Mitigating Microarchitectural Vulnerabilities to Improve Cloud Security and Reliability

by

Kevin Robert Loughlin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Professor Baris Kasikci, Chair
Professor Todd Austin
Professor Onur Mutlu
Professor Moinuddin Qureshi
Professor Zhengya Zhang

Kevin Robert Loughlin
kevlough@umich.edu
ORCID iD: 0000-0003-4647-3201

To my parents, Kimm and Pat, and all that they have done for me.

ACKNOWLEDGMENTS

I'm very happy with my decision to have pursued a PhD, most notably because of the huge number of people that have made the journey so rewarding.

I'd first like to thank my family, whose nurture, care, and encouragement provide me with all the love and opportunities that I could ever ask for.

I'd like to thank the University of Michigan Computer Science and Engineering (CSE) department, the Semiconductor Research Corporation Applications Driving Architectures (SRC ADA) Center, the National Science Foundation Graduate Research Fellowship Program (NSF GRFP), and the Google PhD Fellowship program, each of which have funded my research at different points during my PhD.

I'd next like to thank James Mickens, who introduced me to research in undergrad and whose mentorship and support inspired and enabled me to pursue a PhD. In addition to being an intelligent and caring mentor, James has a special gift to make me laugh and smile no matter how I am feeling.

Upon my entrance into graduate school, despite me not officially being in his lab, Tom Wenisch and his students welcomed me to all lab activities with open arms. I am thankful for the advice and support that all of these individuals gave to "the kid" (yours truly).

Following what was perhaps the most difficult time of my PhD, when I felt burned out and out of ideas, I am grateful that Stefan Saroiu brought me on as a summer research intern at Microsoft. He opened my eyes to a lifetime's worth of research problems and ideas, and I continue to benefit from and enjoy his valuable insights, perspective, and advice to this day.

I share this same gratitude for Alec Wolman, whose mentorship has also been a vital part of my success from my very first day interning at Microsoft. I feel very fortunate to have returned to Microsoft for a second summer to intern under Alec's mentorship, and I greatly appreciate our continued collaborations following these internships.

I would also like to thank Shobha Vasudevan and Joe Wenjie Jiang, my co-hosts for my summer research internship at Google. Their patient teaching and excitement for inter-field collaboration vastly increased my knowledge of machine learning, experience which I find more and more valuable each day.

I thank Jon McCune, who has served as my Google PhD Fellowship mentor during my PhD. Jon has, and continues to, provide me with excellent career advice and perspective.

I next thank my dissertation committee members (Todd Austin, Onur Mutlu, Moin Qureshi, Zhengya Zhang, and my advisor, Baris Kasikci) for serving on my committee. Their input and insights have improved the quality of my research, especially by expanding the ways in which I identify, think about, and solve research problems.

I thank the members of Efeslab, with whom I have proudly shared a lab space for the past five years. My work has greatly improved as a result of many wonderful conversations and times that we've had together, both in-person and remote.

I would also like to thank all of my wonderful co-authors and collaborators during my time in graduate school. I have learned so much from working with such talented and intelligent individuals, and I could not have completed my degree without them.

I am immensely grateful to my friends for all the times that we've shared and for supporting me in both the good times and the bad. I feel an immense sense of pride and fulfillment when I reflect on the fantastic people who I am so lucky to call my friends. Also, one time I forgot to mention my friend and housemate Eli Goldweber in a paper's acknowledgements section, so I definitely owe him a shout-out here. Eli, I admit it: you are better than me at Mario Kart.

Finally, I'd like to thank my wonderful advisor, Baris Kasikci. I have been fortunate to spend five excellent years learning to be both a better researcher and a better person from Baris. I will forever cherish his unwavering support through the highs and lows of my graduate school experience.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

Cloud providers must isolate each execution context—e.g., a virtual machine (VM)—atop shared hardware. Unfortunately, commodity hardware only strongly enforces context isolation at the architectural level, failing to enforce isolation in the microarchitectural implementation of hardware. The lack of microarchitectural isolation yields a wide range of threats to system security and reliability, including denial-of-service, data loss, data leakage, and even system subversion.

Accordingly, this dissertation presents mitigations for two of the most prominent classes of modern microarchitectural vulnerabilities: transient execution attacks on CPUs—which allow arbitrary data to be leaked from processors via mis-speculation and timing side channels—and Rowhammer—which corrupts and potentially leaks data in DRAM via memory access patterns that produce silicon-level disturbance effects. In particular, *DOLMA* provides the first hardware mitigation against all demonstrated transient execution attacks at the time of publication. *Stop! Hammer Time* presents hardware primitives upon which scalable and flexible software defenses can be built across the taxonomy of Rowhammer mitigations. *MOESI-prime* introduces coherence-induced hammering, the first form of hammering shown to occur in non-malicious code, and provides a corresponding coherence protocol-based mitigation. Finally, *Siloz* isolates different VMs to private DRAM subarray groups (across which Rowhammer attacks are ineffective), thereby preventing inter-VM Rowhammer bit flips.

CHAPTER 1

Introduction

Execution context isolation is a fundamental security and reliability requirement in multitenant computing environments such as the cloud. It is vital that data belonging to one context—e.g., a process, enclave, or virtual machine (VM)—cannot be accessed or modified by another context without explicit permission. However, the level of context isolation provided by today's systems is out-of-sync with the security and reliability needs of cloud providers and customers alike.

Namely, current hardware only strongly enforces context isolation at the architectural (i.e., direct access) level, failing to prevent cross-context interactions at the microarchitectural (i.e., indirect access) level [77, 251, 338]. For instance, while architectural page table permission bits prevent software from explicitly bypassing hardware permission checks, attackers can still read or corrupt data via implicit microarchitectural behavior. Indeed, transient execution attacks [22, 35, 45, 156, 158, 164, 179, 191, 194, 196, 226, 227, 229, 245, 264, 266, 267, 268, 283, 291, 296, 297, 301, 302, 303, 316, 326] demonstrate that remote adversaries can transiently bypass architectural context isolation to leak arbitrary data from another context.

Worsening the problem, data is vulnerable to microarchitectural exploits both on-processor and in-memory. For example, as demonstrated by DRAM disturbances/Rowhammer attacks [51, 52, 53, 59, 72, 75, 91, 92, 98, 124, 125, 131, 147, 147, 152, 152, 152, 160, 160, 167, 180, 185, 215, 218, 219, 233, 241, 250, 261, 269, 288, 289, 299, 300, 325, 328, 346], frequent accesses to an *aggressor* row of DRAM within a short time period can corrupt or leak data in physically-proximate *victim* rows via induced charge leakages, regardless of execution context.

Fundamentally, microarchitectural exploits stem from a pervasive problem in today's systems: commodity microarchitectures do not strongly enforce context isolation (e.g., VM-to-VM). Such microarchitectures have arisen due to outdated threat models and corresponding legacy designs that form the basis of modern hardware, with microarchitectural context isolation coming as an afterthought. Given the ongoing deluge of exploits, it is crucial to provide both hardware and software mitigations for microarchitectural vulnerabilities.

Accordingly, in this dissertation, I present various mitigations for both transient execution attacks and Rowhammer bit flips—two of the most concerning microarchitectural exploits in today's

systems. I first present a hardware-based information flow control defense against transient execution attacks (§1.1). Thereafter, I present hardware-software co-design mitigations for Rowhammer bit flips (§1.2), followed by introducing a new form of hammering that occurs in non-malicious code, as well as a corresponding mitigation (§1.3). Finally, I present a software defense against inter-VM Rowhammer bit flips, deployable on today's (vulnerable) hardware with negligible performance impact (§1.4).

Thesis Statement: Providing microarchitectural context isolation in CPUs and DRAM efficiently and feasibly mitigates hardware-based vulnerabilities in cloud systems.

1.1 DOLMA: Securing Speculation with the Principle of Transient Non-Observability

Modern processors allow attackers to leak data during transient (i.e., mis-speculated) execution through microarchitectural covert timing channels. While initial defenses were channel-specific, recent solutions employ speculative information flow control in an attempt to automatically mitigate attacks via any channel. However, I demonstrate that the current state-of-the-art defense fails to mitigate attacks using speculative stores, still allowing arbitrary data leakage during transient execution. Furthermore, I show that the state of the art does not scale to protect data in registers, incurring 30.8–63.4% overhead on SPEC 2017, depending on the threat model.

I then present DOLMA, the first defense to automatically provide comprehensive protection against all known transient execution attacks. DOLMA combines a lightweight speculative information flow control scheme with a set of secure performance optimizations. By enforcing a novel principle of *transient non-observability*, DOLMA ensures that a time slice on a core provides a unit of isolation in the context of existing attacks. Accordingly, DOLMA can allow speculative TLB/L1 cache accesses and variable-time arithmetic without loss of security. On SPEC 2017, DOLMA achieves comprehensive protection of data in memory at 10.2–29.7% overhead, adding protection for data in registers at 22.6–42.2% overhead (8.2–21.2% less than the state of the art, with greater security).

1.2 Stop! Hammer time: Rethinking Our Approach to Rowhammer Mitigations

Rowhammer attacks exploit electromagnetic interference among nearby DRAM cells to flip bits, corrupting data and altering system behavior. Unfortunately, DRAM vendors have opted for a

blackbox approach to preventing these bit flips, exposing little information about in-DRAM mitigations. Despite vendor claims that their mitigations prevent Rowhammer, recent work bypasses these defenses to corrupt data. Further work shows that the Rowhammer problem is actually *worsening* in emerging DRAM and posits that system-level support is needed to produce adaptable and scalable defenses.

Accordingly, I argue that the systems community can and must drive a fundamental change in Rowhammer mitigation techniques. In the short term, cloud providers and CPU vendors must work together to supplement limited in-DRAM mitigations—ill-equipped to handle rising susceptibility—with their own mitigations. I propose novel hardware primitives in the CPU's integrated memory controller that would enable a variety of efficient software defenses, offering flexible safeguards against future attacks. In the long term, I assert that major consumers of DRAM must persuade DRAM vendors to provide precise information on their defenses, limitations, and necessary supplemental solutions.

1.3 MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads

Adversaries typically mount Rowhammer attacks via instruction sequences that are carefully-crafted to bypass CPU caches. However, I discover a novel form of hammering that I refer to as *coherence-induced hammering*, caused by Intel's implementations of cache coherent non-uniform memory access (ccNUMA) protocols. I show that this hammering *occurs in commodity benchmarks* on a major cloud provider's production hardware, the first hammering found to be generated by non-malicious code. Given DRAM's rising susceptibility to bit flips, it is paramount to prevent coherence-induced hammering to ensure reliability and security in the cloud.

Accordingly, I introduce MOESI-prime, a ccNUMA coherence protocol that mitigates coherence-induced hammering while retaining Intel's state-of-the-art scalability. MOESI-prime shows that most DRAM reads and writes triggering such hammering are unnecessary. Thus, by encoding additional information in the coherence protocol, MOESI-prime can omit these reads and writes, preventing coherence-induced hammering in non-malicious *and* malicious workloads. Furthermore, by omitting unnecessary reads and writes, MOESI-prime has negligible effect on average performance (within $\pm 0.61\%$ of MESI and MOESI) and average DRAM power (0.03%–0.22% improvement) across evaluated ccNUMA configurations.

1.4 Siloz: Leveraging DRAM Isolation Domains to Prevent Inter-VM Rowhammer

Today's cloud DRAM lacks strong isolation primitives, highlighted by Rowhammer bit flips. Systems with state-of-the-art hardware mitigations remain vulnerable, turning cloud providers toward software defenses. However, existing software mitigations incur high performance/memory overhead or suffer from significant gaps in protection.

Accordingly, I introduce Siloz, a hypervisor that uses a *subarray group* DRAM isolation primitive to enable efficient protection against inter-VM Rowhammer. Siloz exploits the insights that (a) Rowhammer can only flip bits in DRAM rows located in the same subarray—not across subarrays—and (b) VMs can be isolated to groups of subarrays without sacrificing bank-level parallelism, a key component of DRAM performance. Siloz prevents inter-VM bit flips by placing each VM's and the host's data into private subarray groups. To additionally ensure that a VM cannot escape its provisioned subarray group(s), Siloz provides integrity protection for extended page tables (EPTs). We show that Siloz's implementation has negligible effect on average performance across various cloud workloads, SPEC CPU 2017, and PARSEC 3.0 (within $\pm$0.5% of baseline Linux/KVM).

1.5 Summary

My dissertation research improves system security and reliability by providing various mitigations for transient execution attacks on CPUs and Rowhammer bit flips in DRAM. The rest of this dissertation is structured as follows. In chapter 2, I describe DOLMA, my proposed mitigation for transient execution attacks. I then highlight the desirability of hardware-software co-design in Rowhammer mitigations and propose hardware primitives to enable scalable and flexible software defenses (chapter 3). I subsequently present *coherence-induced hammering*—a novel form of DRAM hammering which occurs in non-malicious code—and a corresponding mitigation: MOESI-prime (chapter 4). I next introduce Siloz, a hypervisor that provides inter-VM Rowhammer protection by isolating different VMs to private subarray groups (chapter 5). I conclude with a recap of my completed work and my vision for future work (chapter 6).

CHAPTER 2

DOLMA: Securing Speculation with the Principle of Transient Non-Observability

2.1 Introduction

Speculative execution is a crucial performance optimization for modern processors. Unfortunately, the ongoing deluge of transient execution attacks [22, 35, 45, 156, 158, 164, 179, 191, 194, 196, 226, 227, 229, 245, 264, 266, 267, 268, 283, 291, 296, 297, 301, 302, 303, 316, 326] demonstrates that the implementation of speculative execution in commodity processors allows attackers to leak data during transient (i.e., mis-speculated or wrong-path) execution. Specifically, attackers exploit transient micro-ops whose operands are leaked via *covert timing channels*—e.g., hardware structures like the data cache (D-cache), which exhibit operand-dependent timing.

Transient execution attacks can be classified into two primary categories [34]. The first class of attacks rely on delayed handling of microarchitectural exception-like conditions—henceforth referred to as exceptions—to leak data (e.g., Meltdown [179] and similar attacks [35, 227, 229, 245, 266, 268, 283, 296, 297, 301, 302, 316]). In certain commodity processors, speculative reads can access data in spite of—or because of—exceptions. The exception is not handled until the associated micro-op reaches commit, offering attackers a window in which data can be transmitted through covert timing channels. Thankfully, all known Meltdown-type attacks can be thwarted by handling potential exceptions earlier in the pipeline, such that transient reads do not propagate data to dependent micro-ops [179, 315].

The second class of attacks do *not* rely on delayed exception handling, and instead solely exploit hardware mispredictions to leak data (e.g., Spectre [158] and similar attacks [22, 45, 156, 158, 164, 191, 194, 196, 226, 267]). For instance, Spectre v1 [158] shows that an attacker in one security domain can mis-train the branch predictor to transiently bypass a bounds check in a victim domain, thereby allowing micro-ops following a branch to leak victim data. Contrary to Meltdown-type attacks, there is no known comprehensive solution for Spectre-type attacks, apart from disabling speculation.

Because the majority of transient execution attacks use the D-cache as the covert channel [35, 45, 156, 158, 164, 179, 191, 194, 226, 227, 229, 264, 266, 268, 283, 291, 296, 297, 301, 302, 303, 316, 326], initial defenses such as InvisiSpec [331] and others [2, 143, 154, 177, 255, 258, 259] have focused on protecting the D-cache. However, these solutions do not prevent numerous other covert channels [22, 34, 196, 245, 267, 298, 327] from leaking data during transient execution.

Recent solutions [17, 76, 163, 265, 315, 338, 340] acknowledge the shortcomings of cache-centric mitigations, and instead employ *speculative information flow control* to prevent secrets from entering *any* covert timing channel until speculation resolves. Unfortunately, current defenses are not comprehensive. For example, manual defenses [76, 265, 338] require error-prone annotations of secrets to limit performance overhead.

On the other hand, existing automatic defenses [17, 315, 340] suffer from high overhead. As such, they focus on the protection of speculatively-accessed data (e.g., data in memory at the beginning of the speculation window) and fail to comprehensively protect non-speculatively-accessed data (to a first approximation, data in registers at the beginning of the speculation window). For example, NDA [315] conservatively prohibits speculative micro-ops from propagating their results to *any* of their dependent micro-ops until speculation resolves. Thus, NDA eschews knowledge of the microarchitecture to achieve channel-agnostic protection, resulting in high overheads. NDA incurs 22.3% overhead to protect data in memory against Spectre-type attacks, and 100% overhead to supplementally protect against Meltdown-type attacks on SPEC 2017. To provide even partial protection for data in registers, NDA's performance overheads rise to 45–125%, respectively.

The current state-of-the-art defense, STT [338], uses speculative taint tracking to only delay dependent micro-ops that affect processor backend timing (e.g., during execution) or frontend timing (e.g., during fetch) as a function of their operands. Thus, STT is able to significantly improve upon the overheads of channel-agnostic solutions such as NDA and variants of SpecShield [17]. Nonetheless, according to our evaluation, the overhead of protecting data in memory with STT is still 8.7–44.5%, with those figures rising to 30.8–63.4% if one extends STT to protect data in registers.

More importantly, I demonstrate that STT still allows arbitrary data leakages during transient execution. Despite documented transient execution attacks exploiting speculative stores [35, 228, 291, 303], STT assumes stores in isolation are safe unless the processor permits speculative cache line invalidations [291, 340]. However, even without speculative invalidations, stores can still leak information. I demonstrate a novel variant of Spectre [158] that uses a speculative store to transmit data through the TLB, despite STT's protections being enabled. Thus, STT does not yield the comprehensive protection it claims to offer; an attacker can still leak arbitrary data under both its Spectre-type threat model and Meltdown-type threat model.

In this paper, I present DOLMA, the first defense to automatically provide comprehensive pro-

tection against all existing transient execution attacks. DOLMA combines a speculative information flow control scheme with a set of secure performance optimizations, allowing it to protect data in both memory and—optionally—registers at tenable overhead. At a high level, DOLMA extends the microarchitecture to track speculative control and data dependencies, restricting execution as needed to prevent transient operand values from affecting processor timing.

DOLMA's key innovation is ensuring that a time slice on a core provides a unit of isolation in the context of known transient execution attacks. By enforcing a novel principle of *transient non-observability*, DOLMA can allow secure speculative access to select core-local resources (e.g., the TLB, L1 cache, and variable-time functional units) without loss of security.

In line with prior defenses [143, 177, 259, 315, 331, 340], DOLMA's default protection policy assumes a processor immune to Meltdown-type attacks, and therefore only provides mechanisms to mitigate Spectre-type attacks. However, as faulty data propagation is still possible in recent Intel processors [229, 266, 297, 301, 303], DOLMA additionally provides a conservative policy that extends its protections to Meltdown-type attacks.

I evaluate DOLMA on SPEC 2017 [32] in gem5 [24] and McPAT [178], using the same baseline processor as recent solutions [315, 340]. I show that DOLMA incurs negligible (¡1%) area overhead and improves both security and performance over the state of the art [340]. DOLMA offers protection for data in memory at 10.2–29.7% performance overhead (energy: 10.8–29.2%), with protection for data in memory and registers incurring 22.6–42.2% performance overhead (energy: 22.4–40.9%).

In summary, this paper makes the following contributions:

- I present a novel variant of Spectre [158] that uses a speculative store to transmit data through the TLB, demonstrating that the state-of-the-art defense (STT [340]) is still vulnerable to arbitrary data leakages.
- I define and enforce the principle of transient non-observability, enabling secure speculative access to select core-local resources.
- I introduce DOLMA, the first defense to provide automatic comprehensive protection against existing transient execution attacks for data in both memory *and* registers.
- I improve both state-of-the-art security and performance, mitigating all existing transient execution attacks on data in memory at 10.2–29.7% overhead, as well as those on data in registers at 22.6–42.2% on SPEC 2017 [281].

Our implementation and evaluation infrastructure is open-source [181], including our gem5-compatible transient execution attack suite used for penetration testing.

2.2 Background

I first give background on speculative execution in modern out-of-order processors. I then describe how transient (i.e., mis-speculated) execution can be exploited to leak secrets.

2.2.1 Speculative, Out-of-Order Processors

A modern out-of-order (OoO) processor fetches instructions in program order and decodes them into micro-ops. OoO processors keep track of program order via a circular queue called the re-order buffer (ROB). Micro-ops enter at the tail of the ROB in-order upon dispatch, and exit from the head of the ROB in-order upon commit. However, rather than waiting for all elder micro-ops to retire, micro-ops in the ROB issue (i.e., begin executing) as soon as their operands become ready—potentially out of program order. Thus, OoO processors avoid idle execution units, exploiting instruction-level parallelism to improve efficiency over in-order processors.

To further improve efficiency, processors implement control-flow and data-flow speculation to avoid pipeline stalls. For example, the branch prediction unit (BPU) avoids stalls at fetch via control-flow speculation on a branch's target address (i.e., the next program counter) prior to branch resolution. The memory dependency unit (MDU) helps avoid stalls at issue via data-flow speculation on when a load with ready operands can bypass an elder store with unresolved operands.

Additionally, numerous modern processors do not handle exception-like conditions until the associated micro-op reaches commit, thereby implementing *exception speculation*. Specifically, these processors allow read micro-ops (e.g., loads) to broadcast their results to their dependants regardless of potential exceptions (e.g., permission faults).

In the event of mis-speculation, the processor must be able to revert to non-speculative state in order to maintain program correctness. Thus, when the processor detects mis-speculation for a given micro-op, younger entries in the ROB are *squashed*, meaning their effects will never become architecturally-visible. If necessary, the mis-speculated micro-op is re-issued according to non-speculative state, and execution resumes on the correct path.

2.2.2 Transient Execution Attacks

Squashing ensures that transient execution does not become architecturally-visible. However, the *microarchitectural* effects of transient execution may still be visible, depending on the processor implementation. Thus, under certain conditions, attackers can exploit covert timing channels to leak data.

