
    // method signatures
}

nospec impl LeaveNoTrace for Password {
    // method implementations
}
```

C++, C#, and Java, but Rust's `unsafe` is a cleaner example of high-level language syntax for clearly delineated blocks of code.

5.2.4 Compiler toolchain modifications

Taking a step back from the high-level language to the compiler toolchain that supports it, the challenge at this layer is how to capture and preserve information about the `nospec` feature through all compilation phases, in order to finally output non-speculative instructions.

LLVM is a collection of libraries and tools, used for both static and dynamic compilation of programming languages. The compiler for the Rust programming language uses LLVM for code generation: one of the later stages of Rust compilation produces output in LLVM Intermediate Representation (IR)—a kind of heavily annotated assembly language—which LLVM takes as input to run optimization passes and generate machine code for the target architecture as the final output. LLVM is designed to be extensible and allow for custom behavior at every stage of the compilation process, including code generation for a wide variety of hardware architectures and even non-traditional code generation targets such as WebAssembly.

At the highest level of LLVM IR produced by the Rust compiler, LLVM has a built-in system for attaching metadata to annotate the IR instruction stream with additional information. The most common use of the metadata feature in LLVM is the `!dbg` identifier to capture source-level debug information in a standard form, and preserve that information through any optimization or tranformation passes, all the way through to the final generated machine code. It would be possible to add a `!nospec` metadata identifier to LLVM, and then modify the Rust compiler to annotate IR instructions, functions, and modules with the metadata identifier, corresponding to the high-level language code blocks, functions, and traits defined with the `nospec` keyword. The following code example in LLVM IR is a conditional branch, which branches to the destination label `true` when the condition evaluates as true, and to the label `false` when the condition evaluates as false. Normally, LLVM would compile this conditional branch as a speculative branch instruction in the standard ISA, but adding the metadata identifier `!nospec` to this conditional branch would tell LLVM to compile it as a non-speculative branch instruction in the extended ISA.

```
br i1 %cond, label %true, label %false, !nospec !0
```

At the lowest level of LLVM's code generation, the extended demi-speculative RISC-V ISA would need to be defined as a separate target, though it would inherit almost all features from the existing standard RISC-V target. Each instruction added in the demi-speculative RISC-V extension in Section 5.2.1 would require adding an entry in the `TargetInstrInfo` class for the target to describe the instruction.

Optimization or transformation passes in LLVM discard any metadata annotation they are unable to recognize, so each pass to be used in the compilation of the LLVM IR produced by the Rust compiler would need to be modified to preserve the custom `!nospec` metadata identifier. The instruction selection pass would also need to be modified, to recognize the metadata identifier, and use it to select the extended RISC-V demi-speculative instructions instead of the standard RISC-V instructions.

Overall, the modifications to LLVM required to support an extended RISC-V ISA are not trivial, but they do lie within the realm of custom compiler features that LLVM was designed to support.

5.3 Evaluation

While the ability to disable speculation for small regions of code is a security advantage over speculating all code, implementing demi-speculative features at the ISA level is not radically more secure than the heterogeneous muilticore alternative outlined in Chapter 3 and is less secure than entirely eliminating speculation with the standard ISA approach in Chapter 4. Taking full advantage of the extended RISC-V ISA requires a web of changes through multiple layers of systems software. Such changes are feasible, but also disruptive, in a way that may initially make it more difficult to validate the security of the overall system.

From a performance perspective, the demi-speculative ISA approach has an advantage over the heterogeneous multicore approach in Chapter 3, in that it avoids the overhead of inter-process communication between speculative and non-speculative regions of code. However, this advantage is balanced against the potential performance loss of a more complex microarchitecture pipeline combining speculative and non-speculative features.

From a portability perspective for multitenant infrastructures such as cloud and containters, the disadvantage of the demi-speculative ISA approach is that host and guest operating systems and workloads would be highly dependent on the low-level details of the extended ISA. A host operating system or guest image compiled for a demi-speculative RISC-V ISA on one server is not portable to a standard RISC-V ISA on another server, though the source code could be portable as long as the compiler ignores any demi-speculative annotations when compiling for targets that do not support them. On the other hand, the demi-speculative ISA approach also has an advantage in resource allocation over the heterogeneous multicore approach—because every core is identical and able to run in a non-speculative mode, the host is free to allocate workloads to any core. With the heterogeneous multicore approach in Chapter 3, the host must allocate speculative workloads to speculative cores, and non-speculative workloads to non-speculative cores. Since the hardware for a heterogeneous multicore is manufactured with a fixed number of speculative and non-speculative cores, there is no flexibility to change that allocation in response to demand. Across a data center with thousands or hundreds of thousands of servers, an imbalance in the utilization of the speculative and non-speculative cores could result in a substantial performance loss through unused resources and wasted capacity.

Ultimately, the heterogenous multicore approach is easier to deliver in the short-term future, but the demi-speculative ISA approach and the standard ISA approach with superpositional pipelines are worth further research in the longer-term.

5.3.1 Performance

For performance evaluation, we use a modified version of the BOOM core, called Dyon.[6] Similar to the way performance on the Gluon multicore depends heavily on whether a workload runs on a speculative core or a non-speculative core, performance on the Dyon core depends heavily on whether a workload has been compiled with speculative instructions or non-speculative instructions. In this limited prototype, a benchmark with all speculative branch instructions has the same performance by IPC on a Dyon core as on

[6] Like Gluon and Tachyon, there is no particular significance to "Dyon", we have only adopted names for clarity of exposition. In the domain of physics, a dyon is a hypothetical particle that has both electric and magnetic charges. The word also has the same Latin root as "dyad", meaning a thing composed of two distinct parts.

a BOOM core, and a benchmark with all non-speculative branch instructions has the same performance by IPC on a Dyon core as on a Tachyon core. A more interesting question is what happens when a workload mixes speculative and non-speculative instructions. To model the effects of mixed workloads without changing the compiler toolchain or recompiling benchmarks, we built a series of variations on the Dyon core, shown in Table 5.3, where instead of making the branch instructions in the standard ISA all speculative or non-speculative, we made some branch instructions in the standard ISA speculative and some non-speculative. The Dyon cores were built and executed within the Chipyard framework using the Verilator RTL simulator for a cycle-accurate behavioral model, and using the FireSim [61] simulation platform for cycle-exact microarchitectural simulation deployed on AWS F1 FPGAs.

Table 5.3: Characteristics of Dyon variations.

Variant	Speculative	Non-Speculative
Dyon.A	BGE	all other branches
Dyon.B	BEQ	all other branches
Dyon.C	all other branches	BEQ
Dyon.D	all other branches	BGE

5.3.1.1 RTL simulation

Evaluating the Dyon core variations in Table 5.3 on a baseline set of benchmarks from the RISC-V project [100], the results in Figure 5.1 show that performance of the benchmarks by IPC varies depending on the relative proportion of speculative to non-speculative branch instructions. In this particular set of benchmarks, `bge` branch instructions (branch if greater or equal) are relatively rare, so for the Dyon.A variant—which only speculates `bge` branch instructions and runs all other branch instructions as non-speculative—the performance by IPC is only slightly better than the Tachyon core. But, for the Dyon.D variant—which speculates all branch instructions except `bge`—the performance by IPC for most of the benchmarks is only slightly worse than the BOOM core. In three of the benchmarks (`multiply`, `spmv`, and `vvadd`) the performance on the Dyon.D variant is slightly better than the BOOM core, which could be explained if some `bge` branches were misspeculated on BOOM, so that the non-speculative `bge` ends up performing better by avoiding the hit of misspeculation.

In contrast, the `beq` branch instructions (branch if equal) are far more common in the `dhrystone` and `towers` benchmarks in Figure 5.1, so the performance of these benchmarks on the Dyon.B variant—which only speculates `beq` branch instructions—is substantially better than the Tachyon core in the `dhrystone` benchmark, and nearly as good as the BOOM core in the `towers` benchmark. And, for the Dyon.C variant—which speculates all branch instructions except `beq`—the performance by IPC is substantially worse than the BOOM core on the `towers` benchmark and almost as bad as the Tachyon core in the `dhrystone` benchmark. In the other benchmarks, the `beq` branch instruction is rare, so the performance on the Dyon.B variant is about the same as the Dyon.A variant, and the performance on the Dyon.C variant is about the same as the Dyon.D variant.

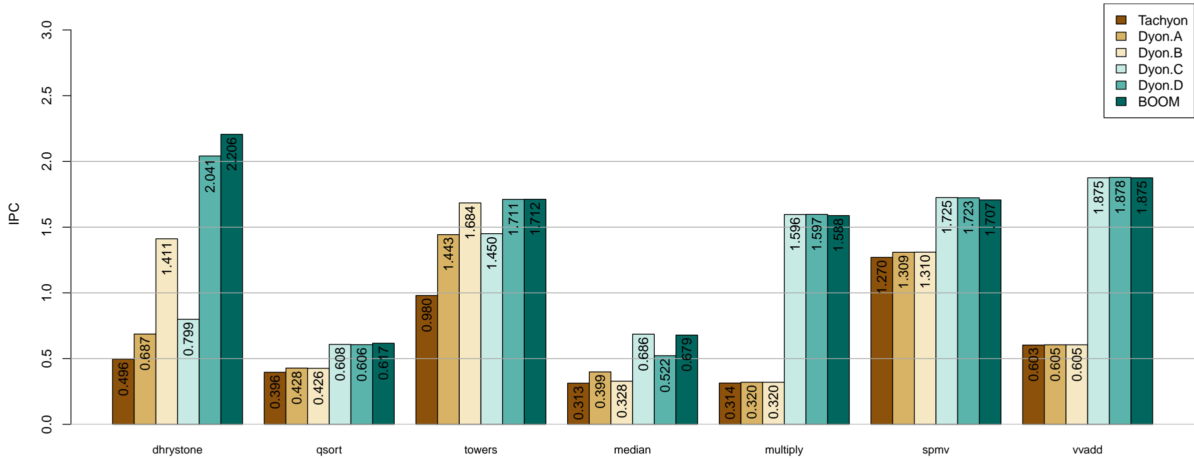

Figure 5.1: Comparing a series of variations on the Dyon core.