Meltdown-type attacks. Meltdown [179] and similar exploits [35, 227, 229, 266, 268, 283, 296, 297, 301, 302, 316] exploit exception speculation to leak data. By allowing data propagation to

```
1  // assume probe_array is flushed from cache
2  // speculatively access secret (will fault)
3  secret_byte = *kernel_addr;
4  // transmit by caching dependent element
5  tmp = probe_array[secret_byte * 512];
6  ...
7  // later in code, after recovering from fault
8  // infer secret via min time index (cached)
9  for (guess = 0; guess < 256; guess++) {
10   start_time = rdtscp();
11   tmp = probe_array[guess * 512];
12   times[guess] = rdtscp() - start_time;
13 }
14 secret = get_min_index(times);
```

Listing 2.1: Pseudocode for Meltdown [179]. The attacker exploits delayed fault handling to speculatively transmit kernel data via the D-cache timing side channel.

proceed until the exception is handled at commit, processors present a transient attack window during which hardware protections can be bypassed. Attackers ensure that the sensitive data can be later inferred—in spite of squashing—by transmitting the value through microarchitectural state that is not reverted during squashing (e.g., D-cache lines).

A simplified version of Meltdown is shown in Listing 2.1. Key to the attack is the probe array, which the userspace attacker flushes from the D-cache prior to the attack. During the transient execution window (starting at line 3), the attacker is able to load a kernel value due to delayed exception handling. The attacker then uses that kernel value as an index into the probe array, loading the corresponding element into the cache (line 5). Since the cache update is not reverted during squashing, the attacker can later infer the secret value by timing access to each element in the probe array (lines 9–13). The element that is accessed most quickly corresponds to a cache hit, revealing the secret value (line 14).

The recent MDS attacks [35, 266, 301, 302, 303] similarly exploit exception speculation to leak data. However, unlike Meltdown, the address of the data leaked during transient execution does not necessarily correspond to the faulty load's address. Rather, the processor transiently forwards in-flight data: either arbitrary data, or data whose address matches a subset of the faulty load's address bits. CrossTalk [245] builds upon MDS primitives to leak data through the so-called *staging buffer* on Intel CPUs (shared amongst all cores).

Prior work [315, 327, 331] has additionally theorized that various hardware events (e.g., interrupts, microcode assists, Intel TSX transaction aborts, etc.) could produce dangerous transient behavior in a similar way to microarchitectural exceptions. Indeed, during the revision of this paper, the TAA [266, 301] variants of MDS attacks exploited TSX transaction aborts. I consider these events to be special types of microarchitectural exceptions, where all micro-ops succeeding the event should be considered faulty until the processor pipeline is flushed.

Spectre-type attacks. Spectre [158] and similar exploits [22, 45, 156, 158, 164, 191, 194, 226, 267] do not rely on exception speculation, but rather solely exploit control-flow or data-flow spec-

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by caching dependent element
6   tmp = probe_array[secret_byte * 512];
7 }
```

Listing 2.2: Pseudocode for Spectre [158]. The attacker exploits a misprediction in victim code to speculatively transmit victim data via the D-cache timing side channel.

ulation arising from hardware prediction units to leak data. Prior to prediction resolution, a *Spectre gadget* transiently executes, transmitting data through a covert channel. The attacker later recovers the value using techniques similar to those in Meltdown.

A simplified version of Spectre is shown in Listing 2.2, also using the D-cache as the transmission channel. As in Meltdown, the attacker relies on a probe array to help leak the secret value. For simplicity, the attacker and victim share access to the probe array in our example. However, I note that the attacker and victim arrays can be at different physical (and virtual) memory locations; the arrays must merely compete for the same cache lines.

The attacker trains the victim code to transiently jump from a branch (line 2) to a vulnerable gadget (lines 3–6). The branch condition does not have to be related to the secret, and the gadget can be anywhere in the program; for simplicity, I show the gadget in the body of the mispredicted branch. Inside the gadget, vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and loads the corresponding element into the D-cache (line 6). The attacker later times access to each probe array element to retrieve the secret value.

Notably, recent exploits [268, 297] demonstrate that transient execution attacks may combine delayed exception handling and explicit hardware mispredictions to leak data. Because these exploits still rely on exception-like conditions, I consider them to be Meltdown-type, not Spectre-type.

2.3 Problem

Providing secure speculative execution requires that a processor does not leak transient operand values. In this section, I show that no existing defense satisfies this requirement, due to design flaws and security-performance trade-offs.

2.3.1 Cache-Centric Defenses

Since the majority of transient execution attacks leak data through the D-cache, early defenses have focused on the D-cache transmission channel [2, 143, 177, 255, 258, 259, 331]. Though effective in

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by updating TLB via store
6   probe_array[secret_byte * 4096] = tmp;
7 }
```

Listing 2.3: Pseudocode for the access and transmit phases of a new Spectre [158] variant that leaks data through the D-TLB using a store micro-op.

protecting this channel, these works do not mitigate numerous other covert channels [22, 34, 196, 245, 267, 298, 327].

2.3.2 Memory-Centric Defenses

Recent solutions [17, 315, 340] acknowledge the shortcomings of cache-centric defenses, and instead focus on automatically preventing the speculative transmission of secrets via *any* covert channel. However, these solutions only protect data that is speculatively-accessed (e.g., loaded from memory during speculation); they fail to provide comprehensive protection for data in registers at the beginning of the speculation window.

In a transient execution attack on memory, prior work [154, 267, 315, 340] notes that the attacker relies on a two-step Spectre gadget; the gadget first *accesses* the secret by loading it into a register, and then *transmits* the secret via a dependent micro-op whose execution yields operand-dependent timing variations. Thus, attackers seeking to exploit victim programs rely on the presence of such two-step gadgets in the victim binary.

However, in the case of an attack on an unprivileged (e.g., general-purpose) register-based secret, the *access* step can be performed non-speculatively (e.g., the victim loads the secret into the register file prior to the beginning of the speculation window). Thus, if the attacker wishes to leak this register-based secret, they only need to execute the transmit portion of the classic Spectre gadget (line 6 of Listing 2.2). A "register" Spectre gadget is therefore embedded within every "memory" Spectre gadget, meaning *there are at least as many register Spectre gadgets as there are memory Spectre gadgets*.

Despite the risk of register leakages, automatic defenses [17, 340] are often only evaluated on protecting memory-based secrets, as a security-performance trade-off. An exception to this—NDA [315]—demonstrates that adding just *partial* protection for data in registers raises overhead from 22.3–100% to 45–125% on SPEC 2017 (depending on the threat model).

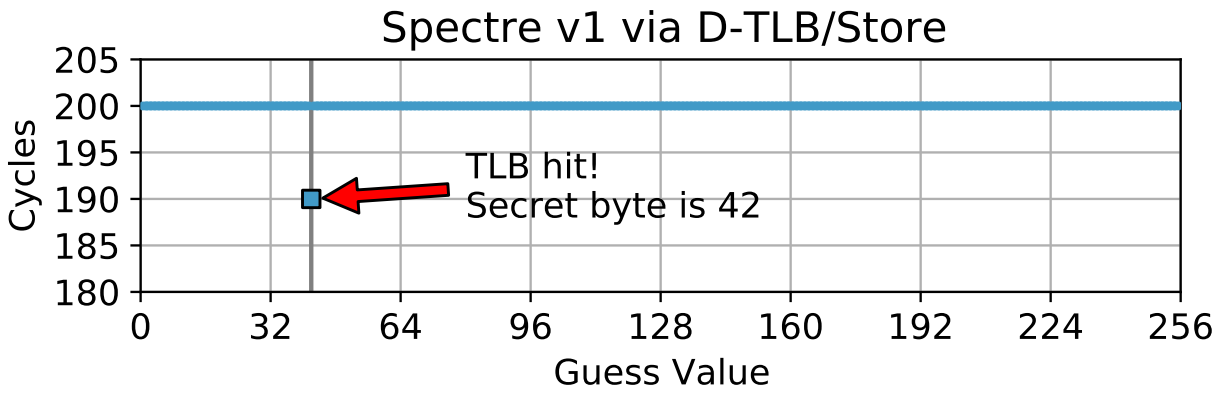

Figure 2.1: Leaking a speculatively-accessed secret through the D-TLB—despite enabling STT [340] protection—via a speculative store in the gem5 simulator [24].

2.3.3 Attacking the State of the Art

The current state-of-the-art defense, STT [340], introduces the concept of speculative taint tracking to protect speculatively-accessed data during transient execution. In this section, I show that arbitrary data can still be leaked in spite of STT.

Despite existing transient attacks exploiting speculative stores [35, 228, 291, 303], STT incorrectly assumes that prohibiting store-triggered speculative cache coherency invalidations is sufficient to prevent transmission via stores in isolation [291, 340]. However, while stores might not speculatively modify cache state on many processors, stores can still leak information via the TLB—including on the processor used in STT's evaluation—among other channels [24, 35, 40, 340].

As a result of this erroneous assumption, STT does not comprehensively prevent transient execution attacks that use stores to transmit a secret-dependent address, whether Spectre-type or Meltdown-type. Here, I demonstrate the most straightforward store-based exploit for brevity. I defer discussion of an additional, more subtle vulnerability in STT to DOLMA's design (§2.5.4).

Listing 2.3 displays the pseudocode for a novel Spectre variant that uses a transient store to leak data through the D-TLB, building on prior work [86] exploiting the TLB side channel. Inside the Spectre gadget (lines 3–6), vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and *speculatively stores the corresponding address in the TLB* (line 6). The attacker later recovers the secret using aforementioned techniques.

The result of running this attack with STT's protections enabled atop our baseline version of the gem5 simulator [24] is shown in Fig. 2.1. As pictured, the Spectre variant clearly leaks the secret byte (42). Thus, *arbitrary data can be leaked during transient execution on STT-protected processors*.

2.4 Scope of Protection

DOLMA considers an attacker exploiting transient execution to leak secrets (i.e., data) through any covert timing channel. DOLMA does not consider non-speculative side channels [70, 71, 89, 243, 332, 333], nor side channels that require physical access to the machine during the attack (e.g., power [159] and EM [223]). While physical side channels are viable sources of leakage, timing channels currently expose a larger threat surface, as they are remotely-exploitable.

DOLMA offers two protection policies, based on the processor's implementation of speculative execution. Technically-speaking, all micro-ops are speculative until they reach the head of the re-order buffer (ROB), at which point they are guaranteed to not be squashed. However, depending on the microarchitecture, not all speculation can leak secrets. For simplicity, in the rest of this text, I assume that "speculation" refers to the subset of speculation that poses a security threat. I precisely define the speculative scenarios under consideration in each protection policy.

DOLMA's protection policies can additionally be tuned based on the data that the user wishes to protect. For instance, if the user only wishes to protect speculatively-accessed data (e.g., data in memory at the beginning of the speculation window, as opposed to data already loaded into registers), they may disable a subset of DOLMA's protections accordingly.

2.4.1 *DOLMA-Default*

DOLMA-Default assumes that the processor inherently mitigates all Meltdown-type attacks by preventing potentially faulty micro-ops from broadcasting (i.e., propagating) their results to dependent micro-ops. Therefore, *DOLMA-Default* only addresses Spectre-type attacks.

DOLMA-Default considers all hardware prediction units (e.g., units that speculate on control dependencies or data dependencies) to be sources of speculation. Thus, *DOLMA-Default* considers any micro-op fetched (control dependency) or issued (data dependency) as a result of a hardware prediction unit to be a potential source of leakage. While the exact units are implementation-specific, I detail generalizable considerations for both a typical control-flow prediction unit (the branch prediction unit) and a typical data-flow prediction unit (the memory dependency unit).

Branch Prediction Unit (BPU). The BPU can induce transient execution in three scenarios. First, the BPU can mispredict whether a branch is taken, as shown in Fig. 2a. Second, the BPU can mispredict the target of the branch. Thus, *DOLMA-Default* must prevent information leakages stemming from micro-ops following a branch in the ROB, until the prediction resolves as correct or the processor squashes.

In the third scenario, the BPU can mispredict a non-branch to be a branch (i.e., before decoding the non-branch's opcode, the BPU mispredicts that the instruction is a branch and fetches from the wrong address). However, because the misprediction is realized at decode (an in-order stage),

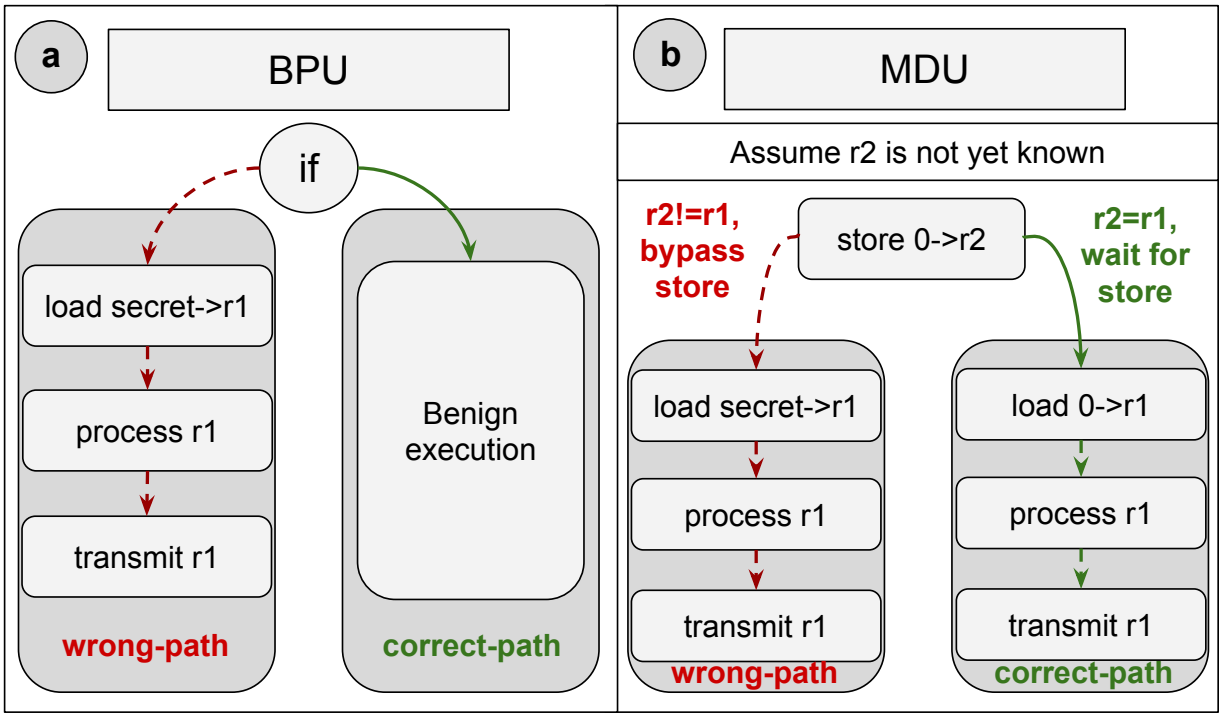

Figure 2.2: Examples of transient execution arising from hardware mispredictions in the (a) branch prediction unit (BPU) and (b) memory dependency unit (MDU).

the younger (transient) micro-ops can be squashed prior to operand resolution. Thus, operand-dependent timing variations are not possible.

Memory Dependency Unit (MDU). The MDU can induce transient execution for one or two reasons, depending on the memory consistency model: speculative store bypass (SSB) and speculative load bypass (SLB).

Speculative Store Bypass (SSB): The MDU may induce transient execution by allowing a load to bypass an earlier, unresolved store [226, 331], as shown in Fig. 2b. If the store resolves to an address used by the load, the load and its dependants must be squashed. Accordingly, *DOLMA-Default* must prevent leakages stemming from any load-dependent micro-ops, until all prior stores resolve.

Notably, *DOLMA-Default* need not prevent leakages stemming from the load itself in bypass scenarios, unless the load is already under consideration (e.g., due to following an unresolved branch). To understand this intuition, I consider the two possible scenarios for a speculative store bypass attack. First, the load can be used to access a secret in memory. In this case, the load relies on a dependent micro-op to transmit the secret, meaning the load itself need not be considered.

Second, the load can be used to leak the (register-based) load address, which is presumed to be a secret. However, speculation does not change the load's address; it only potentially changes the value returned by the load. Even if the load is mispredicted, it will be re-executed with the same

operand—the secret. Thus, this scenario is a non-speculative side channel, and is explicitly outside of DOLMA's threat model.

Speculative Load Bypass (SLB): The MDU may induce transient execution for a second reason in memory consistency models that enforce a form of total store ordering. In such models, transient execution can arise when a younger load bypasses an elder, unresolved load [259, 327]. If the elder load resolves to an address used by the younger load—and the cache line for the address is invalidated in the interim—the younger load and its dependants must be squashed to enforce memory consistency.

DOLMA-Default only considers dependent micro-ops of SSB loads, and not the dependants of SLB loads. SSB allows a single thread of execution to transiently read secrets explicitly overwritten in program semantics, posing an obvious security threat. On the other hand, an SLB load only reads stale data if the cache line is invalidated by another core. For memory shared among cores, such writes could occur at an arbitrary time. Thus, the programmer cannot assume the stale data has been overwritten before these loads execute, and must therefore reason about the safety of dependent micro-ops irrespective of speculation. As such, *DOLMA-Default* does not consider dependants of SLB loads.

2.4.2 *DOLMA-Conservative*

Despite the existence of a comprehensive solution for all Meltdown-type attacks (namely, preventing data propagation in the presence of potential microarchitectural exception-like conditions), faulty data propagation is still possible in recent Intel processors [229, 266, 297, 301, 303]. Therefore, *DOLMA-Conservative* assumes that loads and load-like privileged register reads can transiently bypass exception-like conditions, inducing exception speculation until they retire. Thus, in addition to the speculation considerations of *DOLMA-Default*, *DOLMA-Conservative* prevents leakages stemming from all dependants of a load-like micro-op, until the load-like micro-op retires.

2.4.3 Simultaneous Multi-Threading

In the context of transient execution attacks, simultaneous multi-threading (SMT) can be used to access secrets (e.g., MDS attacks [35, 266, 301, 302, 303] can access secrets from a sibling logical core) or to transmit secrets (e.g., SMotherSpectre [22] can transmit a secret via issue port contention between attacker and victim sibling logical cores). Under DOLMA as well as prior speculative information control flow defenses [17, 315, 340], SMT *accesses* are safe, provided that the accessed data cannot modify a transmission channel (e.g., the D-cache) as a function of its value during speculation.

This leaves the question of how to deal with speculative SMT transmission channels. SMT contention creates a myriad of potential transmission channels—both speculative and non-speculative—via resource contention for core-local resources such as the TLB, L1 cache, and each functional unit. Thus, in the presence of SMT, prior work makes the performance-inhibiting assumptions that (1) *all* unsafe TLB/L1 accesses must be delayed, and (2) *no* unsafe micro-ops may use fast-path optimizations (e.g., variable-time arithmetic) [17, 315, 340].

However, DOLMA shows that these assumptions are unnecessary in the context of existing transient execution attacks. Even without DOLMA, potential SMT transmission channels are comparatively difficult to exploit in production environments. Namely, the attacker and victim must be co-scheduled on the same physical core and contend for the same secret-dependent resource on the exact same processor cycle. Indeed, unlike notoriously-reliable channels such as the D-cache [35, 45, 156, 158, 164, 179, 191, 194, 226, 227, 229, 264, 266, 283, 291, 296, 297, 297, 301, 302, 303, 316, 326], speculative transmission via SMT contention has only been demonstrated by a single attack (SMotherSpectre [22]). Nonetheless, I show that DOLMA's design naturally mitigates SMotherSpectre in §2.5.4.

2.5 Design

DOLMA has two primary goals. First, in the context of each protection policy, the value of a transient operand (i.e., an operand of a micro-op that will be squashed) cannot affect the timing of non-transient micro-ops. Second, in order to make such security tenable for real-world systems, DOLMA must incur as little performance overhead as possible.

At a high level, DOLMA adds state to track the speculation status of each micro-op in the re-order buffer (ROB). DOLMA then uses this state to restrict (e.g., delay) execution, such that transient operands cannot observably affect timing.

Given the overhead of related defenses [17, 315, 340], DOLMA's key contribution is enforcing a novel principle of transient non-observability that obviates the need to delay execution in certain contexts. In doing so, DOLMA enables protection to scale to registers with tenable performance overhead.

In this section, I first introduce the principle of transient non-observability (§2.5.1). I then provide the classifications for micro-ops that DOLMA uses to enforce this principle (§2.5.2). With these definitions, I explain DOLMA's optimizations for traditional sources of transmission (§2.5.3). I subsequently identify a remaining vulnerability in the state of the art [340] and present DOLMA's mitigations for this and related channels (§2.5.4). Finally, I specify the microarchitectural state and logic used to appropriately restrict speculative execution (§2.5.5) and lift these restrictions when speculation resolves (§2.5.6).

2.5.1 Transient Non-Observability

To prevent transmissions of secrets, DOLMA enforces a novel principle of *transient non-observability*. With regards to DOLMA's timing channel protection policies, transient non-observability is achieved by ensuring that the value of a transient (i.e., destined to squash) operand cannot affect the cycle upon which a non-transient micro-op commits—thereby preventing timing-based leakages.

More precisely, transient operand values must not cause timing variations in non-transient micro-ops via (a) out-of-order contention for core-local resources, (b) simultaneous uncore/offcore resource access, or (c) persistent state modifications—i.e., modifications that survive the transient window. Notably, such leakages can occur both via data flows (e.g., a specific microarchitectural buffer entry is accessed/modified based on a secret operand) or control flows (e.g., state is only modified on a conditional path, revealing the value of a secret conditional predicate).

In this sense, DOLMA's principle of transient non-observability is similar to the principle of speculative non-interference [93, 340, 341]. The key difference is that prior work assumes *all* operand-dependent timing variations (e.g., variable-time arithmetic and TLB/cache accesses) are inherently unsafe, as an SMT adversary (i.e., an adversary executing simultaneously on the same physical core) can observe these variations via core-local contention. This limitation yields designs that stall all variable-time micro-ops until speculation resolves, inhibiting performance [17,315,340]. However, as I will demonstrate (§2.5.4), DOLMA naturally mitigates SMotherSpectre [22]—the only transient execution attack to have demonstrated transmission via SMT contention—enabling a set of secure performance optimizations over prior work.

2.5.2 Micro-op Classification

Inducive and Resolvent Micro-ops. In order to identify the beginning and end of each speculation window, DOLMA requires the manufacturer to denote a set of *inducive* and *resolvent* micro-ops. An inducive micro-op is any micro-op that can induce speculation, such as a control-flow micro-op (branch prediction) or a load (memory dependency prediction, value prediction, etc.). More specifically, a control-flow micro-op—or branch—is any micro-op that can explicitly alter program control flow (e.g., a jump, call, or return); branch prediction encompasses the BPU structures used to predict the result of these micro-ops (e.g., the branch history table [BHT], branch target buffer [BTB], and return stack buffer [RSB]).

A resolvent micro-op is any micro-op that can resolve speculation. Note that the same micro-op can induce and resolve a speculation window (e.g., a control-flow micro-op induces speculation at fetch and resolves speculation at execute). In other cases, a speculation window can be induced and resolved by different micro-ops (e.g., memory dependency speculation is induced by loads and

resolved by stores).

Given a specific microarchitecture, enumerating inducive and resolvent micro-ops is trivial: the manufacturer must already define an exhaustive list of these micro-ops in order to implement their processor according to its ISA specification. If the manufacturer were to omit such a micro-op, transient micro-ops would be able to retire their effects to architectural state, violating the ISA specification and thus program correctness.

Unsafe Micro-ops. In DOLMA, *unsafe* micro-ops are speculative micro-ops whose operand values can be transmitted during transient execution via corresponding timing variations. Unsafe micro-ops can be further classified as *backend-unsafe* (e.g., loads can transmit through backend channels such as the D-cache), *frontend-unsafe* (e.g., control-flow micro-ops can transmit through frontend channels such as the BTB), or both.

Because DOLMA considers timing channels, micro-ops are only classified as unsafe in the context of timing leakages. However, mitigating other operand-dependent channels would simply require the manufacturer to denote additional micro-ops as unsafe (e.g., via microcode updates).

While the exact set of unsafe micro-ops is microarchitecture-specific, I discuss common examples in modern processors. I precisely define the set of unsafe micro-ops for the microarchitecture used in our evaluation in §2.7, manually enumerating this set using the aforementioned criteria for transient non-observability (i.e., operand-dependent out-of-order contention for core-local resources, simultaneous uncore/offcore resource access, and persistent state modifications). Notably, this set includes all micro-ops classified as high covert channel risk (CCR) in prior work [17]. Furthermore, unlike the state of the art [340] evaluated on the same processor, DOLMA's set of unsafe micro-ops includes all applicable micro-ops whose operands are leaked in documented transient execution attacks.

For an arbitrary microarchitecture, exhaustively identifying unsafe micro-ops requires a formal timing analysis of the RTL code, and is ongoing work. The state of the art [83] requires the programmer to manually annotate portions of the circuit description, limiting scalability to modern processors. Therefore, formal verification of DOLMA's security on an arbitrary processor necessitates advancements in these methods.

2.5.3 Optimizations for Traditional Backend Channels

Existing speculative information flow control defenses [17, 315, 340] delay all unsafe micro-ops until speculation resolves. In select cases, DOLMA likewise delays unsafe micro-ops. However, DOLMA's principle of non-observability—combined with minor modifications to the processor—allows a restricted form of speculative execution in two key scenarios. I describe the optimizations here, and show that they do not directly produce backend timing variations. I demonstrate that the

optimizations cannot influence frontend state (and thus, cannot indirectly produce backend timing variations) in §2.6.

Variable-Time Execution. On a traditional processor, all micro-ops that vary execution time as a function of their operands would be unsafe. While many micro-ops only produce core-local modifications that are reverted upon squashing, they may still alter the cycle upon which other micro-ops retire due to out-of-order contention for core-local resources.

More precisely, the operand-dependent contention produced by variable-time computation is problematic when it occurs between a younger (transient) micro-op and an elder (non-transient) micro-op. While the pipeline frontend is in-order, such out-of-order contention is indeed possible in the processor backend (i.e., issue and onwards).

Accordingly, to obviate unsafe backend contention, DOLMA employs a simple policy. At a high level, DOLMA's strategy is to ensure that—when an elder and younger micro-op compete for the same backend resource—the elder micro-op is unconditionally granted access to the resource. While I cannot list every possible example of backend contention, I describe our techniques for issue and writeback ports that generalize to other contention sources.

At issue, elder micro-ops can forcibly evict younger (unsafe) micro-ops from execution units when no units would otherwise be available; the younger micro-ops are then re-issued once safe. At writeback, a priority queue ensures that the eldest micro-ops obtain access to writeback ports each cycle. That is, if there are P ports and N micro-ops ready to writeback (where $N > P$), the P eldest micro-ops obtain the ports.

With this policy, the operands of variable-time micro-ops are transiently non-observable if they (a) do not affect uncore/offcore resource accesses, and (b) do not produce operand-dependent persistent state modifications. Although these criteria conventionally include variable-time ALU micro-ops, other micro-ops clearly remain unsafe, even if core-local. For example, NetSpectre [267] shows that AVX micro-ops reset a persistent powerdown timer upon execution, meaning (operand-dependent) timing variations in AVX execution would ultimately produce (operand-dependent) persistent modifications. Thankfully, in such cases where updates are off the critical path (i.e., not required for the speculative computation), DOLMA can mitigate the channels without performance loss by only performing the updates upon commit. I discuss how DOLMA prevents leakages via conditional (e.g., control-dependent) usage of resources like the AVX powerdown timer in §2.5.4.

Delay-on-Miss. Memory micro-ops (loads and stores)—produced by a variety of high-level instructions [245]—pose a greater challenge, as they can both access uncore/offcore resources *and* produce persistent state modifications that greatly affect performance. For example, memory micro-ops can produce speculative, operand-dependent contention for or modifications to the D-TLB, D-cache, load-store queue, memory dependency unit, prefetching infrastructure, global

staging buffer, and associated metadata for these structures (e.g., replacement policy data). Thus, speculative memory micro-ops would normally be unable to execute without leaking secrets. However, it is possible to avoid delaying memory micro-ops in the common case without loss of security.

DOLMA novelly applies the technique of "delay-on-miss" [259] to speculative stores, building on prior work that uses delay-on-miss to achieve efficient protection for speculative loads. At a high level, delay-on-miss allows speculative memory micro-ops that hit in first-level core-local structures (e.g., the L1 TLB and—in the case of loads—L1 cache) to execute without stalling until speculation resolves. A speculative memory micro-op that misses in these structures vacates its execution unit and is placed into a dedicated stall queue (as can already be done to mask the latency of TLB misses/page table walks). Such a design allows other in-flight memory micro-ops to proceed with execution. When speculation resolves, the stalled memory micro-op is re-issued without restriction.

Importantly, DOLMA ensures that memory micro-ops do not affect replacement policy metadata or memory dependency predictions until speculation resolves, thereby eliminating these potential channels. Furthermore, if a speculative memory micro-op triggers a prefetch, the prefetch is likewise constrained to delay-on-miss behavior. Finally, because only core-local memory micro-ops are legal, the global staging buffer cannot be altered. Thus, delay-on-miss prevents transmission at two levels: the explicit channels of speculative modifications to TLB and cache entries, as well as more subtle channels of speculative updates to associated state.

2.5.4 Mitigating Remaining Sources of Transmission

Store-to-Load Forwarding. As noted in prior work [340], store-to-load forwarding provides an additional source of backend leakage for memory micro-ops. If a load has a complete match with an unsafe store in the store buffer, the load will not issue a memory request, and will instead use the data from the store buffer. Thus, the decision to (not) issue a memory request reveals the store's address operand.

DOLMA's contribution in this regard is to identify and address another source of leakage via store-to-load forwarding. Namely, prior work [340, 342] does not handle the case of a partial hit (i.e., where a strict subset of the load's address range is found in the store buffer), instead erroneously assuming that the only two possible cases are a complete hit or miss. However, in the case of a partial hit, neither the store buffer nor lower levels of the memory hierarchy hold the correct data in its entirety. Thus, depending on how the microarchitecture handles partial hits, the load may stall until the store completes, revealing information about the store's address via timing.

Fortunately, combined with DOLMA's protections for variable-time execution, the same pro-

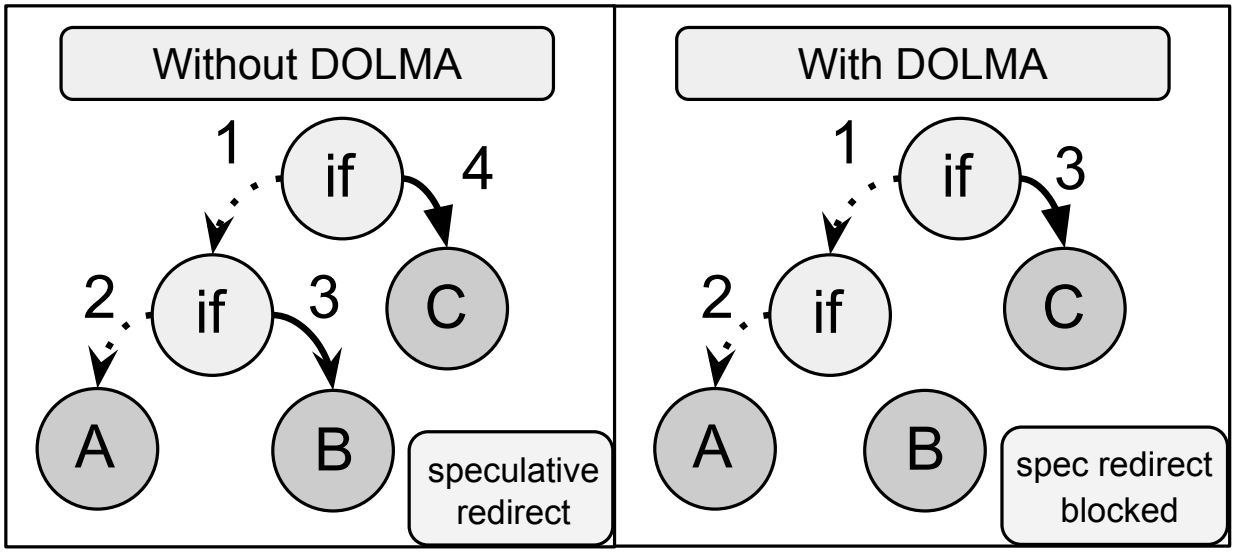

Figure 2.3: Without DOLMA (left), the processor speculatively redirects fetch from A to B, dependent upon a transient predicate value. With DOLMA (right), speculative fetch redirects are blocked until speculation resolves (C), thereby preventing predicate-dependent execution. Dashed lines indicate predictions, while solid lines indicate fetch redirects.

tection mechanism works for both total and partial store buffer hits in the presence of stalling. That is, the processor unconditionally issues the load to the cache hierarchy, and simply ignores the response in the event of an unsafe buffer hit. If the hit was partial (meaning the buffer does not contain all necessary data), the load re-issues once the store is safe and complete.

Speculative Fetch Redirects. Control-flow micro-ops and any remaining inducive/resolvent micro-ops provide common examples of frontend-unsafe micro-ops, because these micro-ops can leak their operands via speculative fetch redirects [340], as shown in Fig. 2.3. For instance, if a speculative (e.g., nested) control-flow micro-op resolves as incorrect, the micro-op must signal to the frontend to redirect fetch to the appropriate program counter. However, the new PC is determined by the control-flow micro-op's predicate, meaning such a redirect leaks the predicate via dependent updates to frontend covert timing channels (e.g., the I-TLB, I-cache, and BPU), as well as potential backend covert channels (e.g., resets of the AVX powerdown timer via subsequent conditional execution [267]).

Like the state of the art (STT [340]), DOLMA additionally provides protection against more subtle sources of speculative fetch redirects. Consider the case of redirects caused by memory ordering violations (i.e., load-store aliasing, where an inducive load incorrectly bypasses an unresolved store). Such a redirect can reveal information about the load's address operand (namely, that it conflicted with that of a prior store). Thus, the redirect is clearly unsafe while the load itself is unsafe . However, even if the younger load is safe (for instance, not dependent upon any inducive loads), the elder store can still be unsafe. Accordingly, a redirect in this scenario leaks

	micro-ops			DOLMA-Default					DOLMA-Conservative				
a	1	load	r0 -> r1	-	-	-	-		U	-	-	-	U: until retirement
	2	add	r1, r2	-	-	-	-		-	-	D	-	D: until line 1 retires
	3	jump	r1	U	-	-	-	U: until executed/squashed	U	-	D	P	D+P: until line 1 retires
b	1	store	r2 -> r3	-	-	-	-		-	-	-	-	
	2	load	r0 -> r1	U	-	-	-	U: until line 1 executes/squashed	U	-	-	-	U: until retirement
	3	add	r2, r3	-	-	-	-		-	-	-	-	
	4	load	r1 -> r4	-	-	D	-	D: until line 2 resolves	U	-	D	-	D: until line 2 retires
c	1	cmp	0x0, r0	-	-	-	-		-	-	-	-	
	2	jne	r1	U	-	-	-	U: until executed/squashed	U	-	-	-	U: until executed/squashed
	3	load	r2 -> r3	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	4	load	r4 -> r5	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	5	jump	r3	U	C	-	P	P: until line 2 resolves	U	C	D	P	D+P: until lines 3 retires

Figure 2.4: Comparing *DOLMA-Default*'s and *DOLMA-Conservative*'s handling of speculation status in the ROB in three scenarios. **U** = **U**nresolved, **C** = Control-Dependent, **D** = **D**ata-Dependent, and **P** = **P**ending-Redirect. Example (a) shows a non-retired load. Example (b) shows an unresolved speculative store bypass. Example (c) shows an unresolved branch, with a nested branch blocked due to a speculative fetch redirect (line c5).

the store's address operand in an identical fashion to that of an unsafe load and must likewise be delayed. Although STT's implementation code [342] allows the redirect before the store is safe, I note that STT's design correctly mentions the need to delay such redirects until both the load and store are safe [340].

By comprehensively prohibiting speculative fetch redirects, DOLMA mitigates all channels that rely on conditional transient execution to leak data (e.g., the AVX powerdown timer [267]). Notably, this protection likewise mitigates SMotherSpectre [22], the only transient execution attack to have demonstrated transmission via SMT contention. In order to create reliable contention on issue ports, SMotherSpectre uses a secret-dependent speculative redirect to fetch and issue micro-ops. In the context of Fig. 2.3, the speculative redirect is performed based on the secret being zero or non-zero. When the fetched micro-ops (either A or B) reach issue, they compete with micro-ops from the adversary's sibling logical core for different ports. However, the specific ports contended depend on the (different) opcodes between A and B, thereby revealing the secret value. Under DOLMA, this and similar scenarios are impossible, as speculative fetch redirects are prevented.

2.5.5 Enforcing Restrictions

Both *DOLMA-Default* and *DOLMA-Conservative* must restrict unsafe micro-ops that are control-dependent or data-dependent upon inducive micro-ops, delaying unsafe micro-ops that would produce observable modifications to the microarchitecture. As previously-mentioned, branch speculation provides an example of control-dependency restriction: any unsafe micro-op following a branch (e.g., jump, call, or return) in the ROB must be restricted. Memory dependency speculation

provides an example of data-dependency restriction: *DOLMA-Default* must restrict the dependants of loads that bypass stores during execution. *DOLMA-Conservative* expands this mechanism to all loads (and load-like privileged register reads) in order to additionally handle exception speculation.

In order to track the speculation status of each micro-op in the pipeline, DOLMA conceptually extends each ROB entry with four bits, as shown in Fig. 2.4: **U**nresolved, **C**ontrol-Dependent, **D**ata-Dependent, and **P**ending-Redirect. If a micro-op is squashed, the extra bits are ignored.

Unresolved. DOLMA marks an inducive micro-op as *unresolved* until (a) its associated speculation window resolves, and (b) all elder micro-ops are also resolved. Assuming all elder micro-ops are resolved, a control micro-op resolves when it is executed. Under *DOLMA-Default*, loads are only inducive if they are issued as a result of a hardware prediction unit (e.g., speculative store bypass). Thus, such loads resolve when the corresponding prediction resolves (e.g., the bypassed store executes). Under *DOLMA-Conservative*, all load-like micro-ops are assumed to be unresolved until they retire, in order to handle exception speculation.

Control-Dependent and Data-Dependent. Speculative control dependencies can be easily tracked in DOLMA: any micro-op following an unresolved branch in the ROB is control-dependent on that branch, until the next branch introduces a new set of control dependencies.

Like prior work [340], DOLMA tracks speculative data dependencies via the register rename table. In particular, if a micro-op X consumes the output of an inducive micro-op (or its dependants), then DOLMA marks X as data-dependent. Data dependency status is propagated during broadcast (i.e., wakeup of dependent micro-ops).

Notably, reservation station entries for unsafe micro-ops are also extended with the OR of their micro-op's control-dependent and data-dependent status bits. The processor uses this signal to ensure that unsafe micro-ops do not transmit information. For instance, outgoing memory requests are tainted for unsafe micro-ops, such that the L1 cache will know to return without fetching from L2 upon a miss. As another example, DOLMA uses the dependency status—along with ordering information from the ROB—to prevent unsafe backend contention (e.g., issue/writeback port contention between elder micro-ops and younger unsafe micro-ops).

When an unsafe micro-op is issued, a copy of its issue queue entry is placed into a dedicated *unsafe* queue for in-flight unsafe micro-ops. If an unsafe micro-op executes without stalling, its unsafe queue entry is freed. For unsafe micro-ops that cannot complete for safety reasons, each queue entry holds the index of its youngest unresolved inducer. Such a design allows for efficient wakeup when the micro-op becomes safe [340]. Specifically, if a stalled micro-op's youngest inducer is resolved, the inducer broadcasts its ROB index to this queue such that dependent micro-ops are marked as ready to issue.

Pending-Redirect. Finally, when a frontend-unsafe micro-op would initiate a fetch redirect, its ROB entry is instead marked as *pending-redirect*. Like backend-unsafe micro-ops, the frontend-

unsafe micro-op also vacates its execution unit and awaits a safety broadcast.

2.5.6 Clearing Speculative Status

DOLMA only clears micro-ops when they become non-speculative in the context of DOLMA's threat models. For control-dependent micro-ops, this means that all elder control-flow micro-ops must be resolved. For data-dependent micro-ops, this means that all elder loads and associated resolvent micro-ops (e.g., stores) must be resolved.

When stalled backend-unsafe micro-ops are cleared, they are marked as ready to re-issue from the stall queue. When pending frontend-unsafe micro-ops are cleared, they signal their delayed redirect. Cleared micro-ops compete with the regular stream of micro-ops for backend ports. As previously stated, elder micro-ops are given preference during (re-)issue; however, DOLMA does not increase the issue width.

2.6 Security Analysis

The goal of our supplemental security analysis is to show that the optimizations afforded by our notion of non-observability do not introduce speculative timing channels in the context of DOLMA's protection policies. I base our reasoning on features of the baseline processor [24, 139] used in similar defenses [315, 340] (including our own), and argue that the same logic can be applied to any microarchitecture satisfying the general properties I describe here.

DOLMA introduces two optimizations due to non-observability. First, DOLMA allows for variable-time arithmetic. Second, DOLMA uses delay-on-miss [259] for speculative loads and stores. I demonstrated that these optimizations cannot directly produce timing variations in processor backend state in §2.5. Here, I demonstrate that transient execution cannot influence frontend timing on a DOLMA-protected processor (and thus, cannot indirectly produce backend timing variations).

Proof Sketch. On our processor, four events can influence frontend state on any given cycle. I show each event is invariant of transient values in the context of DOLMA's protection policies.

(1) Backend Redirect: The backend can redirect fetch to a new PC as a result of a predicate resolution (e.g., branch or memory dependency). DOLMA delays fetch redirects until speculation resolves, meaning transient micro-ops in the backend cannot initiate a fetch redirect. Furthermore, since elder micro-ops are given preference for backend resources, a transient micro-op cannot affect the length of the speculation window (and thus, cannot influence the cycle upon which a backend redirect is performed). Thus, backend fetch redirects are invariant of transient data.

(2) Frontend Redirect: The frontend can redirect fetch to a new PC as a result of a branch

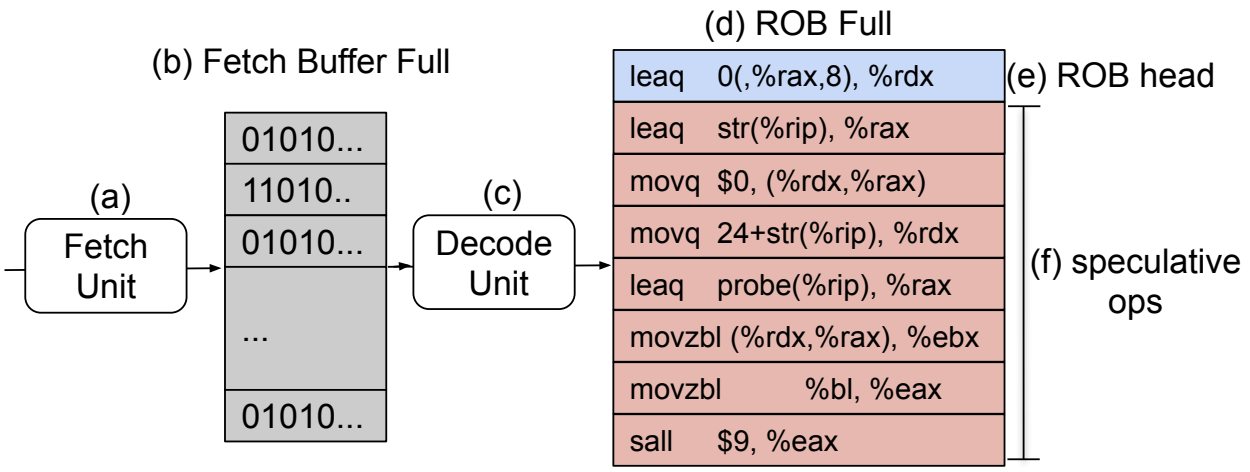

Figure 2.5: A simplified example of how a pipeline backup can cause the fetch buffer to fill.

prediction. Since DOLMA delays fetch redirects until speculation resolves, DOLMA prevents transient data from entering the BPU. Thus, frontend fetch redirects are invariant of transient data.

(3) Full Fetch Buffer: The processor may not increment the PC on a given cycle if the fetch buffer (i.e., the buffer for fetched instructions, before they are decoded and inserted into the ROB) is full. Delaying fetch redirects until speculation resolves—coupled with giving elder micro-ops priority in the backend— prevents transient operands from affecting the the processor frontend state (including the fetch buffer). Thus, it suffices to show that transient micro-ops cannot indirectly influence the state of the fetch buffer via a pipeline backup.