These results indicate that while workloads with mixed speculative and non-speculative branch instructions do pay a performance penalty, they do so in proportion to their use non-speculative instructions. Workloads that make only light use of non-speculative instructions, suffer only minor loss of performance. This effect of gradually degraded performance sets the demi-speculative ISA approach apart from most current approaches to mitigating the speculative execution vulnerabilities, because it puts the systems software developer in control of when to choose speculation for performance, and when to trade some performance for the security of eliminating speculation.

5.3.1.2 FPGA simulation

As in Section 4.3.1.2, we evaluated the Rocket, BOOM, and Dyon cores simulated with FireSim on AWS F1 FPGAs, using the SPEC CPU2017 intspeed benchmarks. The FireSim simulations ran at a host frequency of 65MHz on the FPGAs, and modeled the system running at 1GHz, configured as single-core systems with 512KB L2, 4MB simulated L3, and 16GB DRAM, and with 32KB L1I, 32KB L1D on the BOOM and Dyon cores, but 16KB L1I, 16KB L1D on the Rocket core. The SPEC benchmarks were compiled with `gcc`, with `-O3` optimizations.

Evaluating the Rocket, BOOM, and Dyon cores in FPGA simulation on the SPEC CPU2017 intspeed benchmarks, the results in Figure 5.2 show a pattern of relative performance between the cores similar to the RISC-V benchmarks in Figure 5.1—performance of the benchmarks by IPC varies depending on the relative proportion of speculative to non-speculative branch instructions. The performance of all benchmarks on the Dyon.B variant—which only speculates `beq` branch instructions—is better than the Rocket core and at least slightly better than the Tachyon core. The performance of most benchmarks on the Dyon.C variant—which speculates all branch instructions except `beq`—is much closer to the performance of the BOOM core. Only one benchmark (`gcc`) performs better on the Dyon.B variant than the Dyon.C variant, because the `beq` branch instruction occurs so frequently in the compiled benchmark that speculating that one type of branch instruction has a greater performance impact than speculating all other branch instructions.

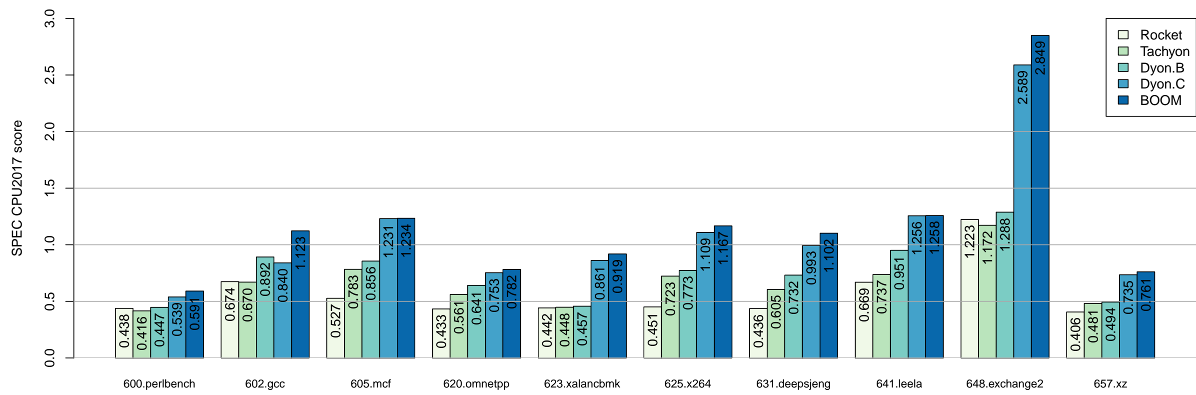

Figure 5.2: Comparing a series of variations on the Dyon core.

Chapter 6

Conclusions

The discovery of the speculative execution vulnerabilities in 2018 created a shock wave in hardware and software security research that continues to unfold. New variants on the vulnerabilities are published regularly, usually with a collection of mitigations for the symptoms of each variant. Some mitigations are adopted by major hardware vendors and deployed by public providers of multitenant infrastructures. Many proposed mitigations never see significant adoption because the performance penalty of the specific mitigation is too high, or because the performance penalty of deploying all the relevant mitigations is so high that hardware vendors and infrastructure providers choose to adopt only a subset of mitigations with the greatest impact for their particular target use case. The current mitigations work around the symptoms of specific variants, but they do not address the root cause of the speculative execution vulnerabilities, which is a fundamental design flaw in the speculation features of modern superscalar hardware architectures. If we continue this way, we can look forward to many generations of hardware debilitated by performance penalties from increasing layers of mitigations as new variants are discovered, and yet still vulnerable to variants that have yet to be discovered, or variants that have been discovered by malicious parties and have yet to be reported or mitigated. There is no real resolution to the speculative execution vulnerabilities on the horizon.

This dissertation has sought to demonstrate the potential of research avenues that fundamentally rethink the role of speculation in modern hardware architectures. We have explored whether speculation could be partially or completely eliminated, and the security and performance implications of doing so.

Chapter 2 established the background for this dissertation. We described the unique requirements that multitenant infrastructure environments have for hardware and software security, and how key concepts of multitenant infrastructures developed over time. We then outlined how the speculative execution vulnerabilities undermine crucial assumptions about hardware security that researchers and the industry have been making for decades.

Chapter 3 explored the potential for heterogeneous multicore architectures that combine speculative and non-speculative cores. Gluon-type multicores make it possible run a process or thread as non-speculative, but cannot provide any tighter level of control over speculation. Performance is determined by which core runs the workload—big speculative cores in a Gluon multicore perform as well as a single speculative core and small non-speculative cores in a Gluon multicore perform as poorly as a single non-speculative core. Because heterogenous multicore systems are already shipping in (mobile and laptop) production hardware, Gluon-type systems are more amenable to production deployments in the short-term future. However, these systems are inflexible for resource allocation—the decision

of how much capacity to allocate to speculative and non-speculative execution must be made at the time of hardware manufacture—which makes them unsuitable for large-scale multitenant infrastructures.

Chapter 4 explored the logical limits of performance for a modern superscalar architecture without speculation. Tachyon-type cores entirely eliminate speculative execution, so they are more secure than either Gluon-type multicores or Dyon-type cores, but they also cannot use speculation to improve performance even when it would be safe to do so. Performance of a Tachyon-type core is never as good as an equivalently sized speculative core, but it can be improved by increasing the size of fetch and issue stage components of the pipeline. Initial results suggest that a full implementation of superpositional pipelining may further improve the performance of Tachyon-type cores. It is unlikely that non-speculative superscalar cores will ever match the performance of unmitigated speculative cores, but it is feasible that they may reach the point of consistently performing as well as or better than speculative cores with all relevant speculative execution mitigations applied. The performance potential of Tachyon-type cores, combined with the ability to completely eliminate the speculative execution vulnerabilities, places them in the running as a viable solution. Tachyon-type cores may be suitable for special-purpose multitenant infrastructure deployments that exclusively serve privacy-centric workloads or are hosted in legal jurisdictions with strong data privacy laws. However, keeping speculative execution features as an option for peak performance will be desirable for most general-purpose multitenant infrastructure deployments.

Chapter 5 explored the potential for hardware architectures that include both speculative and non-speculative instructions on a single core. Dyon-type cores give the systems software developer precise control over when to use speculation features, at the level of a single instruction. Performance degrades gradually in proportion to the use of non-speculative instructions—code compiled as entirely speculative pays no performance penalty, and code compiled as lightly non-speculative only pays a minor performance penalty—so Dyon-type cores have peak performance equivalent to speculative cores, while also allowing for fine-grained control over speculative execution. Dyon-type cores are more flexible than Gluon-type multicores for resource allocation in large-scale multitenant infrastructures, because all cores in the system are identical and equally able to combine speculative and non-speculative execution. Dyon-type cores are most suitable for general-purpose multitenant infrastructure deployments, because they put the choice of when to trade performance for security into the hands of the customers, and maintain flexibility in resource allocation, without sacrificing performance in the common case.

6.1 Future work

The scope of this dissertation has been limited to what one researcher can accomplish in a short period of time. While the work is enough to reveal promising potential, it is tantamount to the opening bars of a symphony. A complete hardware design and implementation based on the concepts discussed would be an extensive and multi-year research agenda for a group of researchers. This section briefly discusses some future research directions suggested by this work.

6.1.1 Heterogeneous multicores

While Gluon-type multicores are not a good choice for large-scale multitenant infrastructures, they may be a reasonable approach to consider for mobile and laptop devices, which have less extreme requirements for flexible resource allocation. The disruption of shifting to Gluon-type multicores would be particularly minimal in products where the hardware is already running a combination of big and little cores for performance and energy consumption considerations, as in some modern smartphones and Apple's M1 laptops.

It would be interesting to investigate the performance impact of splitting up workloads to keep security-critical sections of code on non-speculative cores. We anticipate that the added overhead of inter-process communication between speculative and non-speculative sections of code running on different cores would have a performance penalty, but measuring how much of a penalty would require a set of custom benchmark workloads to explore the relative performance of fully speculative, fully non-speculative, and partially speculative workloads on Gluon-type multicores.

It would also be worth exploring whether the performance gap between the speculative and non-speculative cores of a Gluon-type multicore could be improved by using Tachyon-type cores instead of in-order scalar cores for the non-speculative half. Such a system would be less useful for reducing energy consumption in mobile and laptop devices, but could be more useful for improving security on desktop devices without sacrificing performance.

6.1.2 Non-speculative cores

The limited prototype of Tachyon-type cores in Chapter 4 only disables branch prediction on a superscalar core. It does not fully implement superpositional branch pipelining, and still uses speculative memory loads. The most promising future research directions for Tachyon-type cores are to fully implement superpositional branch pipelining—freeing up the fetch stage bottleneck for branches by fetching and issuing instructions from both branch paths in parallel, as in Section 4.2.2—and to implement speculation buffers—keeping partially speculative memory loads but isolating them from the cache hierarchy, as in Section 4.2.5.