I trace back from the "full fetch buffer" scenario shown in Fig. 2.5 to demonstrate that only non-speculative micro-ops can cause the fetch buffer to fill. The fetch unit **(a)** fetches instructions into the fetch buffer **(b)**. The fetch buffer becomes full when the decode unit **(c)** cannot process instructions on a given cycle. The decode unit cannot process instructions if the ROB **(d)** is full. Finally, the ROB is full if the micro-op at the head of the ROB **(e)** cannot retire.

However, the head of the ROB is—by definition—non-speculative. Thus, this is only a concern if younger (speculative) micro-ops prevents the head from retiring. Since DOLMA gives elder micro-ops priority in the backend, such a scenario is impossible. Therefore, only non-speculative micro-ops can cause the fetch buffer to fill, meaning the fetch buffer is invariant of speculative micro-ops **(f)**, and thus transient data.

(4) Variable Fetch Latency: The processor may not increment the PC on a given cycle if a fetch request is delayed (e.g., due to an I-TLB or I-cache miss). Fetch latency is a function of frontend state (e.g., the PC, BPU, I-TLB, and I-cache), which by (1)–(3), is invariant of transient data. Thus, fetch latency is invariant of transient data.

Therefore, processor frontend state is invariant of transient operand values in the context of DOLMA's threat models.

Parameter	Value
Architecture	x86-64 at 2.0 GHz
OoO Core (No SMT)	8-issue, 32 LQ entries, 32 SQ entries, 192 ROB entries, 4096 BTB entries, 16 RAS entries
OoO Core (2-SMT)	8-issue, 16 LQ entries per thread, 16 SQ entries per thread, 91 ROB entries per thread, 4096 BTB entries (dynamically partitioned), 16 RAS entries per thread
L1-I/L1-D Cache	32 KB, 64B line, 8-way set associative (SA), 4 cycle round-trip (RT) latency, 1 port
L2 Cache	2 MB, 64 B line, 16-way SA, 40 cycle RT latency
DRAM	50 ns response latency

Table 2.1: gem5 simulation configuration.

2.7 Evaluation

I evaluate DOLMA's gem5 [24] implementation against the SPEC 2017 [281] benchmark suite. I estimate area and energy with McPAT [178], incorporating recommended changes for increased accuracy [324]. I sample performance throughout each benchmark's execution via the Lapidary simulation sampling framework [224, 225], which employs the SMARTS methodology [323]. More specifically, Lapidary converts periodic GDB coredumps from each benchmark's execution on real hardware into gem5 checkpoints. Following the methodology used in NDA [315] (a prior speculative information flow control defense), I configure Lapidary to warm microarchitectural structures for $5,000,000$ instructions before measuring the performance of $100,000$ instructions, repeated for each checkpoint.

I evaluate DOLMA with and without simultaneous multi-threading (SMT) enabled. I generate SMT workload pairings from the SPEC 2017 benchmarks using the "Balanced Random" methodology developed by Velasquez et al. [304]. This methodology ensures that each benchmark appears an equal number of times across all pairings.

In line with prior speculative information flow control defenses [315, 331, 340], I use gem5's OoO processor as our baseline. The processor's set of inducive micro-ops includes control-flow micro-ops (i.e., jumps, calls, and returns) and loads, while its resolvent micro-ops include control-flow micro-ops and stores. Its set of unsafe micro-ops includes control-flow micro-ops, loads, and stores—consistent with the micro-ops identified as high covert channel risk (CCR) in prior work [17]. The processor configuration is listed in Table 2.1.

I additionally compare the performance of DOLMA to the state-of-the-art speculative information flow control defense, STT [340]. As STT provides memory-only protection, I extend STT to enable optional protection for registers. I compare DOLMA to STT under both memory-only protection modes (M) as well as memory and register (M+R) modes.

Although STT's gem5 implementation is publicly-available [342], it was necessary to port STT as modifications to DOLMA for two key reasons. First, STT's baseline performance differs significantly from that of prior speculative information flow control defenses, rendering fair comparisons impossible. For example, I found that for the *mcf* benchmark, STT's baseline yielded approximately 30% higher average cycles-per-instruction compared to the baseline of NDA (and our

Defense	Overhead	Overhead-SMT	*Speculation*			
			Control (M)	Control (R)	Data	Exception
Baseline OoO	0±3.8%	0±3.0%				
STT-*Spectre* (M)	8.7±4.2%	3.2±3.2%	○			
***DOLMA-Default* (M)**	10.2±4.3%	3.4±3.2%	●		●	
STT-*Futuristic* (M)	44.5±4.6%	25.5±3.6%	○		○	○
***DOLMA-Conservative* (M)**	29.7±4.7%	16.2±3.5%	●		●	●
STT-*Spectre* (M+R)	30.8±5.0%	17.3±3.6%	○	○		
***DOLMA-Default* (M+R)**	22.6±4.8%	9.8±3.4%	●	●	●	
STT-*Futuristic* (M+R)	63.4±5.0%	36.8±3.8%	○	○	○	○
***DOLMA-Conservative* (M+R)**	42.2±5.4%	22.4±3.7%	●	●	●	●

● Mitigates all existing attacks
○ Mitigates all existing attacks, except select transmissions via stores

Table 2.2: DOLMA compared to STT [340] in terms of total CPI overheads and mitigated attacks, using memory-only protection variants (M) as well as memory and register protection variants (M+R). *Control* transient execution attacks refer to transient execution arising from branch predictions, differentiated by whether memory or registers are leaked. *Data* transient execution attacks refer to transient execution arising from data predictions (e.g., memory dependency speculations). *Exception* transient execution attacks refer to Meltdown-type attacks that exploit delayed microarchitectural exception handling. Overhead ranges reflect 95% confidence intervals.

own), significantly skewing results. Second, SMT support for x86-64 is not functional in the STT prototype.

2.7.1 Performance Evaluation

Single Thread. The per-benchmark geometric mean cycles per instruction (CPI) for *DOLMA-Default* and *DOLMA-Conservative* across SPEC 2017 are shown in Fig. 2.6, provided for both memory-only (M) as well as memory and register (M+R) protection variants. I display these numbers alongside corresponding STT variants, and depict 95% confidence intervals for the reported CPIs.

For protection against Spectre-type attacks, STT provides STT-*Spectre*. However, unlike *DOLMA-Default*, STT-*Spectre* does not mitigate Spectre-type attacks exploiting data speculation, such as speculative store bypass [226], nor various transmissions via stores. STT-*Spectre* (M) incurs 8.7% overhead, while STT-*Spectre* (M+R) incurs 30.8% overhead. Thus, despite offering greater protection, *DOLMA-Default* (M) (10.2%) yields comparable overhead to STT-*Spectre* (M) (8.7%), and *DOLMA-Default* (M+R) (22.6%) scales to registers significantly better than STT-*Spectre* (M+R) (30.8%).

To provide the additional protection against Meltdown-type attacks offered by *DOLMA-Conservative*, STT-*Futuristic* incurs 44.5% (M) and 63.4% (M+R) overhead, but fails to protect select store-based transmissions. In contrast, *DOLMA-Conservative* only incurs 29.7% (M) and

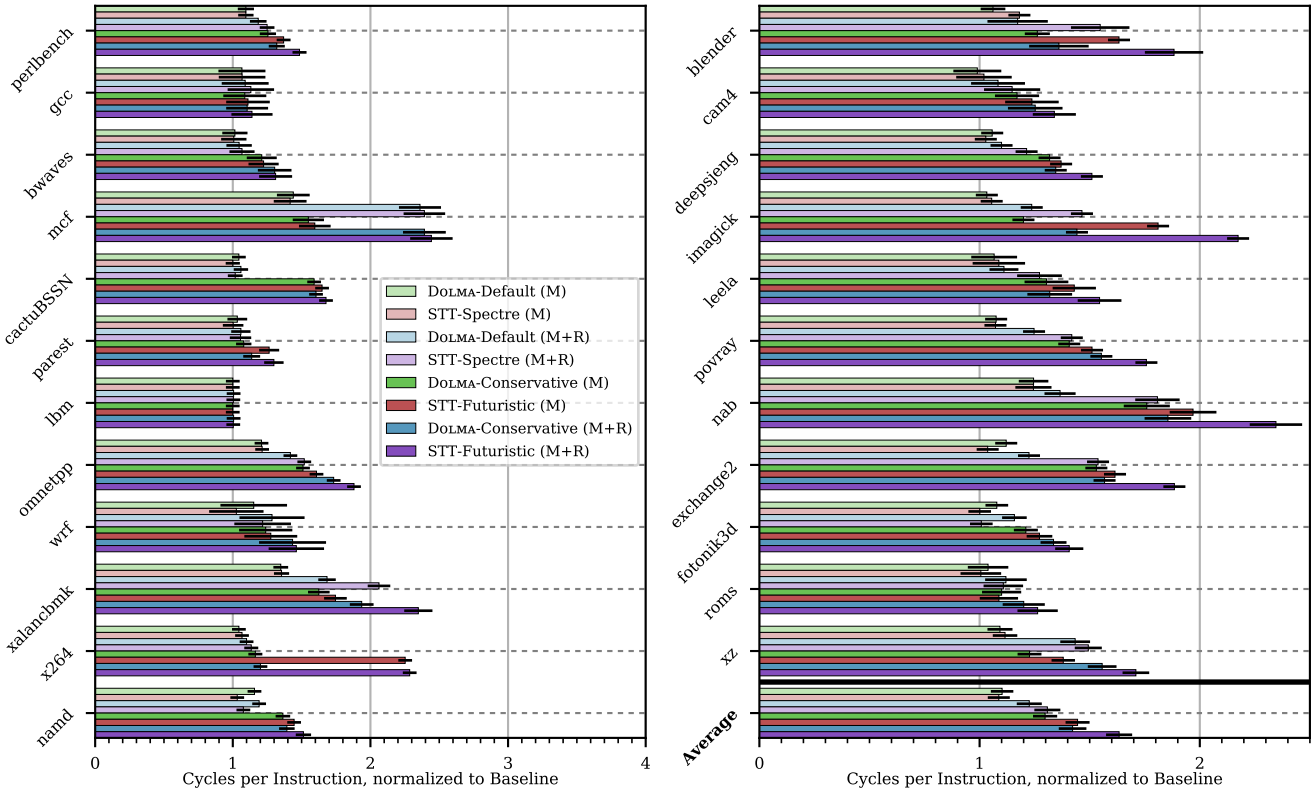

Figure 2.6: DOLMA's single thread performance on SPEC 2017, compared to STT [340]. Error bars depict the 95% confidence intervals.

42.2% (M+R) overhead to protect against all existing Meltdown-type and Spectre-type attacks on data in memory and registers, respectively.

DOLMA's ability to provide protection at lower overhead than STT primarily arises from the use of delay-on-miss for memory micro-ops. While STT insecurely allows all speculative stores to execute, STT conservatively delays all unsafe loads. In contrast, DOLMA only delays unsafe loads and stores when they miss in the TLB and—in the case of loads—the L1 cache.

With SMT (2 Threads). I compare the geometric mean of total CPI overhead across 2 threads between DOLMA and STT in Table 2.2, alongside single thread means. Reported CPIs are listed with 95% confidence intervals. For both DOLMA and STT, I find that the performance overhead of protection decreases with SMT enabled. This arises due to the fact that when a micro-op from some thread A is stalled for protection, some other thread B can potentially still make progress.

As with single-threaded configurations, I find that both *DOLMA-Default* and *DOLMA-Conservative*—unlike STT—prevent all existing transient execution attacks at mostly lower overheads. *DOLMA-Default* (M+R) (9.8%) again scales to registers far better than STT-*Spectre* (M+R) (17.3%), and *DOLMA-Conservative* likewise achieves lower overhead than STT-*Futuristic*—16.2% versus STT's 25.5% (M) and 22.4% versus STT's 36.8% (M+R). The only exception to

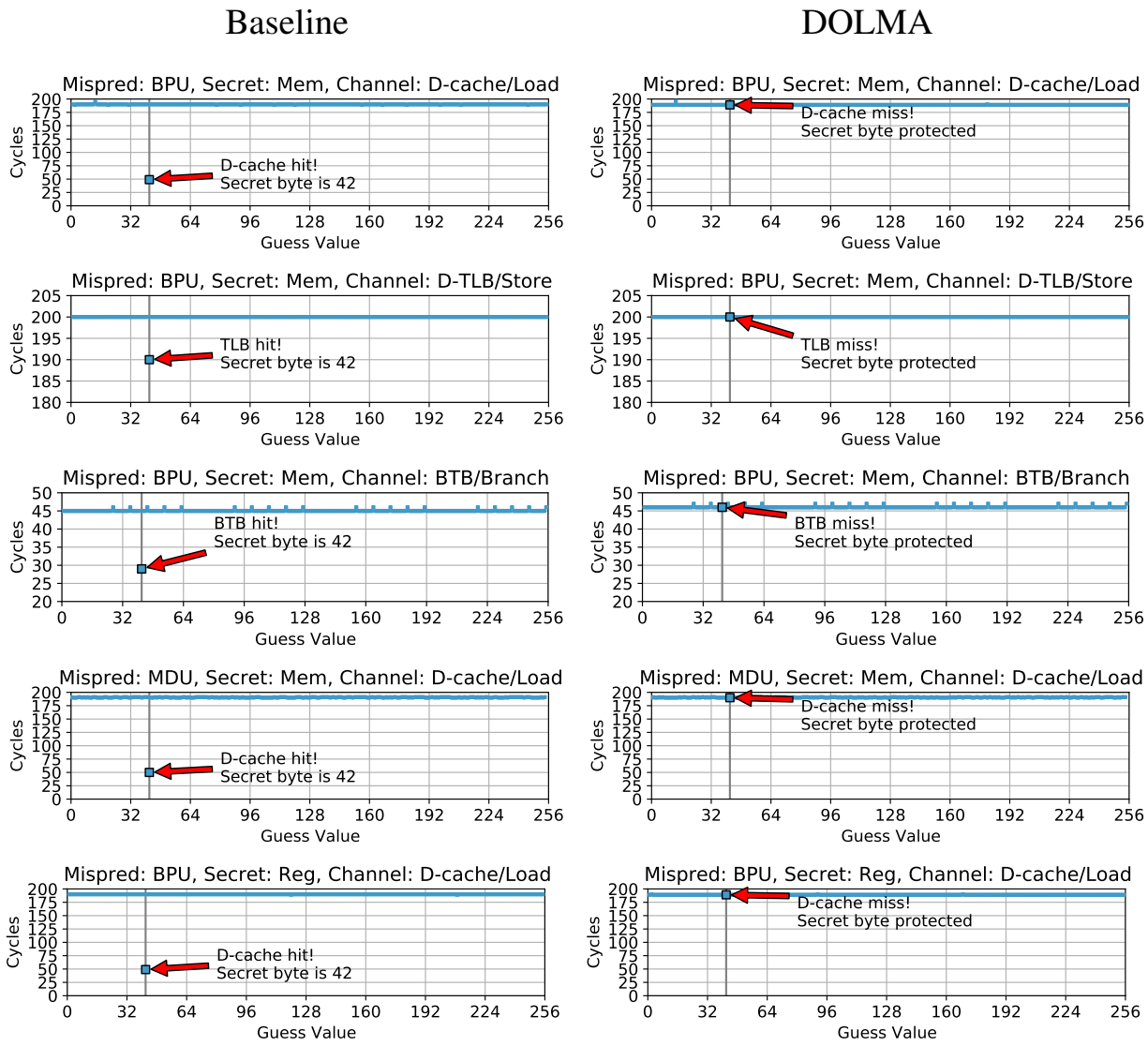

Figure 2.7: A demonstration of DOLMA's effectiveness in mitigating various covert timing channels. Each of the attacks leaks the value of the secret byte (42) on a baseline OoO processor (left) in gem5 [24]. In contrast, a DOLMA-protected processor prevents data from entering these channels during transient execution, thereby mitigating the attacks (right).

this trend is *DOLMA-Default* (M) (3.4%) versus STT-*Spectre* (M) (3.2%), where performance is still roughly equivalent despite DOLMA's additional protections for store-based transmission and data speculation.

2.7.2 Security Evaluation

I additionally compare the effectiveness of DOLMA against transient execution attacks with STT in Table 2.2. *DOLMA-Default* blocks all documented Spectre-type attacks, and *DOLMA-*

Mode	Energy	Energy (SMT)
DOLMA-*Default* (M)	10.8%	4.46%
DOLMA-*Conservative* (M)	29.2%	16.4%
DOLMA-*Default* (M+R)	22.4%	10.4%
DOLMA-*Conservative* (M+R)	40.9%	21.9%

Table 2.3: DOLMA's normalized total energy usage (processor and caches) compared to a baseline single thread and SMT processor, respectively.

Conservative blocks all documented Spectre-type and Meltdown-type attacks. STT-*Spectre* variants fail to address any Spectre-type attack that exploits data speculation [226]. Additionally, all STT variants fail to comprehensively address store-based transmission—including the speculative TLB modifications and speculative partial store buffer hits mentioned in this paper.

Penetration Testing. Although simulating every known transient execution attack is not possible in gem5, I have ported a diverse set of transient execution attacks into an open-source, gem5-compatible test suite [181]. The goal of this test suite is to directly demonstrate the ability of DOLMA—as well as future defenses—to mitigate transient execution attacks across a wide range of covert timing channels (e.g., backend channels such as the D-cache and D-TLB, as well as frontend channels such as the I-cache and BTB), unsafe micro-ops (e.g., memory micro-ops and branches), types of speculation (e.g., control and data), and locations of secrets (e.g., in-register and in-memory).

I depict DOLMA's effectiveness in mitigating a sampling of these transient execution attacks in Fig. 2.7. Namely, I show that DOLMA mitigates timing-based transmission of speculatively-loaded data through the D-cache (load micro-op), BTB (branch), and D-TLB (store). I additionally show that DOLMA's protection applies to speculative store bypass [226] as well as attacks on non-speculative register data. As pictured, DOLMA defeats all attempted transient execution attacks, regardless of the covert channel, unsafe micro-op, type of speculation, and location of the secret.

2.7.3 Area and Energy Estimates

I provide area and energy estimates for DOLMA using McPAT [178] with recommended changes for increased accuracy [324]. I model DOLMA's conceptual changes to the microarchitecture as follows. The unsafe queue is conservatively implemented as a second copy of the issue queue, extended with $1 + \log_2(\mathsf{sizeof}(\text{ROB}))$ bits per entry to hold the pending redirect bit as well as ROB index of the youngest unresolved inducer (set to 0 if all are resolved, meaning the micro-op may re-issue). Each functional unit (e.g., ALUs and FPUs) and entry in the load-store queue is extended with $\log_2(\mathsf{sizeof}(\text{ROB}))$ bits to indicate the corresponding micro-op's position in the ROB; such a design allows DOLMA to enforce its backend contention policy. Finally, similar to prior work [340], entries in the frontend register alias table are extended with $\log_2(\mathsf{sizeof}(\text{ROB}))$ bits to

indicate an operand's youngest unresolved inducer (either directly produced by a data dependency, or indirectly via a control dependency).

For area, I find that DOLMA's overhead is negligible when configured for either single threaded or SMT execution, incurring 0.9% overhead compared to respective baselines. For energy, as shown in Table 2.3, DOLMA's normalized total energy usage is dominated by its increase in execution time; energy usage overheads for both single thread and SMT configurations roughly correspond to performance overheads for the respective baselines. Therefore, in line with performance overheads, normalized energy usage for SMT configurations incurs lower overhead (normalized to an SMT baseline) than a single thread configuration (normalized to a single thread baseline).

2.7.4 Limitations

First, as the gem5 baseline processor does not allow faulty data propagation, I are unable to directly demonstrate the effectiveness of *DOLMA-Conservative* against Meltdown-type attacks. However, given that *DOLMA-Default* clearly prevents SSB [226]—and the restriction policy *DOLMA-Default* applies to SSB load dependants is extended to *all* load dependants in *DOLMA-Conservative*—I argue that *DOLMA-Conservative* indeed mitigates Meltdown-type attacks.

Second, I only demonstrate transmission via the BTB using the simpler of gem5's two BTB implementations (i.e., one that uses a less complex indexing function). However, as speculative BTB transmission has been demonstrated on real hardware [196], it is clearly a viable channel.

Third, the gem5 baseline only features constant-time ALU and FPU operations, meaning DOLMA's benefits over prior work [340] for these operations are not modelled.

Fourth, because gem5's system emulation mode does not add latency for TLB misses, our figures include an artificial TLB miss latency of 10 cycles for visualization purposes. I conservatively calculated this latency by assuming a 2-cycle penalty for the initial miss, plus a 4-cycle L1 lookup for each TLB stage. I verified that a TLB hit only occurs for the secret value in the simulator.

Fifth, modeling hardware in software simulators limits evaluation accuracy in the name of implementation feasibility. This limitation is particularly prevalent for total energy estimates, which depend on the accuracy of gem5 performance numbers and McPAT calculations.

2.8 Related Work

Transient execution attacks. Spectre [158] and Meltdown [179] are the first known attacks that exploit speculative execution to leak data via microarchitectural timing side channels. Since then, a wave of attacks have emerged. Most of these attacks use the D-cache as a timing side channel [35, 45, 156, 158, 164, 179, 191, 194, 226, 227, 229, 264, 266, 268, 283, 291, 296, 297, 301, 302, 303, 316,

326]. Attackers have also demonstrated speculative data leaks through the AVX unit [267], issue ports [22], I-cache [196], BTB [196], and global staging buffer [245], as well as suggested the possibility of speculative data leaks through the TLB [259, 331]. Recent work demonstrates that TSX Asynchronous Aborts can also be exploited to leak secrets [266, 301].

Software Mitigations. Due to the difficulty of patching deployed hardware, numerous software patches for transient execution attacks exist. Unfortunately, no software-only techniques provide comprehensive protection.

For Meltdown-type attacks, software mitigations tend to focus on enforcing stronger isolation between security domains. For example, kernel address space layout randomization (KASLR) increases the difficulty of finding kernel data to leak [90]. However, while KASLR makes Meltdown more difficult to exploit, it does not altogether prevent it. Kernel page table isolation (KPTI) defeats the original Meltdown variant by placing kernel data in a separate address space [56, 90]. However, KPTI does not prevent other Meltdown-type attacks [35,227,266,268,283,296,302,316]. Other proposed defenses offer attack- or channel-specific OS/VMM code modifications. For instance, flushing the cache on context switches between privilege levels only mitigates the cache channel [111, 316].

A wider variety of software mitigations have been proposed for Spectre-type attacks. Compiler techniques include modifying vulnerable code patterns to prevent a subset of transient execution. For example, Retpoline [293] protects call and return instructions from speculatively leaking values on the return stack buffer as in Spectre-RSB [164]. Unfortunately, Retpoline fails to protect against other Spectre variants.

Other compiler techniques insert LFENCEs or add artificial data dependencies to prevent transient loads [38, 231, 276, 287, 307], potentially using program analysis techniques or hardware-software contracts to identify information flows [93, 94, 307]. These techniques can mitigate attacks on memory-based secrets, such as Spectre v1 and Spectre v2 in some cases. However, they either fail to protect register-based secrets, fail to cover all Spectre variants (e.g., SSB [226]), or incur higher overhead than the state-of-the-art hardware defense [340].

Hardware Mitigations. Hardware defenses offer the ability to mitigate transient execution attacks at their microarchitectural sources [205, 315]. The comprehensive solution for all Meltdown-type attacks is to prohibit potentially faulty micro-ops from propagating their results in future processors [179, 315]. In the interim, microcode patches have been individually issued for Meltdown-type attacks [111, 113, 302, 316].

Hardware patches also exist for certain Spectre-type attacks. Processors can automatically insert LFENCEs after branches and context switches via microcode [112, 113], as well as disable SSB [7, 114]. SpecCFI [163] prevents Spectre v2 by restricting speculative jumps to an authorized set of targets. None of these techniques, nor their union, can mitigate all Spectre variants.

MI6 [30] provides secure enclaves in an out-of-order processor via microarchitectural resource isolation (e.g., flushing core-local structures on context switches). Compared to DOLMA, MI6 does not support SMT and requires the use of a software monitor executing non-speculatively to manage transitions between the enclave and outside world.

InvisiSpec [331] and others [2, 143, 177, 255, 258, 259, 331] only protect select load-based transmission channels (e.g., the D-cache), in contrast to speculative information flow control defenses such as DOLMA. The InvarSpec microarchitecture [349] optimizes these cache-centric defenses, using compiler-generated instruction annotations to help determine when a load's execution would not explicitly reveal speculative operand values.

Manual speculative information flow control defenses [76, 265, 338] require the programmer to annotate secrets for protection, as opposed to the automatic protection provided by DOLMA. The strict timing requirements for annotated data ensure that speculative execution produces neither transient *nor non-transient* side channel leakages (i.e., in the event speculation resolves correctly). While effective in providing protection for annotated data, the security of manual defenses relies on proper programmer annotation of secrets.

Existing automatic speculative information flow control defenses [17, 315, 340] prevent varying sets of speculative dependants from issuing until speculation resolves. Notably, SpecShield [17] only protects speculatively-accessed data (e.g., data in memory), and NDA [315] does not prevent leakage of register-based secrets via a single transient micro-op. STT [340] fails to comprehensively mitigate transmissions via stores, whether secrets are in memory or in registers. In contrast, DOLMA protects against all known transient execution attacks, and incurs 8.2–21.2% less overhead than the state of the art [340] when scaling to protect data in registers.

Finally, during the revision of this paper, the authors of STT published an optimization framework (SDO [339]). Like DOLMA, SDO improves performance over STT primarily by allowing speculative loads to safely execute in certain scenarios. SDO creates a "data-oblivious" load, which behaves independently of its operands as well as other unsafe operands. However, to achieve such a load, SDO requires that (1) for each operand-dependent resource access (e.g., a cache bank access), the load instead accesses all such resources (e.g., all banks), and (2) the load blocks all other accesses to these resources until complete. Accordingly, an attacker could intentionally issue speculative data-oblivious loads to temporarily deny cache access to other tenants, in a similar manner to prior cache denial-of-service attacks [18, 320]. Furthermore, SDO does not address any of the store-based security vulnerabilities present in STT and does not consider the effects of the staging buffer [245]. Therefore, SDO requires additional considerations for multitenant environments.

2.9 Conclusion

Efficiently mitigating transient execution attacks is challenging. Initial hardware mitigations focused on cache transmission [2, 143, 154, 177, 259, 331]. Manual speculative information flow control defenses [76, 265, 338]—though effective—require error-prone annotations of secrets. Automatic solutions fail to comprehensively protect data in registers [17, 315, 339, 340] or memory [339, 340]. DOLMA introduces a novel principle of transient non-observability, combining a lightweight speculative information flow control design with a set of secure performance optimizations to protect data in memory and registers against all existing transient execution attacks.

CHAPTER 3

Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations

3.1 Introduction

DRAM is the most prominent main memory technology, attractive due to its high density and low cost. DRAM *cells* are organized in row-column arrays, accessed by first *activating* a row (i.e., connecting it to a buffer) and then reading from or writing to this buffer. Since cells leak charge over time, rows are periodically *refreshed* to retain their data.

Unfortunately, as DRAM density increases with successive module generations (desirable for cost and efficiency reasons), so too does the electromagnetic interference among physically-proximate DRAM rows. Ultimately, this rising interference increases *DRAM disturbances* [152], wherein bit values in nearby rows of DRAM are flipped.

Concerningly, Rowhammer attacks [53, 59, 75, 91, 92, 124, 131, 147, 152, 167, 180, 218, 241, 250, 269, 288, 299, 300, 325] show that certain memory access patterns can increase the frequency of DRAM disturbances. In particular, frequently activating one or more *aggressor* rows—prior to the scheduled refreshes of nearby *victim* rows—may cause bit flips in the victim rows.

These hardware-level bit flips manifest as system-level problems, with particularly troubling ramifications in multi-tenant computing environments (e.g., the cloud). For instance, one tenant may corrupt the data of another, leading to data loss or machine shutdown/failure. In other scenarios, flips of security-critical bits (e.g., page table permission bits [269]) can compromise an entire host.

To date, DRAM vendors have shown years of unwillingness (perhaps due to economic reasons) to provide a comprehensive solution to Rowhammer. Despite vendor claims that their defenses prevent all Rowhammer attacks [172, 206], recent work [59, 75] demonstrates that Rowhammer exploits remain viable. Given the blackbox nature of vendor mitigations, system administrators are left largely powerless to prevent these attacks, just as they were the original exploits.

In fact, recent work [147] argues that optimal defenses should include *software* support. The authors show that proposed hardware mitigations [152,171,337] struggle to scale or cannot provide comprehensive protection given increasing DRAM density. Consistent with these findings, state-of-the-art follow-up defenses [236, 336] are limited by worsening performance overhead and a need for increasing SRAM or CAM area (i.e., relatively-expensive memory) as density increases.

However, existing software defenses [13, 27, 31, 41, 91, 123, 123, 161, 175, 300, 322, 347] cannot achieve compehensive and practical protection due to the lack of hardware-based Rowhammer management primitives. For instance, ANVIL [13] relies on information from performance counters that do not account for direct memory accesses (DMAs); this leaves the system vulnerable to DMA-based Rowhammer attacks [300, 317], a concerning threat surface for cloud providers.

The shortcomings of both hardware and software defenses highlight the need for a hardware-software co-design to mitigate Rowhammer. More specifically, current hardware defenses need software support to adapt and scale to emerging attacks, while software defenses need hardware assistance to effectively mitigate attacks.

Fortunately, despite years of incomplete and blackbox mitigations from DRAM vendors, *CPU* vendors can still provide hardware assistance for defenses. I argue that CPU vendors should add a new set of Rowhammer management primitives to the CPU's integrated memory controller. Compared to DRAM vendors, CPU vendors have shown a willingness to expose a relatively-high number of memory management features to programmers, including a variety of performance counters [110] and BIOS configuration parameters [115].

I motivate our proposed memory controller primitives with key insights about Rowhammer attacks and existing defenses, producing a novel taxonomy of mitigation approaches: *isolation-centric*, *frequency-centric*, and *refresh-centric*. I show that system admins (e.g., cloud providers) can use our primitives to produce scalable and adaptable software defenses according to this taxonomy. Finally, I conclude with a long-term outlook on how major consumers of DRAM can drive the changes in hardware-software co-design needed for a comprehensive solution to Rowhammer.

3.2 Background

In this section, I provide background on DRAM and a novel taxonomy of Rowhammer defenses to understand our new hardware primitives and software defenses.

3.2.1 DRAM+Rowhammer: A Crash Course

DRAM modules (e.g., SO-DIMMs in laptops and DIMMs in servers) consist of numerous *banks*, where each bank is a set of row-column *subarrays* of cells. A cell's charge distinguishes a partic-

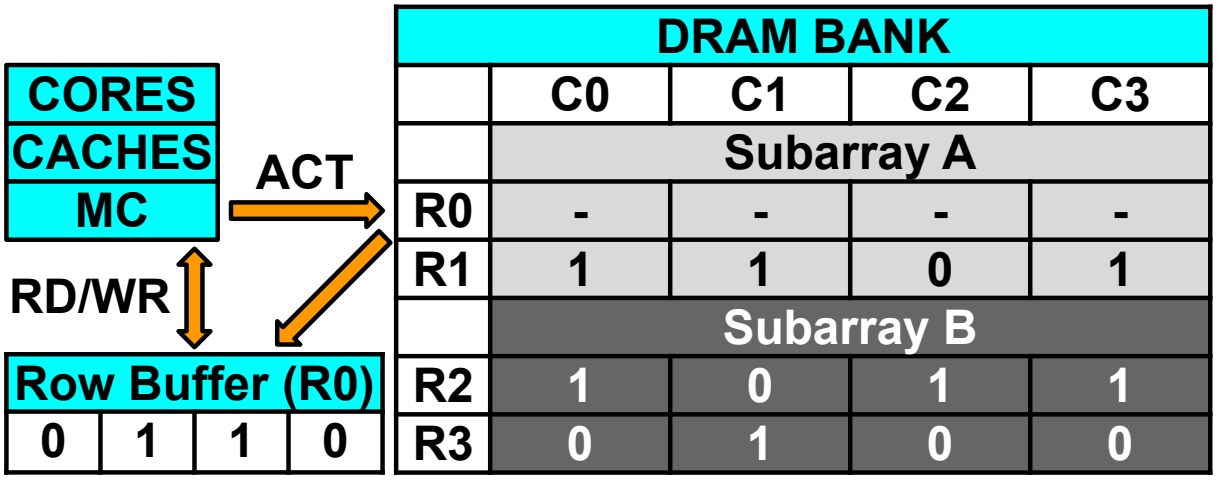

Figure 3.1: A simplified memory system. Memory controller MC activates row $R0$ in subarray A, connecting it to the bank's row buffer for read/write commands.

ular bit as either 0 or 1.

Modules are programmed via a memory controller. For instance, the memory controller converts requests targeting CPU physical addresses into commands targeting *DDR logical* addresses (e.g., bank, row, column) according to a fixed mapping determined by BIOS settings [51].

To actually read/write the cells within a particular row, the memory controller must first issue an *activate* (ACT) command to the row containing the cells, thereby connecting this row to its encompassing bank's *row buffer* for processing. Such an ACT is shown in Fig. 3.1, where a module is depicted as a single bank with two 2×4 subarrays for simplicity. I note that each bank has its own row buffer, and a bank may contain hundreds of subarrays that share its row buffer.

At this point, the memory controller can issue read (RD) or write (WR) commands to cache line-sized column offsets within the activated row, until the row is *precharged* (i.e., de-activated); precharge (PRE) commands are typically issued so that another row in the same bank may occupy the row buffer for RDs/WRs. In line with processor cache behavior, RDs/WRs that hit in the row buffer are faster than those that necessitate an ACT before the data access can proceed.

Because DRAM cells leak their charges over time, the memory controller periodically issues *refresh* (REF) commands such that each row's cells are recharged (i.e., repaired) before losing their bit values. Typically, each 8 KB row must be refreshed within 64 milliseconds of its last refresh, where the module cycles through its rows during this *refresh interval*. I note that an ACT of a row also repairs the row as a side effect; thus, an ACT can essentially be used for row refresh.

Unfortunately, Rowhammer attacks [152] show that frequent ACTs of the same row(s)—induced by certain memory access patterns—can corrupt data in *physically-proximate* rows. In particular, alternating RDs or WRs to a set of *aggressor* rows within a single bank necessitate alternating ACTs of these aggressors due to bank conflicts (i.e., row buffer contention). In turn, be-

cause of the electromagnetic interference among physically-proximate rows, these frequent ACTs may disturb the charges in nearby *victim* rows.

More precisely, each row can safely withstand a per-module *maximum activation count* (MAC) of ACTs within a refresh interval. However, if one or more aggressors surpass their MACs before a (potential) victim row is refreshed, the victim's data may be corrupted. Victim rows are those found up to b rows away from an aggressor, where b defines an aggressor's *blast radius* (which varies across DRAM technologies).

Notably, attackers with knowledge of DRAM address mappings can target *specific* data for corruption. While DRAM occasionally remaps two logically-adjacent rows to different internal locations [51], these remaps (and thus, internal adjacency) can be revealed via established methods. In particular, prior work [51, 75, 147, 161] uses the success or failure of Rowhammer attacks themselves—which require physically-proximate rows—to infer row adjacency.

3.2.2 Rowhammer Mitigations: A Taxonomy

At a high level, mounting a Rowhammer attack requires three conditions. First, at least one victim row must be located within the blast radius of at least one aggressor row. Second, one or more of the aggressor rows must be activated greater than MAC times within a refresh interval. Third, the victim row(s) must not themselves be refreshed before the aggressor(s) surpass the MAC.

Thus, Rowhammer defenses should eliminate one of these conditions, yielding our novel taxonomy of viable mitigations: *isolation-centric*, *frequency-centric*, and *refresh-centric*. I note that concurrent work [336] offers a similar taxonomy.

Isolation-Centric. Isolation-centric mitigations (e.g., [27, 31, 161, 322]) aim to physically isolate the rows from two different trust domains such that no cross-domain aggressor-victim relationships exist (e.g., a process cannot hammer another). For instance, ZebRAM [161] places b restricted-use "guard" rows between each potential aggressor-victim pair (where b is equal to the blast radius). Notably, isolation-centric mitigations typically do not prevent intra-domain DRAM disturbances (i.e., where an aggressor-victim relationship exists within a single domain, potentially inadvertently).

Frequency-Centric. Frequency-centric mitigations (e.g., [308, 336]) try to prevent the dangerously-frequent ACTs of aggressor rows needed to disturb nearby victim rows. For instance, BlockHammer [336] throttles (i.e., rate-limits) ACTs of aggressor rows according to a set of proposed memory controller counters, ensuring the number of ACTs to any row during a refresh interval stays below the MAC.

Refresh-Centric. Finally, refresh-centric mitigations (e.g., [13, 75, 152, 171, 279, 337, 347]) seek to refresh potential victim rows before they experience bit flips. More specifically, these

defenses use a set of hardware and—in some cases—software mechanisms to identify potential victim rows. The defense systems then proactively refresh these victims before the corresponding aggressor row(s) reach their MACs.

3.3 D(R)AMit, I Can't Do It by Myself!

Concerningly, recent work [147] demonstrates that the Rowhammer problem is worsening in successive DRAM generations. Specifically, as emerging DRAM technology nodes become denser, the electromagnetic interference among rows worsens, resulting in greater blast radii and orders-of-magnitude fewer ACTs (i.e., lower MACs) needed to induce charge leakages. Furthermore, lower MACs imply that a greater number of rows can act as aggressors (i.e., bypass the MACs).

Thus, while various hardware defenses have been proposed [75, 81, 84, 97, 144, 147, 149, 152, 171, 172, 206, 236, 279, 308, 310, 311, 336, 337], recent work has concluded that even the state of the art among them either (a) cannot provide comprehensive protection or (b) require significant overheads to scale to denser DRAM technology [147]. Ultimately, **DRAM experts have identified hardware-software cooperative mitigations as a key avenue for addressing the scalability challenges of Rowhammer defenses going forward [147]**.

Unfortunately, sufficient hardware support for Rowhammer defenses is unlikely to come from DRAM vendors in the immediate future. First, DRAM vendors continue to expose little information about their Rowhammer mitigations and potential limitations, possibly due to a desire to maintain trade secrets about their DRAM design.

Second, even *today's* DRAM modules (let alone tomorrow's) are still vulnerable to Rowhammer [59, 75], despite vendors originally claiming the opposite [172, 206]. Prior work [59, 75] has shown that in-DRAM blackbox defenses (Target Row Refresh, or TRR) mitigate attacks by tracking a small number n of aggressor rows (where n varies by module and vendor), but can be bypassed with $> n$ aggressors. Given increasing numbers of aggressors, this a bleak observation.

3.4 Changing the Game with New Primitives

In contrast to hardware defenses, software defenses tend to require less invasive—if any—changes to memory system hardware, with the added benefit that software implementations allow for adapting to yet-unknown exploit patterns. Unfortunately, as I will show, software defenses presently lack sufficient support from hardware to provide comprehensive and practical protection against Rowhammer attacks.

However, while a solution to the Rowhammer problem would ideally include changes to DRAM, I argue that the current limitations of software defenses can still be overcome via minor

changes to the CPU's integrated memory controller. Compared to DRAM vendors, CPU vendors have demonstrated a willingness to expose a plethora of information and configuration parameters, including various memory controller performance counters [110] and memory configuration settings in the BIOS [115].

I therefore discuss three key limitations in the context of implementing isolation-, frequency-, and refresh-centric software mitigations on existing CPUs. I then describe how to address each of these limitations with simple extensions to the memory controller (summarized in Table 3.1), thereby forming the primitives necessary to produce efficient and practical isolation-, frequency-, and refresh-centric defenses in software. I plan to precisely evaluate the benefits/drawbacks of these defenses in future work (e.g., using the gem5 [24, 188] microarchitectural simulator); the RISC-V [11] ecosystem offers a viable alternate evaluation platform.

Notably, I largely assume that the host OS (e.g., the hypervisor) is trusted to implement and enforce these mitigations. I provide considerations for enclave memory (e.g., Intel SGX [57]) at the end of this section.

3.4.1 Isolation-Centric: Interleave It To Me

Problem: Interleaving Mixes Trust Domains. Implementing isolation-centric defenses in software—wherein aggressor rows from one trust domain d_1 (e.g., a process) cannot disturb victim rows from another domain d_2—requires that the host OS's page-level memory allocator can ensure data from d_1 is outside the blast radius b of data from d_2 in DRAM. Such isolation has been achieved by placing b "guard" rows between trust domains [161], or using a bank-aware memory allocator [27, 31, 343] to place d_1 and d_2 on different banks.

However, prior work [27, 31, 161, 343] fails to sufficiently account for the complexity and performance benefits of *memory interleaving* on modern systems. Because a bank can only process one command at a time, the memory system interleaves (i.e., spreads) consecutive cache lines from the CPU's physical address space across the system's numerous banks [345]. Such interleaving achieves bank-level parallelism when accessing physically-consecutive cache lines (i.e., consecutive lines can be accessed simultaneously for efficiency).

Unfortunately, such interleaving also distributes lines from different pages (i.e., potentially different trust domains) into the same bank. While interleaving can be disabled in the BIOS (allowing the memory allocator to isolate pages from different domains to specific banks), this eliminates the performance benefits of bank-level parallelism [43, 153, 169, 286, 345] (i.e., parallelism measured to reduce execution time for certain applications by over 18% [286]) and is thus an undesirable solution for production environments.

Primitive: Subarray-Isolated Interleaving. I argue that there is a middle ground, where

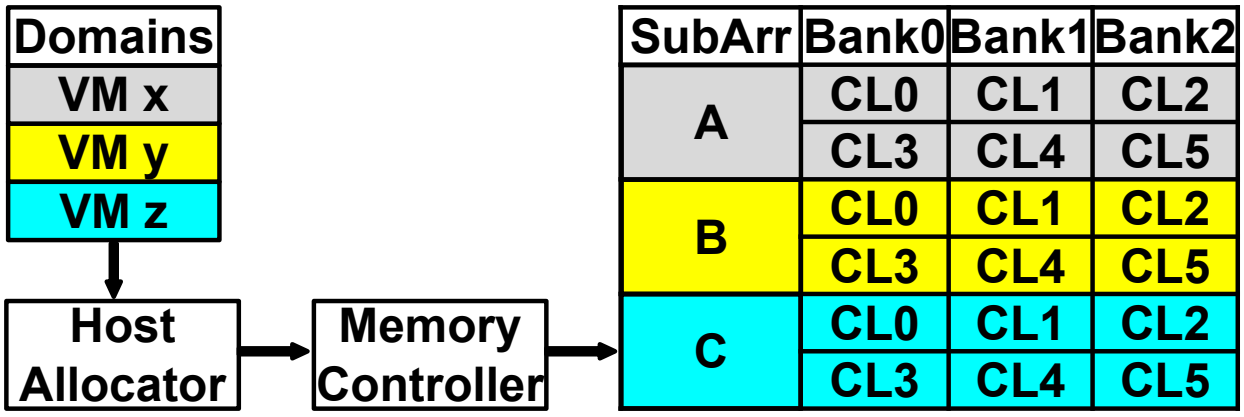

Figure 3.2: An example of subarray-isolated interleaving. The host memory allocator and memory controller cooperate to ensure that different trust domains (VMs x, y, and z) can reap the performance benefits of interleaving their consecutive cache lines CL0-CL5 across banks 0, 1 and 2. For security, the lines from each domain are restricted to per-domain, Rowhammer-isolated subarray(s): in this case, single subarray mappings of $x \rightarrow A$, $y \rightarrow B$, and $z \rightarrow C$.

existing interleaving (and its performance benefits) can remain fully-enabled, while pages from different trust domains can be isolated. Namely, rather than BIOS support for *bank*-aware memory allocation, I propose support for *subarray*-aware memory allocation.

Recall from Fig. 3.1 that a DRAM bank consists of a set of row-column subarrays. Notably, each subarray within a bank is electromagnetically-isolated from the others (i.e., they do not share bit lines) [43, 153], meaning data from different trust domains can be placed on different subarrays to prevent inter-domain aggressor-victim relationships. For instance, in a cloud environment, subarray isolation can be used to prevent inter-VM Rowhammer attacks.

Therefore, I posit that CPU vendors should provide a BIOS configuration option to enable *subarray-isolated interleaving* in each memory controller, as shown in Fig. 3.2. This option provides the host OS with two key features. First, cache lines from the same page will map to the same *subarray group* (i.e., set of specific subarrays across banks). Second, the host OS will be able to specify—either directly or indirectly—the trust domain of each page, such that the memory controller enforces that pages from the same trust domain map to the same subarray group.