The Chipyard framework and FireSim platform used in this dissertation have somewhat rigid definitions of the cache and memory hierarchy, so something like the gem5 simulator is a better choice for experimenting with speculation buffers for memory loads. We found gem5's RISC-V implementation to be highly unstable, even with default core and cache implementations and plain vanilla SPEC CPU 2006/2017 benchmark workloads. Any researcher continuing this work should be prepared to either invest a substantial amount of time in improving gem5's RISC-V implementation, or else migrate the experiment to the ARM or x86 architectures, which have mature and stable implementations in the gem5 simulator.

Several further research avenues are worth considering for improving the performance of Tachyon-type cores. Increasing the size of fetch and issue stage components in the pipeline showed potential in the prototype, and could be explored further in combination with superpositional branches and speculation buffers. The largest Tachyon prototypes were too large to fit on an AWS F1 FPGA, but FireSim's Golden Gate compiler does have some limited ability to split RTL simulations across multiple FPGAs, while still producing bit-identical, cycle-accurate results—which may make it possible to test larger cores on

FireSim, instead of only on Verilator. Multi-threading is another avenue worth exploring for Tachyon-type cores, for efficiency through parallelism rather than speculation. Running multiple threads on each core means the pipeline is less likely to stall, because even if it is held up on one workload waiting for a non-speculative branch condition to evaluate or a memory load to complete, it can still keep instructions for other workloads flowing through the execution stage. The addition of dedicated functional units for evaluating branch conditions, separate from the general ALUs, might be worth exploring as a way to improve throughput by ensuring that branch conditions are always evaluated as soon as possible, and not held up by other arithmetic or logic instructions in the pipeline.

6.1.3 Demi-speculative cores

The limited prototype of Dyon-type cores in Chapter 5 only disables branch prediction for the non-speculative instructions in the ISA extension. It does not fully implement superpositional branch pipelining, and still uses speculative memory loads. All the research avenues suggested above for Tachyon-type cores would also benefit Dyon-type cores.

One research avenue that might be worth pursuing for demi-speculative cores—instead of adding non-speculative instructions as an ISA extension—would be to split the RISC-V or ARM user mode into a more privileged speculative user mode (using speculative instructions) and a less privileged non-speculative user mode (using alternative non-speculative instructions). Selecting speculation with RISC-V or ARM modes would not give systems software developers as fine-grained control over speculative execution as an ISA extension can, but it would avoid the complexity of modifying the compiler toolchain to select non-speculative instructions, and would mean that any software compiled for the standard ISA could be run in either the speculative or non-speculative mode. On x86 architectures, a similar effect could be achieved using a model-specific register (MSR) to select non-speculative instructions within (for example) a virtualization context, however x86 MSRs are far less flexible than RISC-V or ARM modes.

Bibliography

[1] W. B. Ackerman and W. W. Plummer, “An Implementation of a Multiprocessing Computer System,” in *Proceedings of the First ACM Symposium on Operating System Principles*, New York, NY, USA: ACM, 1967, pp. 5.1–5.10.

[2] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy, “A Virtual Machine System for the 360/40,” IBM Cambridge Scientific Center, Cambridge, MA, USA, Tech. Rep. 36.010, May 1966.

[3] S. Ainsworth and T. M. Jones, “MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 132–144.

[4] S. Ainsworth and T. M. Jones, “The Guardian Council: Parallel Programmable Hardware Security,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 1277–1293.

[5] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, “Architecture of the IBM System/360,” *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87–101, Apr. 1964.

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” *IEEE Micro*, 2020.

[7] A. Aminot, Y. Lhuillier, A. Chateigner, and H.-P. Charles, “On the advantage of time-varying diversity of workload on functionally asymmetric multi-core,” in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, New York, NY, USA: Association for Computing Machinery, Jan. 2014, pp. 11–13.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The Rocket Chip Generator,” EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2016-17, Apr. 2016, p. 9.

[9] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini, “OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores,” in *Proceedings of the 3rd Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, 2019, p. 6.

[10] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff, “BYOC: A ”Bring Your Own Core” Framework for Heterogeneous-ISA Research,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 699–714.

[11] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, “OpenPiton: An Open Source Manycore Research Framework,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 217–232, Mar. 2016.

[12] G. Banga, P. Druschel, and J. C. Mogul, “Resource Containers: A New Facility for Resource Management in Server Systems,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA: USENIX Association, 1999, pp. 45–58.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, 2003, pp. 164–177.

[14] V. Berstis, “Security and Protection of Data in the IBM System/38,” in *Proceedings of the 7th Annual Symposium on Computer Architecture*, New York, NY, USA: ACM, 1980, pp. 245–252.

[15] T. M. Birhanu, Z. Li, H. Sekiya, N. Komuro, and Y.-J. Choi, *Efficient Thread Mapping for Heterogeneous Multicore IoT Systems*, Research Article, Feb. 2017. [Online]. Available: `https://www.hindawi.com/journals/misy/2017/3021565/`.

[16] A. Bradbury, G. Ferris, and R. Mullins, “Tagged memory and minion cores in the lowRISC SoC,” University of Cambridge, Computer Laboratory, Technical Report lowRISC-MEMO 2014-001, Dec. 2014.

[17] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith, “VM-based Security Overkill: A Lament for Applied Systems Security Research,” in *Proceedings of the 2010 New Security Paradigms Workshop*, New York, NY, USA: ACM, 2010, pp. 51–60.

[18] J. P. Buzen and U. O. Gagliardi, “The Evolution of Virtual Machine Architecture,” in *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, New York, NY, USA: ACM, 1973, pp. 291–299.

[19] C. Canella, J. Van 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,” *arXiv:1811.05441 [cs]*, May 2019.

[20] L. Catuogno and C. Galdi, “On the Evaluation of Security Properties of Containerized Systems,” in *2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS)*, Dec. 2016, pp. 69–76.

[21] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, “BOOM v2: An open-source out-of-order RISC-V core,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep. 2017.

[22] C. Celio, D. A. Patterson, and K. Asanović, “The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun. 2015.

[23] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution,” in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, Jun. 2019, pp. 142–157.

[24] J. Claassen, R. Koning, and P. Grosso, “Linux containers networking: Performance and scalability of kernel modules,” in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2016, pp. 713–717.

[25] E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi, “Multiprogramming STRETCH: Feasibility Considerations,” *Communications of the ACM*, vol. 2, no. 11, pp. 13–17, Nov. 1959.

[26] *Control Program-67 Cambridge Monitor System*. Hawthorne, New York: IBM Corporation, Oct. 1971.

[27] F. J. Corbató, J. H. Saltzer, and C. T. Clingen, “Multics: The First Seven Years,” in *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, New York, NY, USA: ACM, 1972, pp. 571–583.

[28] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, “An Experimental Time-sharing System,” in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, New York, NY, USA: ACM, 1962, pp. 335–344.

[29] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, Sep. 1981.

[30] J. B. Dennis and E. C. Van Horn, “Programming Semantics for Multiprogrammed Computations,” *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[31] L. I. Dickman, “Small Virtual Machines: A Survey,” in *Proceedings of the Workshop on Virtual Computer Systems*, New York, NY, USA: ACM, 1973, pp. 191–202.

[32] X. Dong, Z. Shen, J. Criswell, A. Cox, and S. Dwarkadas, “Spectres, Virtual Ghosts, and Hardware Support,” in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA: ACM, 2018, 5:1–5:9.

[33] S. W. Dunwell, “Design Objectives for the IBM Stretch Computer,” in *Papers and Discussions Presented at the December 10-12, 1956, Eastern Joint Computer Conference: New Developments in Computers*, New York, NY, USA: ACM, 1957, pp. 20–22.

[34] J. P. Eckert, “UNIVAC-Larc, the Next Step in Computer Design,” in *Papers and Discussions Presented at the December 10-12, 1956, Eastern Joint Computer Conference: New Developments in Computers*, New York, NY, USA: ACM, 1957, pp. 16–20.

[35] M. Escouteloup, J. Fournier, J.-L. Lanet, and R. Lashermes, “Recommendations for a radically secure ISA,” in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain: ACM, May 2020, p. 7.

[36] D. Evtyushkin, R. Riley, N. C. a. E. 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*, New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 693–707.

[37] R. S. Fabry, "A user's view of capabilities," University of Chicago, ICR Quarterly Report 15, Nov. 1967.

[38] R. S. Fabry, "Preliminary description of a supervisor for a machine oriented around capabilities," University of Chicago, ICR Quarterly Report 18, Aug. 1968.

[39] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[40] M. J. Flynn and A. Podvin, "Shared Resource Multiprocessing," *Computer*, vol. 5, no. 2, pp. 20–28, Feb. 1972.

[41] J. M. Frankovich and H. P. Peterson, "A Functional Description of the Lincoln TX-2 Computer," in *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, New York, NY, USA: ACM, 1957, pp. 146–155.

[42] S. W. Galley, "PDP-10 virtual machines," ACM, Mar. 1973, pp. 30–34.

[43] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time Protection: The Missing OS Abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–17.

[44] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[45] Q. Ge, Y. Yarom, and G. Heiser, "No Security Without Time Protection: We Need a New Hardware-Software Contract," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–9.

[46] D. Genkin and Y. Yarom, "Whack-a-Meltdown: Microarchitectural Security Games [Systems Attacks and Defenses]," *IEEE Security Privacy*, vol. 19, no. 1, pp. 95–98, Jan. 2021.

[47] R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[48] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović, "Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture," in *Proceedings of Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, Phoenix, AZ, USA: ACM, Jun. 2019, p. 7.