From a software convenience standpoint, a direct specification would allow the host OS and memory controller to coordinate trust domains via an address space ID (ASID) tag per domain, akin to those already used in the TLB. However, if CPU vendors are unwilling to add hardware to track the mappings for a set of ASIDs, the memory controller already provides indirect support via its known set of CPU physical to DDR logical address mappings [51]; knowledge of these mappings has been used to allocate specific physical pages on specific banks [27, 31, 161, 343] and can similarly be used to map specific physical pages to specific subarray groups.

In an ideal world, DRAM vendors would also facilitate subarray-isolated interleaving by exposing DRAM-internal subarray mappings in the DDR logical address space, akin to what is already done with bank mappings. However, even without this information, internal subarray mappings can be inferred via the same methods used to infer internal row adjacency/remappings (§3.2.1); that is, the success or failure of Rowhammer attacks within a bank can be used to infer subarray boundaries from software.

Notably, DRAM could still remap a row from its logical subarray to a different internal subarray, posing a threat to subarray isolation. Thankfully, the host OS can use the inferred DRAM subarray mappings—coupled with knowledge of CPU to DDR logical address mappings—to ensure that all rows are allocated with their *internal* subarrays.

3.4.2 Frequency-Centric: Context Welcome

Problem: ACT Interrupts Lack Context. Implementing frequency-centric defenses in software necessitates the ability to identify potential aggressor rows (i.e., rows experiencing frequent ACTs). While modern Intel memory controllers can count ACTs per channel [109]—as well as interrupt system software after a host OS-configurable number of ACTs—they do not provide any information about the specific row being activated, nor the specific RD/WR command causing the ACT. Thus, system software is powerless to determine which address(es) to take action on in response to an ACT interrupt.

Primitive: Precise ACT Interrupt Events. To support identifying potential aggressors, CPU vendors should augment the existing ACT_COUNT overflow event [109] to report the physical address causing the latest ACT. Doing so would allow the host OS to probabilistically identify and react to potential aggressor rows/hot cache lines within. I note that this address information would be consistent with that already reported by various cache events on Intel [110].

More precisely, recall from §3.2.1 that ACT commands are issued when the cache line needed for a RD/WR is not in a row buffer. Thus, I propose that upon overflow, the ACT_COUNT interrupt event should report the physical (cache line) address of the most recent RD/WR to have triggered an ACT of the row. The host OS can then reset the counter to an arbitrary value, probabilistically identifying aggressors according to specific MACs. By also including a degree of randomness in counter reset values, the host OS can prevent attackers from avoiding detection.

The host OS can use the address information provided by the interrupt to limit near-future ACTs of the encompassing row (i.e., within the refresh interval) in a variety of ways. For instance, software could implement a form of ACT wear-leveling by remapping and moving the row's data to a new physical location, either in DRAM or another storage device. To improve the efficiency of this data transfer, the CPU could support an uncore (i.e., off-core but on the CPU chip) move

instruction using buffers in the memory controller, thereby avoiding the need to transfer data to and from on-core registers to relocate a line.

Alternatively, with the addition of cache line locking support (i.e., instructions or other mechanisms that temporarily pin a line to the processor cache, already available on many ARM processors [10, 87]), system software could use cache line locking for the duration of a refresh interval as a first line of defense. In particular, one or more ways in the LLC could be used for locked lines; data remapping and movement would then only be used as a fallback if the way(s) become full. Ultimately, such locking could improve access time for the line in question, prevent its continued use in ACT generation, and avoid a potentially costly data transfer.

3.4.3 Refresh-Centric: A Refreshing Take

Problem: SW Can't Directly Refresh Rows. Refresh-cen-tric defenses must proactively refresh potential victim rows (i.e., rows that are near aggressor rows). Given methods for determining aggressors (§3.4.2) and row adjacency (§3.2.1), I focus on how to refresh potential victims.

Unfortunately, software has at best inefficient and potentially unreliable mechanisms to refresh rows. In particular, the REF command does not include a row address argument, meaning the memory controller/software cannot use it to refresh specific rows; instead, the ACT command (which takes a row address argument) must be used (§3.2.1). However, software still lacks the ability to directly issue ACTs to DRAM, with this decision left up to the memory controller. Thus, software defenses cannot directly refresh rows.

In fact, software can only potentially refresh a specific row via a series of convoluted memory instructions (e.g., loads/stores). To reach the memory controller, the load/store must first miss in the cache, generally requiring software cache manipulation (e.g., a preceding flush instruction, if available) and strict ordering (e.g., memory fences) to reliably occur. At this point, the memory controller then converts the load/store into a set of RD/WR, ACT, and PRE commands; the specific set of commands issued depends on the state of the row buffers—information which is not directly exposed to software. Ultimately, this indirection introduces inefficiency and imprecision to defenses, especially in the presence of noise (e.g., memory operations from other cores/devices).

Primitive: A Refresh Instruction. To address software's lack of control over which rows are refreshed, I propose that the CPU should expose an instruction REFRESH whose effect is to refresh a specific row of DRAM. I liken this to Intel's patented mechanism for targeted refreshes via the memory controller [16]. However, in our case, the REFRESH instruction would be exposed in the ISA and—crucially—not require additional support from DRAM. Because the ACT side effect of the REFRESH instruction could be abused to carry out a Rowhammer attack, REFRESH should be a host-privileged instruction (i.e., only executable by the host OS).

Class	MC Primitive	Corresponding Software Defense(s)	Optional DRAM Assistance
Isolation	Subarray-isolated interleaving	Subarray-aware memory allocation	Internal subarray mappings
Frequency	Precise ACT interrupt event	Aggressor remapping, cache line locking	-
Refresh	CPU REFRESH instruction	Efficient software refresh of victim rows	REF_NEIGHBORS command

Table 3.1: Summary of proposed memory controller (MC) primitives, corresponding software defense(s), and optional assistance from DRAM by mitigation class.

The REFRESH instruction will take as an argument a virtual address VA—which maps to a particular DRAM row—and a bit AP indicating whether to auto-precharge the row after activation to prevent bank conflicts. The instruction will be implemented as follows. First, the TLB will translate VA to its corresponding physical address, which the memory controller will convert to a row address. Second, the memory controller will issue a PRE command to the row's encompassing bank to clear its row buffer. Third, the memory controller will issue an ACT command to the desired row, thereby effectively performing the refresh. Fourth, if the AP bit is set, the memory controller will issue another PRE command to the bank to clear the row buffer for subsequent accesses.

Finally, in an ideal world with support from DRAM, the DDR standard would include a REF_NEIGHBORS command, similar to that proposed in prior work [171, 236]. However, in addition to taking an aggressor row address as an argument, I propose that the command should also accept a blast radius b for adaptability to emerging threats. DRAM would then automatically refresh the potential victims of the provided aggressor row up to b rows away.

3.4.4 What About Enclave Memory?

The preceding discussion assumes that the host OS is trusted to implement and enforce the described mitigations. Notably, in certain enclave execution contexts (e.g., Intel SGX [57], Intel TDX [117], and AMD SEV [8]), only the enclave itself and hardware are trusted. Thus, these scenarios require additional considerations for software Rowhammer defenses.

Strictly-speaking, if enclave memory is checked for integrity upon access, then Rowhammer attacks can only cause system-wide denial-of-service (as opposed to arbitrary behavior changes stemming from data corruption). More specifically, upon a failed integrity check, the system will lock up and require a reset [124]. Because the host OS is untrusted and can already tamper with the integrity of enclave pages (i.e., without Rowhammer), such denial-of-service attacks are typically not considered in enclave threat models.

However, if enclave memory is *not* integrity-checked upon access, then the system must prevent (or at least, detect and gracefully shutdown upon) bit flips to ensure security. For isolation-centric defenses, the CPU could report the subarray(s) upon which the enclave resides in terms of physical

addresses, such that the enclave may simply verify its virtual to physical address mappings (as is already done [57]). For frequency-centric defenses, the CPU could report ACT interrupts directly to enclaves, such that they might infer they are under attack and either (a) require a remap to a new location or (b) peacefully exit where permissible. Finally, for refresh-centric defenses, I posit that in the presence of subarray-isolated memory, an enclave could be permitted to issue REFRESH instructions to addresses mapped within its address space. I leave the exploration of these solutions to future work.

3.5 Outlook: Optimal Fixes

In this paper, I have described the limitations of and disconnect between existing hardware and software Rowhammer mitigations. To address these issues, I have proposed a variety of CPU-based primitives that would enable effective and practical hardware-software co-design defenses. Nonetheless, while a combination of CPU and software mitigations may prove more immediately-viable than in-DRAM support (e.g., in terms of scalability, adaptability, and economics), the root of the Rowhammer problem lies in DRAM.

Thus, in the long-term, I argue that optimal implementations of our defenses would include collaboration with both CPU *and* DRAM vendors. To achieve cooperation from DRAM vendors, cloud providers, CPU manufacturers, and other major consumers of DRAM must convince DRAM vendors to expose the details—and limitations—of their mitigations. Doing so would allow software, CPU, and in-DRAM mitigations to work in tandem to efficiently and scalably solve the Rowhammer problem once and for all.

CHAPTER 4

MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads

4.1 Introduction

The threat of Rowhammer [152] bit flips (i.e., DRAM disturbances) is a widespread concern, especially in multi-tenant computing environments such as the cloud. Rowhammer arises from frequent activations—to a first approximation, accesses—of the same DRAM rows, which can disturb data in nearby rows due to electromagnetic interference. These bit flips manifest at the system level as data loss, machine failure, or system subversion.

Prior attacks and analyses [51, 53, 59, 75, 91, 92, 98, 124, 125, 131, 147, 152, 167, 180, 218, 233, 240, 241, 250, 269, 288, 289, 299, 300, 325] confirm that malicious adversaries can trigger sufficient activations to flip bits, establishing Rowhammer as a *security* threat. At a high level, existing attacks require a carefully-crafted sequence of instructions to bypass CPU caches and thereby frequently access DRAM. Thankfully, to our knowledge, these instruction sequences have *not* been shown to occur in non-malicious (e.g., commodity) workloads with sufficient frequency to risk bit flips.

However, I present *coherence-induced hammering*, a novel form of Rowhammer that *naturally occurs in commodity benchmarks* on cache coherent non-uniform memory access (ccNUMA) architectures (e.g., multi-socket servers used by a major cloud provider). Notably, coherence-induced hammering instruction sequences occur *without* workload manipulation. Thus, I offer the first evidence of Rowhammer's additional *reliability* threat.

Using DDR4 DRAM access traces, I show that Intel's ccNUMA coherence protocols frequently access DRAM in common data sharing scenarios. In fact, the protocols activate individual rows at rates *previously-shown to induce bit flips*. Amidst rising Rowhammer susceptibility in newer DRAM (i.e., fewer activations needed to flip bits, more rows simultaneously reaching these activation rates, and projections that the problem will continue to worsen [147, 233, 261]), it is paramount

to revisit ccNUMA protocol design before already-vulnerable mitigations [59, 75, 98, 125, 240] are overwhelmed.

I discern that coherence-induced hammering in commodity workloads arises from three phenomena, depending on the protocol. The most basic phenomenon is that of downgrade writebacks [220], a side effect of MESI protocols, where caches must write-back dirty lines before sharing them. In ccNUMA systems—where data can be shared among caches on different nodes—these downgrade writebacks can repeatedly go to DRAM, resulting in hammering. Luckily, downgrade writebacks can be trivially eliminated by adopting widely-used MOESI protocols [220].

Unfortunately, I discover two additional sources of coherence-induced hammering in Intel's MESI-based ccNUMA protocols that are more difficult to address. First, both Intel's broadcast and memory directory protocols issue speculative reads to DRAM as performance optimizations. However, certain data sharing patterns (e.g., migratory [58, 284], as occurs for lock-protected "writer-writer" data) induce repeated, mis-speculated DRAM reads of the same cache lines, triggering row activations that hammer DRAM. Second, Intel's newer memory directory protocol [202] adds another source of hammering. Specifically, inadvertently-redundant writes to the in-DRAM directory—to ensure coherence correctness—frequently-activate the same row(s) of DRAM. As I will show, these phenomena *cannot be prevented by a conventional MOESI protocol.*

Accordingly, in this work, I introduce MOESI-prime: a ccNUMA protocol that mitigates coherence-induced hammering, while retaining the use of Intel's state-of-the-art memory directory for scalability. MOESI-prime is based on the observation that mis-speculated reads and redundant directory writes (the remaining sources of coherence-induced hammering in a conventional MOESI protocol) can be omitted *without loss of correctness*. For instance, a speculative read can be omitted without loss of correctness if it will go unused due to mis-speculation. Likewise, a memory directory write can be omitted without loss of correctness if it is known to be redundant.

I show that adding just two additional stable states (i.e., the states a cache line can be in when a transaction is not already in progress) to a baseline 5-state MOESI memory directory protocol prevents hammering memory directory writes. Our key insight is that coherence-induced hammering only arises in the presence of dirty data. Thus, for "conventional" dirty states (**M** and **O**), I additionally provide "prime" variants (**M'** and **O'**). The prime states behave almost identically to their conventional counterparts to reduce the burden of ensuring protocol correctness (§4.5). The lone difference is that the prime states allow caching agents to recognize scenarios in which memory directory writes are guaranteed to be redundant, enabling safe omission of these writes. Notably, MOESI-prime's 7 stable states fit in 3 bits per cache line, consuming the same area as the 5 stable states of MOESI.

For hammering via mis-speculated reads, a simple change to the existing directory cache's management policy prevents offending reads—and only these reads—from being issued.

I evaluate MOESI-prime in gem5 [24, 188], using a full-system configuration that models a major cloud provider's production hardware. I demonstrate that MOESI-prime prevents identified sources of coherence-induced hammering in both malicious and non-malicious workloads. Additionally, I prove that baseline MESI/MOESI protocols can be transformed into MOESI-prime protocols without loss of correctness. Finally, I show that MOESI-prime's prevention of unnecessary reads and writes has negligible effect on average performance (within $\pm 0.61\%$ of MESI and MOESI baselines) and average DRAM power (0.03%–0.22% improvement) across PARSEC 3.0 [344] and SPLASH-2x [321] in 2-, 4-, and 8-node ccNUMA configurations.

In summary, I make the following contributions:

• Using DDR4 memory access traces—collected from commodity benchmarks on a major cloud provider's production hardware—I discover *coherence-induced hammering*, the first hammering found to occur in non-malicious code.

• I identify hammering sources in Intel ccNUMA protocols.

• I design MOESI-prime, a ccNUMA protocol that prevents coherence-induced hammering, while retaining the use of Intel's state-of-the-art memory directory for scalability.

• I show that MOESI-prime is the first mitigation that simultaneously prevents coherence-induced hammering, improves average DRAM power, and negligibly affects average performance—even slightly *increasing* performance for many workloads.

Our implementation and evaluation infrastructure is open-source [186].

4.2 Background

4.2.1 DRAM and Rowhammer

DRAM cells encode a single bit of information via high/low voltage, and are organized in row-column *banks* (arrays). To access cells within a row, a memory controller first issues an *activate* (ACT) command, connecting the row to its bank's *row buffer*. To read or write at cache line-sized granularity, the controller then issues read (RD) or write (WR) commands to column offsets within the buffer.

Rowhammer [152] is a circuit-level disturbance effect where frequent ACTs of the same row(s) can flip bits in *nearby* rows. For example, as only one row can occupy its bank's row buffer at a time, alternating RDs (or WRs) to *aggressor* rows within a bank require repeated ACTs of each aggressor. However, because of electromagnetic interference, nearby *victim* rows are susceptible to bit flips until they are periodically *refreshed*.

To combat Rowhammer, modern servers rely on error correction (ECC [53]) and target row refresh (TRR [75], a DRAM-internal defense that detects and refreshes select vulnerable rows

ahead of schedule). Unfortunately, these mitigations are not comprehensive, as uncorrected bit flips can be induced despite ECC [53] and TRR [59, 75, 125, 240]. Alternative mitigations yield a range of security-performance trade-offs and are not known to be deployed [13, 16, 21, 27, 31, 41, 73, 81, 84, 147, 149, 150, 152, 161, 171, 175, 183, 199, 236, 256, 300, 308, 322, 336, 337].

4.2.2 ccNUMA Architectures

Cloud providers deploy large quantities of cores and DRAM per server for cost effectiveness and ease of management. Accordingly, modern servers are often architected as non-uniform memory access (NUMA) for performance and scalability. A set of cores (e.g., a socket, cluster-on-die [105], or core complex/chiplet [252]) comprises a processing node, which is associated with a local (near) memory pool that is faster to access than remote (far) memory. Each physical address in the system maps to a local "home" node. Thus, NUMA can provide lower latencies to workloads using local memory, and reduce memory traffic interference among independent tasks on different nodes.

Today's NUMA servers are typically cache coherent (ccNUMA)—i.e., hardware enforces coherence across nodes. Specifically, each line maps to one *home agent* (located at the line's home node), which enforces the line's coherence.

Thus, ccNUMA systems offer a programmer-friendly coherent memory model across nodes, and scheduling flexibility via more cores and memory on one machine. While scheduling workloads across nodes can hurt performance [26, 249], cloud providers and customers benefit from the ability to (1) execute and easily manage workloads needing more resources than there are on a node, and (2) run smaller workloads in "pigeonhole" scheduling cases (e.g., sufficient cores and memory are only available if split across nodes).

4.2.3 Coherence Protocols

Commodity coherence protocols enforce a *single-writer, multiple-reader* invariant, where for a valid cached line, either (a) one core has exclusive write permission, or (b) one or more cores have read-only access. The invariant is typically enforced by *write-invalidate* on servers, where a core invalidates all other line copies before writing (to obtain exclusive access).

Coherence States. Coherence protocols are described in terms of their *stable states*, the states a line may be in when a transaction on the line is *not* in progress. During transactions, lines are in *transient* (i.e., busy) states. Stable states typically encode line validity, read/write permission, and dirty status (i.e., whether a line must be written back). For instance, a basic MSI protocol offers 3 stable states: **M**odified (dirty+writable), **S**hared (clean+read-only), and **I**nvalid (invalid).

A MESI protocol—variants of which are used by modern Intel servers [100]—adds the **E**xclusive state as an optimization, where **E** encodes clean+writable. The extra **E** state avoids

the need to obtain write permission after fetching private data (i.e., data only cached on a single core), reducing coherence traffic.

A MOESI protocol—used by modern AMD servers [54]—also adds the **O**wned state, where **O** encodes dirty+read-only. The potential benefit of using MOESI over MESI is the elimination of downgrade writeback traffic [220], incurred in MESI when a line in **M** is shared for reading with another cache. While the performance and energy difference between MOESI and MESI can be negligible [195], I discuss how MOESI's elimination of downgrade writebacks is critical in preventing a source of coherence-induced hammering in §4.3.2.

Directory/Broadcast. In addition to their stable states, coherence protocols can be classified as directory or broadcast (i.e., directory-less). Upon a private cache miss in a directory protocol, the requesting core looks up a cache line in a shared directory to determine the line's location and coherence state; the directory sends "directed snoops" to fetch a line from the appropriate cache as necessary to maintain coherence.

In broadcast protocols, no directory exists, and the requesting core instead broadcasts its request upon a miss to all other caches to check for the line (i.e., "broadcasted snoops"). While broadcast protocols yield simpler hardware, directory protocols scale better due to reduced coherence traffic (i.e., snoops are often directed to one cache, as opposed to broadcasted).

ccNUMA Considerations. The primary difference between ccNUMA and single-node protocols is that ccNUMA maintains coherence across multiple nodes. Upon an LLC miss, a *broadcast* ccNUMA protocol must send snoops to all other nodes, in case the line is dirty. A *directory* ccNUMA protocol can instead consult a multi-node directory, whose state determines whether snoop(s) must be issued (e.g., if the line is dirty). Given the premium placed on inter-node (e.g., QPI/UPI [100,213]) bandwidth, both Intel and AMD have opted to reduce snoop traffic by defaulting to directory ccNUMA protocols since at least 2017 [54, 121, 165, 202].

In a directory ccNUMA protocol, home agents can track the *local* state of their lines via a single-node directory that Intel calls the snoop filter [203, 334]. However, the agents need additional mechanisms to track the *remote* state of lines. Thus, in the Intel hardware investigated in this work, a ccNUMA directory is provided in DRAM ("below" individual nodes), akin to how the snoop filter is located "below" private caches.

As shown in Fig. 4.1, Intel repurposes 2/64 bits available in DDR4 DRAM for each line's ECC as *memory directory* bits, such that the bits are retrieved for "free" when the line is fetched. The bits can encode three coherence states [208]: *snoop-**A**ll* means the line is potentially dirty on a remote node, requiring a snoop for both read and write requests; *remote-**S**hared* means the line is potentially present (but clean) on remote node(s), only requiring the copies to be invalidated upon write requests; *remote-**I**nvalid* means the line is not remotely-cached.

A line's memory directory state may become stale (e.g., an **A** line is not *guaranteed* to be

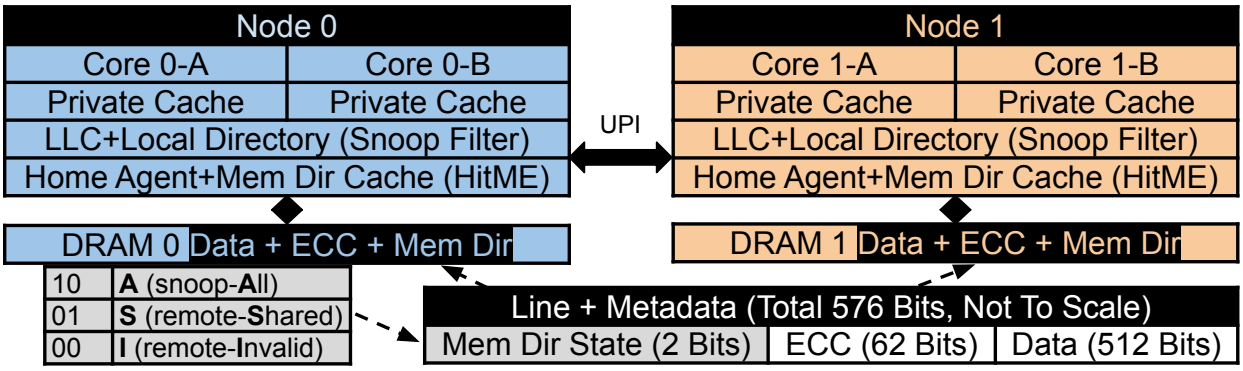

Figure 4.1: A simplified Intel Skylake ccNUMA system. Each line maps to a home agent that maintains coherence across nodes using distributed state. Local state is stored in the LLC+local directory (snoop filter). Remote state is stored in a line's *memory directory* bits. Select remote state is also stored in an on-die directory cache (HitME [208]) to reduce snoop latency.

dirty—or even present—on a remote node) provided that coherence is maintained. For instance, a stale **A** entry preserves correctness (albeit conservatively), simply incurring unnecessary snoops before ultimately servicing the line from the local node or DRAM.