[49] P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson, "A Performance Evaluation of the Intel iAPX 432," *SIGARCH Comput. Archit. News*, vol. 10, no. 4, pp. 17–26, Jun. 1982.

[50] Z. He, G. Hu, and R. Lee, "New Models for Understanding and Reasoning about Speculative Execution Attacks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 40–53.

[51] G. Heiser, "For Safety's Sake: We Need a New Hardware-Software Contract!" *IEEE Design Test*, vol. 35, no. 2, pp. 27–30, Apr. 2018.

[52] J. Horn, *Reading privileged memory with a side-channel*, Jan. 2018. [Online]. Available: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

[53] J. Horn, *Speculative execution, variant 4: Speculative store bypass*, Feb. 2018. [Online]. Available: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[54] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM System/38 Support for Capability-based Addressing," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 341–348.

[55] *iAPX 432 General Data Processor Architecture Reference Manual*. Aloha, Oregon: Intel Corporation, 1981.

[56] Intel, *Deep Dive: Intel Analysis of L1 Terminal Fault*, Aug. 2018.

[57] Intel, *More Information on Transient Execution Findings*, 2018.

[58] Intel, *Q2 2018 Speculative Execution Side Channel Update, revision 1.4*, May 2021. [Online]. Available: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html`.

[59] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," *arXiv:1903.00446 [cs]*, Jun. 2019.

[60] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, Mar. 2015, pp. 342–346.

[61] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA: IEEE, Jun. 2018, pp. 29–42.

[62] Kernel Developers, *Capacity Aware Scheduling*, 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/scheduler/sched-capacity.html`.

[63] Kernel Developers, *Energy Aware Scheduling*, Nov. 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html`.

[64] Kernel Developers, *Energy Model of devices*, 2020. [Online]. Available: `https://www.kernel.org/doc/html/latest/power/energy-model.html`.

[65] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757 [cs]*, Jul. 2018.

[66] 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," *arXiv:1801.01203 [cs]*, Jan. 2018.

[67] L. Kohn and N. Margulis, "Introducing the Intel i860 64-bit microprocessor," *IEEE Micro*, vol. 9, no. 4, pp. 15–30, Aug. 1989.

[68] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," 2018.

[69] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, USA: IEEE Computer Society, Dec. 2003, p. 81.

[70] A.-T. Le, B.-A. Dao, K. Suzaki, and C.-K. Pham, "Experiment on Replication of Side Channel Attack via Cache of RISC-V Berkeley Out-of-Order Machine (BOOM) Implemented on FPGA," in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Virtual, May 2020, p. 4.

[71] Lee and Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.

[72] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Digital Press, 1984.

[73] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan. 2010, pp. 1–12.

[74] S. B. Lipner, W. A. Wulf, R. R. Schell, G. J. Popek, P. G. Neumann, C. Weissman, and T. A. Linden, "Security Kernels," in *Proceedings of the AFIPS National Computer Conference*, New York, NY, USA: ACM, 1974, pp. 973–980.

[75] 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," 2018, pp. 973–990.

[76] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv:1801.01207 [cs]*, Jan. 2018.

[77] R. Lottiaux and C. Morin, "Containers: A sound basis for a true single system image," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001, pp. 66–73.

[78] G. Maisuradze and C. Rossow, "Ret2spec: Speculative Execution Using Return Stack Buffers," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2109–2122, Oct. 2018.

[79] S. Mazor, "Intel's 8086," *IEEE Annals of the History of Computing*, vol. 32, no. 1, pp. 75–79, Jan. 2010.

[80] S. McFarling and J. Hennesey, "Reducing the cost of branches," in *Proceedings of the 13th annual international symposium on Computer architecture*, Washington, DC, USA: IEEE Computer Society Press, May 1986, pp. 396–403.

[81] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv:1902.05178 [cs]*, Feb. 2019.

[82] M. K. McKusick, M. J. Karels, K. Sklower, K. Fall, M. Teitelbaum, and K. Bostic, "Current Research by The Computer Systems Research Group of Berkeley," in *Proceedings of the European UNIX Users Group*, Brussels, Belgium, Apr. 1989.

[83] M. K. McKusick, "Twenty Years of Berkeley Unix - From AT&T-Owned to Freely Redistributable," in *Open Sources: Voices from the Open Source Revolution*, O'Reilly Media, Inc., Jan. 1999.

[84] A. Miller and L. Chen, "An Exercise in Secure High Performance Virtual Containers," Las Vegas, NV, USA, Jul. 2012, p. 5.

[85] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading Kernel Writes From User Space," *arXiv:1905.12701 [cs]*, May 2019.

[86] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee, "Towards an efficient single system image cluster operating system," in *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.*, Oct. 2002, pp. 370–377.

[87] S. Nanba, N. Ohno, H. Kubo, H. Morisue, T. Ohshima, and H. Yamagishi, "VM/4: ACOS-4 Virtual Machine Architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 171–178.

[88] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its protection system," in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, Nov. 1977, pp. 1–10.

[89] R. A. Nelson, "Mapping Devices and the M44 Data Processing System," IBM Thomas J. Watson Research Center, Yorktown Heights, NY, Research Report RC-1303, 1964, p. 44.

[90] P. G. Neumann and R. J. Feiertag, "PSOS revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference*, Dec. 2003, pp. 208–216.

[91] R. M. Norton, "Hardware support for compartmentalisation," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-887, May 2016, p. 86.

[92] D. O'Keeffe, D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch, *Lsds/spectre-attack-sgx*, Jan. 2018. [Online]. Available: `https://github.com/lsds/spectre-attack-sgx`.

[93] A. Opler and N. Baird, "Multiprogramming: The Programmer's View," in *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*, New York, NY, USA: ACM, 1959, pp. 1–4.

[94] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*. Elsevier Science, May 2017. [Online]. Available: `https://books.google.com/books?id=H7wxDQAAQBAJ`.

[95] C. Percival, "Cache Missing for Fun and Profit," in *Proceedings of BSDCan 2005*, Ottowa, Canada, 2005, p. 13.

[96] G. Popek and C. Kline, "A verifiable protection system," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 294–304, Jun. 1975.

[97] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," in *Proceedings of the 18th USENIX Conference on System Administration*, Berkeley, CA, USA: USENIX Association, 2004, pp. 241–254.

[98] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks across Cores Are Real," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 338–353.

[99] A. Randal, "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers," *ACM Computing Surveys*, vol. 53, no. 1, 5:1–5:31, Feb. 2020.

[100] RISC-V International, *RISC-V Benchmarks*, 2021. [Online]. Available: `https://github.com/riscv-software-src/riscv-tests/tree/master/benchmarks`.

[101] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1405–1411, Dec. 1972.

[102] D. Ritchie, "The Evolution of the Unix Time-Sharing System," in *Proceedings of a Symposium on Language Design and Programming Methodology*, vol. 79, London, UK, UK: Springer-Verlag, 1980, pp. 25–36.

[103] N. Rochester, "The Computer and Its Peripheral Equipment," in *Papers and Discussions Presented at the the November 7-9, 1955, Eastern Joint AIEE-IRE Computer Conference: Computers in Business and Industrial Systems*, New York, NY, USA: ACM, 1955, pp. 64–69.

[104] 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*, New York, NY, USA: ACM, 2019, pp. 723–735.

[105] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 88–105.

[106] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," *arXiv:2006.13353 [cs]*, Jun. 2020.

[107] D. Schor, *ARM Neoverse N1 Microarchitecture*, 2019. [Online]. Available: `https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1`.

[108] D. Schor, *Intel Sunny Cove Microarchitecture*, 2020. [Online]. Available: `https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove`.

[109] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 753–768.

[110] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," *arXiv:1807.10535 [cs]*, Jul. 2018.

[111] A. Seznec, "A new case for the TAGE branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA: Association for Computing Machinery, Dec. 2011, pp. 117–127.

[112] A. J. Smith, "Directions for memory hierarchies and their components: Research and development," in *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, Nov. 1978, pp. 704–709.

[113] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[114] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA: ACM, 2007, pp. 275–287.

[115] F. G. Soltis, *Fortress Rochester: The Inside Story of the IBM ISeries*. System iNetwork, 2001.

[116] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv:1806.07480 [cs]*, Jun. 2018.

[117] S. Swanson, L. K. McDowell, M. M. Swift, S. J. Eggers, and H. M. Levy, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 314–340, Aug. 2003.

[118] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, vol. C-19, no. 10, pp. 889–895, Oct. 1970.

[119] J. Van 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)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 991–1008.

[120] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 54–72.

[121] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[122] O. Weisse, J. V. Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," Tech. Rep., Aug. 2018, p. 7.

[123] M. V Wilkes, *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. Amsterdam, The Netherlands: North-Holland Publishing Co., 1979.

[124] M. V. Wilkes and D. W. Willis, “A magnetic-tape auxiliary storage system for the EDSAC,” *Proceedings of the IEE - Part B: Radio and Electronic Engineering*, vol. 103, no. 2, pp. 337–345, Apr. 1956.

[125] M. V. Wilkes, *Time-Sharing Computer Systems*, 2nd ed., 5. MacDonald & Co., 1968.

[126] D. Williams, R. Koller, M. Lucina, and N. Prakash, “Unikernels As Processes,” in *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2018, pp. 199–211.

[127] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, “Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core,” in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain, May 2020, p. 7.

[128] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 428–441.

[129] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, “Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing,” in *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2019. [Online]. Available: `https://www.ndss-symposium.org/ndss-paper/data-oblivious-isa-extensions-for-side-channel-resistant-and-high-performance-computing/`.

[130] Y. Zhai, L. Yin, J. Chase, T. Ristenpart, and M. Swift, “CQSTR: Securing Cross-Tenant Applications with Cloud Containers,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2016, pp. 223–236.

[131] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,” in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, Valencia, Spain, May 2020.