While snoops can be omitted for lines in **S** and **I**, lines in **A** *require* snoops, which incur high latency if a memory directory read is needed. This latency is problematic when inter-node sharing frequently incurs this penalty (e.g., migratory sharing, "repeated writer-writer", of lock-protected data [58, 284]).

To avoid this repeated penalty, each home agent uses an on-die memory directory cache [208, 211] (henceforth referred to simply as a directory cache) for a subset of its lines in **A**. A directory cache hit implies the line must be snooped, obviating the need to read directory state from DRAM. Directory cache entries contain a bit for each node, indicating *which* node must be snooped, and are allocated upon cache-to-cache transfers to a remote writer. Thus, only entries for migratory (i.e., snoop-critical) lines occupy limited on-die area.

4.3 Coherence-Induced Hammering

In this section, I describe how I discovered sources of *coherence-induced hammering* in Intel's cc-NUMA protocols. These phenomena are the first examples known to cause *commodity* workloads to exhibit dangerously-high row ACT rates. I consider a row's ACT rate to be dangerous when it surpasses its *maximum activate count* (MAC)—the industry-standard metric for Rowhammer susceptibility. The MAC is the maximum number of ACTs to a set of aggressor rows within a *refresh window* (64 ms in DDR4 [126]) before any bits may flip in victim row(s). Recent work [52, 147, 261] shows that MACs are falling in newer DRAM, with current MACs as low

as 20,000. The studies use an alternative metric called HammerCount (HC); these HCs corresponds to half the MAC, given that they are calculated using two aggressor rows to target each victim.

4.3.1 Introduction and Methodology

I initially observed hammering in commodity workloads while conducting a study of DDR4 memory access patterns in internal cloud workload benchmarks, used by a major cloud provider. Prior to this study, only intentional Rowhammer attacks had been shown to surpass a DRAM module's MAC, meaning only carefully-crafted (malicious) code was known to risk bit flips.

Our experimental hardware consists of (1) a dual-socket (i.e., ccNUMA) Intel Skylake server configuration deployed by the cloud provider (2400 MHz, DDR4, 2Rx4 DIMMs with Chipkill [60] ECC), and (2) a DDR4 bus analyzer. The bus analyzer records timestamped-traces of DDR4 commands (e.g., ACT, RD, WR) and destination DDR4 logical addresses (e.g., bank, row, column) sent from a memory controller to a DIMM. The analyzer records up to 512 million commands, meaning different programs can be recorded for different amounts of time due to varying amounts of DRAM traffic.

I run Ubuntu Linux 20.04 with KVM [157] as our host OS, conducting experiments outside of production to protect customer privacy. Commodity benchmarks are executed in guest VMs, also running Ubuntu 20.04. Unless otherwise noted, all BIOS settings are the cloud provider's defaults.

For brevity, I provide evidence of hammering in two different cloud workloads (*memcached* [133] and *terasort* [230]), based on internal benchmarks used by the cloud provider. I show that PARSEC 3.0 [344] and SPLASH-2x [321] benchmarks exhibit similar behavior in §4.6.

For *memcached* and *terasort*, I record at least 10 seconds of execution per trace, given bus analyzer storage limits. I calculate the maximum number of ACTs to a single row within any 64 ms refresh window across all traces, and compare this number to modern MACs to assess Rowhammer risk. I measure ACT rates because they provide a relatively-stable metric to reason about Rowhammer across a fleet of servers. In contrast, different DIMMs' susceptibilities to bit flips vary by DRAM vendor, generation, process node variation, TRR implementation, and other factors.

To our surprise, both cloud workloads experience over 20,000 ACTs to a single row within 64 ms (nearly 40,000 for *terasort*), *surpassing modern MACs* and therefore risking bit flips. Furthermore, these ACT rates are almost certainly *under-estimates*, given that I can only record traffic to 1 DIMM out of the many DIMMs used by a workload in a production machine.

	MOESI			MESI		
Event	**C0**	**C1**	**Shared Memory Copy**	**C0**	**C1**	**Shared Memory Copy**
C0 writes	M	I	Stale (C0 has dirty copy)	M	I	Stale (C0 has dirty copy)
C1 reads	**O**	S	Stale (**C0 has dirty copy**)	**S**	S	Up-to-Date (**written back**)

Figure 4.2: Dirty sharing in MOESI (left) versus a downgrade writeback in MESI (right). MESI's lack of **O** (dirty+read-only) means dirty lines must be written back (cleaned) to be shared.

4.3.2 Source #1: Downgrade Writebacks

To determine the root cause(s) of hammering in commodity workloads, I conducted further analysis of the DDR4 access traces. I noticed that in the maximally-activated (hottest) rows, frequently-accessed cache lines often experienced more DRAM writes than reads. This observation was puzzling, as conventional wisdom indicates commodity workloads should almost always yield more DRAM reads than writes.

More specifically, (1) read-only (always clean) data traditionally only requires reads (i.e., no subsequent writebacks), and (2) writes to a word-sized segment (e.g., 8 bytes) in a 64-byte line require a preceding line read to preserve the non-modified portion of the line (unless it is known the entire line will be modified). If a clean line only produces a read, and a dirty line generally produces a read and a write, one would expect to observe more reads than writes.

However, there is a confounding factor in ccNUMA systems: DRAM, not the LLC as in single-node systems, is the point of coherence (i.e., the first level of the memory hierarchy shared among all cores). Thus, coherence traffic that traditionally goes to the LLC in a single node system may now go to DRAM, altering the "conventional" read-write ratio.

One known source of coherence writes are downgrade writebacks [220], incurred in MESI-based protocols (i.e., Intel server protocols [85, 135]). At a high level, the writeback occurs when a dirty line is shared with another cache, such as producer-consumer sharing [48] (repeated writer-reader).

For instance, in Fig. 4.2, core $C0$ has a dirty copy of a line, and core $C1$ requests a read-only copy. Given the *single-writer, multiple-reader* invariant (§4.2.3), lines valid in multiple cores' caches must be read-only, meaning $C0$ must transition from **M** (dirty+writable) to a read-only state to share the line.

While a conventional MOESI protocol (left) allows the responder $C0$ to transition **M** $\rightarrow$ **O** (where **O** encodes *dirty*+read-only), MESI's sole read-only state is **S** (*clean*+read-only). Thus, MESI (right) instead incurs a *downgrade writeback*, such that $C0$'s and $C1$'s copies of the line become clean (**S**) to satisfy protocol requirements.

To test the theory of hammering via ccNUMA downgrade writebacks, I pinned our workloads to a single node—so that downgrade writebacks would go to the node's LLC, not DRAM—and

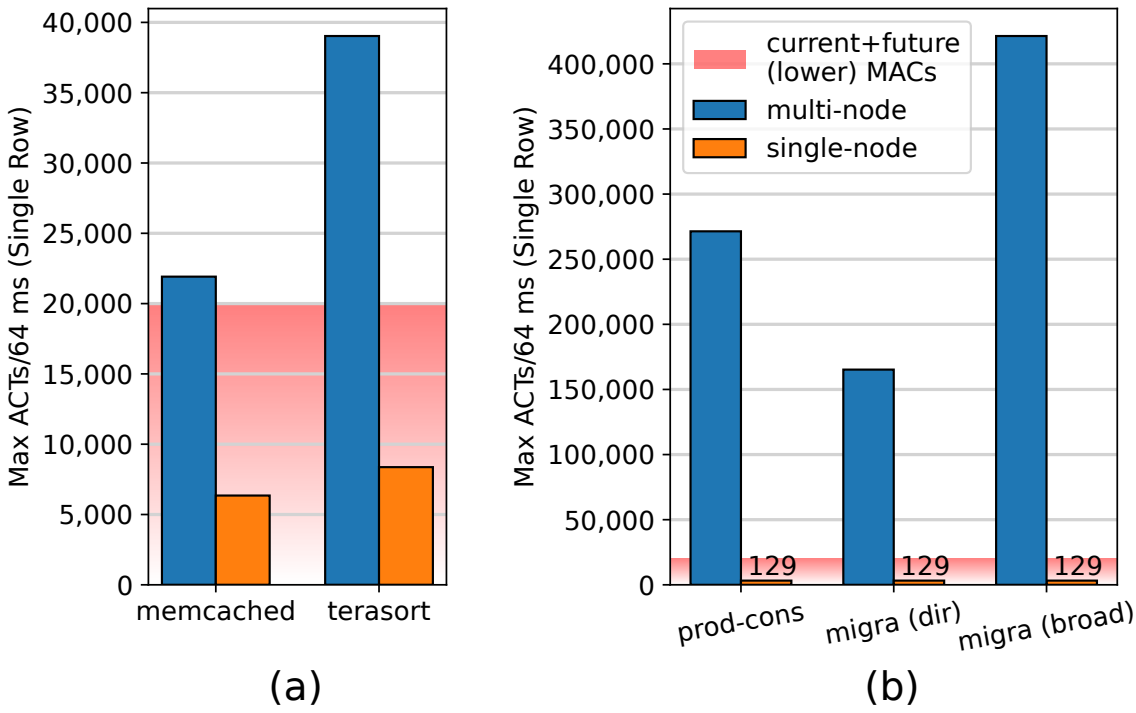

Figure 4.3: Activation (ACT) rates on a major cloud provider's production hardware for (a) commodity benchmarks and (b) worst-case micro-benchmarks. In both cases, dirty sharing across NUMA nodes yields ACTs in excess of current Rowhammer thresholds (MACs).

recorded new traces. As shown in Fig. 4.3(a), the ACTs observed for the cloud workloads (along with micro-benchmarks discussed shortly) *drastically* dropped, from 21,917 to 6,349 (*memcached*) and 39,031 to 8,369 (*terasort*). Furthermore, I observed *more* reads than writes for cache lines within the hot rows, as conventionally expected. This provided strong evidence that downgrade writebacks were causing frequent DRAM writes and preceding ACTs.

To confirm this evidence, I wrote a micro-benchmark (*prod-cons*) designed to generate coherence-induced hammering via downgrade writebacks. More specifically, the benchmark schedules two threads: a producer and a consumer. The producer repeatedly writes to two different cache lines at physical addresses A and B in an alternating fashion, while the consumer repeatedly reads these lines in an alternating fashion. When the consumer reads the producer's **M** copies, a downgrade writeback should occur per MESI requirements.

Notably, I select physical addresses A and B such that they map to different rows within the same bank of DRAM. Thus, alternating downgrade writebacks of the lines necessitate repeated ACTs due to row buffer contention (§4.2.1). I again ran the experiment in two configurations: (1) with the threads pinned to separate NUMA nodes, where downgrade writebacks go to DRAM, and (2) with both threads pinned to a single node, where downgrade writebacks go to the node's LLC.

Echoing the cloud workload data, Fig. 4.3(b) shows that the multi-node execution of *prod-cons*

produces "hammer-level" rates of ACTs (*over 250,000 ACTs* in 64 ms to a single row, $> 10\times$ modern MACs), while the single-node execution does not hammer (just 129 ACTs in 64 ms). The multi-node experiment is therefore indicative of "worst-case" behavior for ccNUMA downgrade writebacks.

After confirming *clean* sharing (i.e., read-only, and thus free of downgrade writebacks) did not yield hammering in either configuration, I concluded that downgrade writebacks are a source of coherence-induced hammering.

Thus, *workloads exhibiting producer-consumer sharing can inadvertently yield coherence-induced hammering on Intel ccNUMA servers*. Adversaries can also *intentionally* hammer by using this common sharing pattern across NUMA nodes—without previously-exploited primitives like cache line flushes, eviction sets, or DMAs.

4.3.3 Source #2: Memory Directory Writes

To determine if downgrade writebacks are the *only* source of coher-ence-induced hammering on Intel ccNUMA servers, I wrote a second micro-benchmark designed to (1) still generate dirty sharing between cores, but (2) not incur downgrade writebacks. Intuitively, if downgrade writebacks were the only hammering source, then our benchmark would not hammer.

Our second micro-benchmark—*migra*—is similar to the previous, except *both* threads write to the line (i.e., migratory sharing [58,284], or "repeated writer-writer"). Because Intel's MESI-based protocol requires a downgrade writeback upon a Get-**S**hared (read-only) request for a line in **M**, I avoid downgrade writebacks by only sending Get-e**X**clusive (read-write) requests between the cores (via stores).

More specifically, if core c_1 has a line in **M**—and core c_2 issues a Get-**X** for the line—core c_1 sends its copy to c_2 and transitions $\mathbf{M} \rightarrow \mathbf{I}$. Thus, c_2 receives the line in **M** (for its own writing) without a writeback. Notably, the behavior of this sharing pattern is identical in conventional MESI and MOESI protocols (given the **S/O** states are not used). Thus, our experiment offers insight both on how the existing Intel protocol behaves and an otherwise-identical MOESI protocol *would* behave. Our simulations comparing MESI and MOESI implementations in §4.6 confirm this reasoning.

As with *prod-cons*, I run *migra* with the threads scheduled on different nodes and on the same node. I refer to *migra* executed atop the default memory directory ccNUMA protocol as *migra (dir)*, in order to differentiate from a separate execution discussed shortly. Fig. 4.3(b) shows that the multi-node experiment *still hammers* (165,233 ACTs). Furthermore, I find the contended cache lines again experience more writes than reads in DRAM. In contrast, and as expected, the single-node experiment does not hammer.

I discovered that others had also reported unusually-high DRAM writes on Intel Skylake servers [120], and suspected memory directory (§4.2.3) writes as the cause. In particular, remote requests for a local cache line may require a DRAM write to track remote copies via memory directory state [202, 203] (e.g., remote-**I**nvalid → snoop-**A**ll), incurring extra writes. Furthermore, because the on-die directory cache for select **A** lines uses write-on-allocate [208] (akin to write-through), even directory cache allocations immediately incur DRAM writes.

I reran our migratory sharing micro-benchmark with the default memory directory protocol disabled in the BIOS—reverting to a *broadcast* ccNUMA protocol to execute *migra (broad)*—to isolate memory directory writes as the culprit. I found that the *write*-based hammering was eliminated when executing *migra (broad)* across NUMA nodes. I therefore conclude that memory directory writes are another source of coherence-induced hammering in Intel's and an otherwise-identical MOESI memory directory protocol during dirty sharing.

4.3.4 Source #3: Speculative Reads

While I no longer observed write-based hammering, I instead observed *read*-based hammering caused by the same lines in *migra (broad)*—421,360 ACTs in Fig. 4.3(b). In fact, I consistently noticed repeated reads of the contended lines in *migra (dir)* as well, albeit two orders-of-magnitude fewer than *migra (broad)*. This hammering was again eliminated when pinning the workload to a single node, indicating a third source of coherence-induced hammering.

Suspecting hardware prefetching as the source of hammering reads, I disabled all prefetchers listed in the BIOS, but still observed repeated reads of the lines. Thankfully, prior work [120, 202] notes an additional source of DRAM reads in broadcast protocols: speculative reads by the home agent.

Namely, upon an LLC miss, broadcast protocols tend to do two operations in parallel as a performance optimization: (1) broadcast snoops to other nodes, and (2) *speculatively read from DRAM*—jump-starting the read that becomes necessary if the snoops fail. Therefore, because migratory sharing among NUMA nodes induces frequent LLC misses for the shared line(s), I believe the corresponding mis-speculated (unused) DRAM reads form a source of coherence-induced hammering.

In particular, during migratory sharing, the line is often (and during our stores-only micro-benchmark, essentially always) in **M** on one of the nodes, meaning no valid copies exist on other nodes. Thus, if node n_1 holds the line in **M**, and a core on node n_2 requests a copy, the request incurs an LLC miss. Subsequently, the home agent issues, in parallel, (1) snoops to the other nodes, one of which will return n_1's dirty copy of the line, and (2) a speculative DRAM read, which will go unused due to the successful snoop response from n_1. As the sharing pattern repeats, so too do

the mis-speculated (unused) DRAM reads, yielding coherence-induced hammering.

To explain the reduced—but nonetheless repeated—number of reads when using the default directory protocol, recall that directory cache *hits* obviate the need for DRAM reads of migratory lines, since they indicate the line must be snooped (§4.2.3). Thus, I infer that the remaining reads appear to indicate directory cache *misses*. Because our micro-benchmark only migrates two lines between the nodes, I find it unlikely that the misses arise from set conflicts in the directory cache (i.e., conflict misses).

Instead, I believe a phenomenon similar to a documented [54] behavior in AMD's MOESI directory protocol is occurring. In particular, when a remote request arrives at the home agent, AMD issues speculative DRAM reads in parallel to local LLC lookups to reduce latency. While Intel's directory cache can prevent these speculative DRAM reads, their patent [208] indicates entries are de-allocated when the local node requests a copy of the line (since, under MESI, the remote will no longer be dirty after responding to the request, obviating the performance benefit of a directory cache entry).

Thus, if a remote request for the line arrives at the home agent after de-allocation, a directory cache miss occurs. At this time, I believe a DRAM read and (local) snoop occur in parallel, just as in AMD's MOESI directory protocol and Intel's MESI broadcast protocol. This explains the remaining hammering DRAM line reads in our directory traces. Therefore, I conclude that coherence-induced speculative DRAM reads can occur in commodity broadcast *and* directory cc-NUMA protocols, irrespective of the use of MESI or MOESI.

4.3.5 Why This Hammering is Problematic

Commodity workloads producing ACT rates known to induce bit flips [52, 147] is a significant cause for concern among cloud providers for several reasons. First, recent studies of data center reliability (e.g., Facebook [66] and Google [104]) have found increasing rates of silent data corruption. Given data corruption is a symptom of Rowhammer—and the community is yet unable to attribute production occurrences to Rowhammer—cloud providers must treat Rowhammer as a potential cause and take appropriate precautions. In particular, silent corruption yields arbitrary behavior, while detected-but-uncorrected corruption yields machine check exceptions (i.e., denial-of-service). Even *corrected* data corruption, used as a proxy for hardware reaching end-of-life, can unnecessarily increase costs.

Second, irrespective of whether *today's* data corruption arises from Rowhammer, cloud providers also need to protect data in *tomorrow's* DRAM. Unfortunately, future DRAM is expected to be more susceptible to Rowhammer [147]. Specifically, given the various benefits of denser DRAM (e.g., performance), manufacturers are projected to increase density—increasing

Rowhammer susceptibility in turn. This projection is supported by prior work [52, 147, 261], which shows that newer, denser DRAM (1) requires fewer ACTs per row to flip bits, and (2) can experience more rows simultaneously surpassing these decreased MACs. Notably, state-of-the-art Rowhammer attacks [75, 98, 125, 233, 240, 300] already exploit as few as 3 rows simultaneously surpassing MACs in order to overwhelm existing mitigations (TRR, §4.2.1) and flip bits.

Our traces therefore offer the first evidence that ccNUMA systems depend on (vulnerable) mitigations to prevent bit flips *triggered by commodity workloads*. Furthermore, while TRR can prevent bit flips that would be caused by the small number of simultaneous aggressors observed within a *single* benchmark (e.g., 1-2), cloud providers must account for numerous individual applications simultaneously hammering and thereby bypassing TRR, an increasingly-likely phenomenon given declining MACs.

Third, while state-of-the-art alternative mitigations [21, 199, 236, 256, 336] can provide comprehensive protection against bit flips, their performance and area overhead rises with increasing susceptibility. While it may be acceptable to slow a malicious Rowhammer attack workload, our finding of coherence-induced hammering in commodity workloads demonstrates that *non-malicious* applications could additionally experience slowdowns proportional to Rowhammer susceptibility. Thus, prior work [147, 183] concludes that software vendors such as cloud providers have a vested interest in exploring and mitigating the phenomena leading to high activation rates *before* widespread problems arise.

4.4 Design of MOESI-prime

In light of modern ccNUMA protocols' susceptibility to coherence-induced hammering, I present a novel protocol—MOESI-prime—that prevents identified sources of coherence-induced hammering in both commodity and malicious workloads. Notably, MOESI-prime achieves such protection while retaining use of Intel's state-of-the-art memory directory design for scalability.

MOESI-prime is designed as simple, well-defined modifications to a baseline memory directory protocol. I build atop the MOESI states, given MESI's susceptibility to hammering downgrade writebacks (§4.3.2). I describe MOESI-prime's novel mechanisms to prevent both hammering directory writes (§4.4.1) as well as hammering speculative reads (§4.4.2), and discuss a safe protocol performance optimization (§4.4.3).

4.4.1 Preventing Hammering Directory Writes

As discerned in §4.3.3, hammering directory writes occur during repeated *dirty* sharing across nodes (i.e., sharing with at least one writer). Thus, MOESI-prime's goal is to obviate the need for

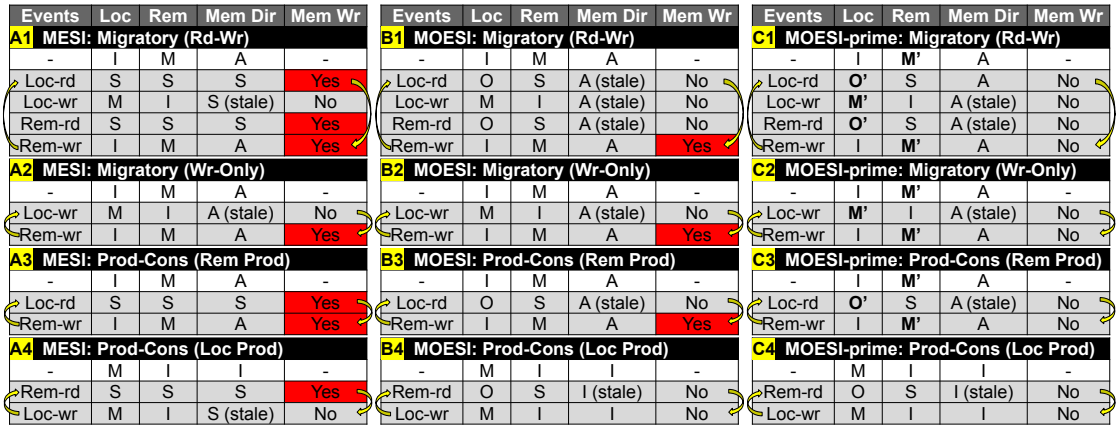

Figure 4.4: Dirty, inter-node sharing in MESI (A1–A4), MOESI (B1–B4), and MOESI-prime (C1–C4) memory directory protocols. Hammering writes (red) are incurred during arrow-denoted cycles. MOESI and MOESI-prime prevent MESI's downgrade writebacks via the **O** state. MOESI-prime also prevents MOESI's redundant writes via new **M'** (**M** + mem dir in **A**) and **O'** (**O** + mem dir in **A**) states. MOESI and MOESI-prime use the "greedy local ownership" optimization introduced in §4.4.3.

directory writes during repeated dirty sharing. I compare this approach to a writeback directory cache—which would at-best reduce the frequency of these (as I will show, unnecessary) writes—in §4.7.2.

Given a local node and one or more remote nodes, I consider dirty sharing between a local and remote node, as well as between two remotes and among more than two nodes.

4.4.1.1 Local-Remote Sharing

There are two basic forms of repeated dirty sharing: migratory [58, 284] (writer-writer) and producer-consumer [48] (writer-reader). Each pattern can be divided into two subcategories. For migratory, there is (1) read-write (where the writers read the line before writing) and (2) write-only. For producer-consumer in ccNUMA, there is the case of a (3) *remote* producer versus a (4) *local* producer.

Thus, Fig. 4.4 shows how MESI (A1–A4), MOESI (B1–B4), and MOESI-prime (C1–C4) memory directory protocols behave during these scenarios. MESI hammers during all forms of dirty sharing, primarily due to downgrade writebacks that are trivially-eliminated by MOESI and MOESI-prime. I therefore mainly focus on the difference between MOESI and MOESI-prime in the remainder of this subsection.

In the unique case of producer-consumer with a local producer, remote node(s) (the consumers) *never* write to the line. Under MOESI (B4) and MOESI-prime (C4), once the line transitions to **M** upon the local node's first write, it remains dirty on the local node (**M** or, if shared, **O**) until written

back.

Crucially, when remote consumer(s) read the line, the memory directory can remain unchanged (potentially stale) until the local copy is written back. This is because the home agent *must* check the local node for a dirty copy upon a remote request, since the memory directory only tracks *remote* coherence state. If a dirty copy is locally-present, the home agent will forward this copy in lieu of the stale memory copy/directory state. Thus, producer-consumer sharing with a local producer already does *not* require repeated directory writes, avoiding directory write-based hammering without changes to a baseline MOESI protocol.

However, in the migratory sharing subcategories and producer-consumer (remote producer), MOESI can hammer (B1–B3). In particular, when a remote node writes to a locally-owned line, the home agent has no way of knowing if the memory directory is already in snoop-**A**ll (albeit possibly stale). Thus, the home agent must conservatively (i.e., potentially redundantly) write **A** to the memory directory. As the remote writer repeatedly acquires exclusive access under such sharing, the directory writes repeat, hammering DRAM.

MOESI-prime exploits the insight that these "dirty sharing" writes can be avoided *if* the home agent knows the memory directory is already in **A**. Thus, MOESI-prime provides an additional "prime" state for each MOESI dirty state (**M** and **O**) to encode this information (C1–C3). **M'/O'** indicate a line is in conventional **M/O**, *and* the memory directory is in **A**.

Intuitively, the prime (**M'** and **O'**) states' prevention of repeated directory writes can be likened to how MOESI's **O** state prevents downgrade writebacks. In MOESI-prime, when a remote writer first writes to a line, the line enters **M'** (given it is dirty+writable on the remote node and **A** in the memory directory). From this point until the prime line's eventual writeback, MOESI-prime enforces two invariants: (1) the line remains prime, and (2) the memory directory is *not* updated.

Accordingly, the home agent knows any line in **M'** (or, if shared, **O'**) is in **A** in the memory directory. Thus, when a remote node writes to a prime line, the home agent can omit the redundant directory write, preventing hammering.

4.4.1.2 Remote-Remote and > 2-Node Sharing

Fig. 4.4 does not depict dirty sharing between two remote nodes (only between a local and a remote), because this sharing is already free of hammering directory writes under MOESI (and hence, MOESI-prime). Specifically, when a remote r_1 requests a dirty line from another remote r_2, the home agent knows that the memory directory must (1) already be in **A**, and (2) remain in **A** until the dirty copy is written back, guaranteeing a directory write is not needed. For MESI, hammering downgrade writebacks occur regardless of which nodes share an initially-dirty line.

Additionally, Fig. 4.4 only shows sharing between 2 nodes. While > 2 nodes can clearly share a line, the memory directory enters **A** (and remains in **A**) so long as *any* remote holds a dirty copy.

If the local node becomes the owner of the dirty copy, the directory entry can be left in **A** (stale), as the local dirty copy will be snooped and override the stale copy in DRAM. Thus, additional sharers do not affect MOESI-prime's ability to prevent hammering directory writes.

4.4.2 Preventing Hammering Speculative Reads

MOESI-prime's key insights for preventing speculative hammering reads are that (1) these reads arise under the same "dirty sharing" scenarios as redundant directory writes (§4.3.4), and (2) requests that hit in the directory cache do *not* result in DRAM reads. Thus, MOESI-prime's goal is to ensure that requests for contended lines almost always hit in the directory cache, whether issued by local or remote nodes.

Unlike hammering directory writes, MOESI-prime does *not* use additional state to prevent hammering reads. Instead, MOESI-prime makes a minor modification to the directory cache behavior described by Intel's patent [208]. Rather than de-allocating/not allocating a directory cache entry when a line migrates to a local writer, MOESI-prime retains/provisions an entry, now pointing to the local node. Thus, subsequent requests will hit in the directory cache, avoiding speculative DRAM reads.

While this policy yields additional contention for directory cache entries, the only lines affected are those that are either (1) cache-to-cache transferred to a remote writer (such that a directory cache entry is allocated), and then transferred to a local owner, or (2) invalidated on remote node(s) by a local writer. As I will show in §4.6, MOESI-prime has negligible effect on performance versus MESI and MOESI baselines, despite this modest increase in contention.

I note that even with MOESI-prime's policy, repeatedly generating set conflicts in the directory cache remains a possible way to maliciously hammer, since the conflict-induced misses could result in hammering reads. However, unlike coherence-induced hammering, I find no evidence of *conflict-induced hammering* in commodity workloads. Furthermore, conflict-induced hammering can be mitigated via existing mechanisms to prevent frequent set conflicts [29, 239, 243, 244, 254, 318], which are complementary to MOESI-prime's protection against coherence-induced hammering.

4.4.3 Optimization: Greedy Local Ownership

Prior work [6,210] shows that MOESI-based protocols can implement different *ownership* policies without loss of correctness when sharing dirty lines. The ownership policy designates whether the requesting cache or responding cache ends a transaction as the line owner. For instance, in the conventional MOESI protocol depicted in Fig. 4.2 (§4.2.3), the responder (C0) retains ownership (**M** $\rightarrow$ **O**) while the requestor (C1) enters **S**. Conversely, in AMD's "Always-Migrate" ownership

policy [174], the responder C0 relinquishes ownership (**M** $\rightarrow$ **S**), and the requestor C1 acquires ownership (enters **O**).

I provide the additional insight that MOESI-based protocols can optimize the ownership policy for improved ccNUMA performance. Consider that (1) an inter-node request goes to the home agent, (2) the home agent forwards the request to the owner, and (3) the owner responds to the requestor. If the owner is *local* (i.e., on the home agent's node), a NUMA hop (interconnect traversal) can be avoided in step (2). Thus, there is benefit in making the local node the owner when possible.

MOESI-prime (and our MOESI baseline) accordingly incorporate a *greedy local ownership* policy to reduce interconnect traffic and latency, as used in Fig. 4.4. This policy ensures that if a dirty line is shared for reading between a local and remote node (i.e., upon a Get-**S**hared request) the local node ends the transaction as the owner (**O/O'**), while the remote becomes a sharer (**S**). Subsequent requests for the line are thus forwarded to this local owner, reducing NUMA latency and contention.

4.4.4 Key Takeaway

MOESI-prime prevents each of the identified sources of coherence-induced hammering: downgrade writebacks via **O/O'**, directory writes via new **M'** and **O'** states, and speculative reads via changes to Intel's directory cache management policy. In §4.6, I show that these protections prevent coherence-induced hammering across a broad range of non-malicious and malicious workloads.

4.5 Protocol Correctness

In this section, I demonstrate that MOESI-prime's two key protocol extensions—the prime (**M'** and **O'**) states and directory cache modifications—preserve coherence. I do so by showing that the addition of MOESI-prime's extensions to an initially-correct baseline memory directory protocol does not allow programs to produce previously-forbidden results.

I assume an Intel-like MESI baseline can be extended with the widely-used **O** state to form an otherwise-identical MOESI protocol. I thus reason about MOESI-prime in the context of a MOESI baseline. I also assume that the MOESI baseline correctly implements greedy local ownership (§4.4.3), noting that prior work [6, 174, 210] demonstrates the validity of denoting either the requestor or responder as the owner.

I additionally note that MOESI-prime's detailed set of coherence states and transitions (along with those of the baseline protocols) are provided in our open-source implementation [185].

4.5.1 Correctness of M' and O' States

For this proof, I model ccNUMA systems as transition systems [15] with states S and a transition relation T. Each state in S represents a state of the entire system at one point in physical time, including all coherence states and the values of every cache line and address in main memory read from or written to by program instructions. A state s_1 can transition to a state s_2 (i.e., $(s_1, s_2) \in T$) if a valid coherence state transition for a cache or directory in state s_1 can lead to state s_2, or if the writing of a value to a cache line or directory entry can change s_1 to s_2. I define a *trace* (i.e., an execution) of a transition system (S, T) as a sequence $s_0 s_1 s_2 s_3 ... s_n$ where $s_0 \in S$ represents the state of the system upon startup and $\forall\, 1 \leq i \leq n, s_i \in S \wedge (s_{i-1}, s_i) \in T$.

Let D and D' be transition systems where D represents the baseline MOESI system and D' represents this baseline with the addition of the **M'** and **O'** states. To prove the correctness of adding the prime states, I first prove the following lemma.

Lemma 1 *Consider any cache line l. Let s_1 be a state that exists in D and D', meaning no cache has a copy of l in **M'** or **O'**. Assume that s_1 in D' can transition to a state s'_2 in D' which has a line for l in **M'** or **O'**. For the case of D, s_1 can transition to s_2, which is identical to s'_2 except that **M'** and **O'** are replaced by **M** and **O**. In addition, define a* completed Put *as a Put request (i.e., writeback) for l that is processed by the directory without another core acquiring ownership of l during the transaction (e.g., by a Get-**X** reaching the directory before the Put). Then, the following conditions hold:*

1. *s_2 and s'_2 will have memory directory states of **A**.*
2. *In the subsequent execution of D, the memory directory state for l will remain **A** until a completed Put occurs.*
3. *In the subsequent execution of D', once a completed Put occurs, no core can then transition to **M'** or **O'** for l until a core becomes a remote owner of l.*

Proof. Condition (1): Starting from state s_1 in D', a line for l has no way to enter **O'** without entering **M'** first. From state s_1, a line for l may enter **M'** in s'_2 in one of two ways, both through the actions of a remote core R. First, R may issue a Get-X request for l. This results in R receiving the line in **M'** and the memory directory entry for l being updated to **A**. The second possibility is if R was in **E** for line l in s_1, and then silently wrote to the line l and transitioned to **M'**. In this case, the memory directory would have been set to **A** when R entered **E**, and would have remained in **A** through the transition to **M'**. This is because changing the memory directory state would require another core to request the line, which would result in R losing its write permissions and thus

being unable to transition to **M'**. The two scenarios in D equivalent to these cases can be obtained by replacing **M'** with **M**. In both cases, the memory directory is set to **A** in s_2. Since the memory directory is in **A** in s_2 and s_2', condition (1) is fulfilled.

Condition (2): Once the memory directory state in D is **A**, by the transition rules of the protocol, the only way for the directory state to change to something other than **A** is for the owning core to execute a Put request (either Put-X or Put-O for **M** and **O** respectively). If this is a completed Put, the directory state will change to **I** (for a Put-X) or **S** (for a Put-O). Note that if a concurrent request that results in ownership is processed by the directory before the Put (i.e., the Put is not a completed Put), ownership will be transferred to the requestor and the requestor will transition to **M** or **O** as appropriate. While the previous owner's Put will be acknowledged, the memory directory state will remain in **A** due to there still being an owner in the system. Thus, condition (2) is satisfied in all cases.

Condition (3): Consider the execution of D' from s_2' onwards. Once a core transitions to **M'** for l in s_2', an instance of **M'** or **O'** for l will remain in the system as long as there is an owner, i.e., until a completed Put occurs. Consider the first such completed Put. By the rules of the protocol, this completed Put must have been issued by the owning core, denoted by C. By virtue of being the owner, C must be in **M'** or **O'** when issuing the Put, and thus no other cores can be in **M'** or **O'** at this point. Furthermore, since C's Put is a completed Put, no other core will gain ownership before C's Put is processed by the directory. The completed Put will relinquish C's ownership (i.e., C will no longer be in **M'** or **O'**). Thus, no instances of **M'** and **O'** remain in the system for line l. Any subsequent instances of **M'** and **O'** in the execution must arise from a core becoming a remote owner of l through one of the two possibilities discussed in the proof of condition (1). Thus, condition (3) is satisfied.■

Using Lemma 1, I can prove Theorem 1 below, showing that **M'** and **O'** do not introduce new program outcomes.

Theorem 1 *For every trace d' that can be generated by D', there exists a trace d that can be generated by D such that the values of every cache line and address in main memory in the final states of d and d' are identical.*

To prove Theorem 1, I create trace d by substituting all instances of **M'** by **M** and **O'** by **O** in the trace d'. **M** and **O** are semantically equivalent to **M'** and **O'** respectively, apart from the writes to the memory directory that **M** and **O** add. Thus, as long as the extra memory directory writes added by this substitution do not change the memory directory state, trace d will be a valid trace for the baseline MOESI system D (since d does not contain any instances of **M'** or **O'**).

Any such extra memory directory writes in d will occur over the events in d corresponding to those in d' between when a core entered **M'** for a given line and when the next completed Put for

that line occurred. (The completed Put of a line removes all existing instances of **M'** and **O'** for that line from the system by condition (3) of Lemma 1). By conditions (1) and (2) of Lemma 1, the memory directory state is guaranteed to be **A** over these ranges for any such lines in d. Thus, any extra memory directory writes in d are guaranteed not to change their corresponding memory directory states from **A** to another state. As a result, all the coherence transitions in trace d remain valid transitions. Trace d is thus a valid trace in the baseline MOESI system. Since I do not change the values of cache lines or main memory when creating trace d from d', the final states of d and d' have identical values for every cache line and main memory address that they model. d thus satisfies the requirements of Theorem 1.■

4.5.2 Correctness of Directory Cache Modifications

MOESI-prime's directory cache modifications eliminate select speculative DRAM reads, based on the understanding that the results of the eliminated reads will always be discarded due to mis-speculation. In the baseline protocol, a hit in the directory cache implies that the line is dirty on a *remote* node. Thus, DRAM need not be read on a directory cache hit, as a snoop of the remote owner will succeed and return the data (§4.3.4).

While the baseline does not provision a directory cache entry if the home node becomes the owner (as described in Intel's patent [208]), MOESI-prime's modification ensures that a directory cache entry also exists in this scenario, pointing to the *local* node. Thus, the invariant that a directory cache hit means that a snoop will succeed is maintained under MOESI-prime.

Specifically, in the new case where the directory cache entry corresponds to local node ownership, it is the local node that will service a snoop, once again making it unnecessary to read DRAM. In addition, since the local node is a dirty owner, an eviction requires a writeback, ensuring the directory cache's knowledge will remain current. If ownership is transferred back to a remote node, the directory cache entry will be updated to point to the remote node, and speculative DRAM reads will be prevented according to the baseline policy.

4.6 Evaluation

I evaluate MOESI-prime in gem5 v21.1.0.2 [24, 188]. I extend an existing directory coherence protocol in the Ruby subsystem to model Intel's memory directory and directory cache. I use a full-system mode configuration with simulation parameters, such as total cache per core and clock speed, that model a major cloud provider's production hardware. For tractable simulation times, I use simple in-order cores atop detailed cache, coherence, and DRAM models. This configuration follows prior work [82], which demonstrates that out-of-order versus in-order execution does

Parameter	Value
TimingSimpleCPU	x86-64, 2.6 GHZ, 8 cores (no SMT), cycle-accurate instr fetches + loads/stores, else 1 cycle/instr; non-pipelined
I-/D-Cache	32 KB, 8-way set associative (SA), 4 cycle RT latency
LLC	2.375 MB/core, 32-way SA, 42 cycle RT, non-inclusive
Directory Cache	16 KB/core, 1B entry, 32-way SA, parallel access w/ LLC
DRAM	16 GB DDR4, 2400 MHz, 2Rx4 (32 banks/node), FR-FCFS [253] scheduling, RoCoRaBaCh [95] address mapping, adaptive page policy, mean 37.5 ns read RT to home agent
NUMA	2, 4, 8 nodes; cores+mem split/node; 32 ns RT interconnect
OS/Kernel Config	Ubuntu 20.04/Linux 5.4.0-88-generic (except as noted)

Table 4.1: gem5 simulation configuration.

not significantly affect the memory system characterization of commodity workloads, given the detailed memory system model.

I run the Ubuntu 20.04 operating system with its default kernel configuration, aside from patches to (1) remove unsupported drivers, and (2) infer the gem5 hardware's NUMA configuration from a boot parameter, since gem5 does not implement BIOS mechanisms that normally report this information to the OS. Our system configuration is listed in Table 4.1.

I compare MOESI-prime to MOESI and MESI memory directory protocols, with the respective protocol enforced for inter-node coherence. For a fair performance comparison, both applicable protocols (i.e., MOESI-prime and MOESI) use our greedy local ownership optimization (§4.4.3). I evaluate 2-node (production-like), 4-node, and 8-node configurations. Cumulative amounts of cache, DRAM, and cores are held constant, split evenly among nodes.

I run 8-thread (1 per core) benchmarks from the PARSEC 3.0 [344] and SPLASH-2x [321] suites, simulating the region-of-interest for each benchmark with the *simmedium* input size. I omit 3/26 benchmarks due to runtime errors on real hardware (*fmm* [260]) and use of unsupported x86-64 instructions in gem5 (*volrend* and *x264* [79, 80, 188]). I are unable to additionally simulate *memcached* and *terasort* (§4.3.1) due to lack of functional IP networking in gem5 for our configuration. For malicious workloads, I use producer-consumer (*prod-cons*, §4.3.2) and migratory sharing (*migra*, §4.3.3) micro-benchmarks that trigger coherence-induced hammering in the baseline protocols.

4.6.1 Highest Activation Rate

To assess the effectiveness of MOESI-prime's mitigations, I analyze the maximum number of ACTs to a single row within any 64 ms refresh window during benchmark execution.

4.6.1.1 Non-Malicious Workloads

Fig. 4.5 depicts the highest ACT rates for each PARSEC 3.0 and SPLASH-2x benchmark, as well as the arithmetic mean per configuration.

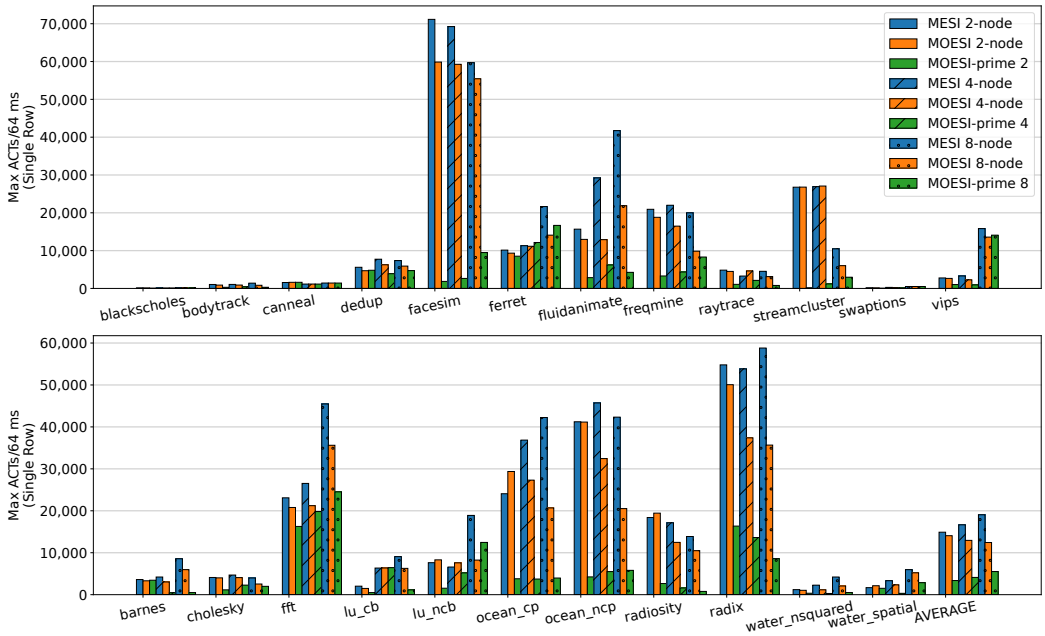

Figure 4.5: Highest ACT rates for PARSEC 3.0 [344] and SPLASH-2x [321] benchmarks across MESI, MOESI, and MOESI-prime.

I find that MOESI-prime's mitigations for coherence-induced hammering reduce highest ACT rates on average by 77.38% (2-node), 75.30% (4-node), and 71.06% (8-node) compared to MESI. In contrast, MOESI only prevents downgrade writebacks, and achieves at best a 34.71% decrease (8-node), with just a 5.58% decrease in the 2-node configuration.

Under MOESI-prime, each benchmark's maximally-activated row receives an average of 20.62%, 26.81%, and 28.29% (2-, 4-, 8-nodes) coherence-induced ACTs—i.e., ACTs due to memory directory reads/writes (and downgrade writebacks in the case of MESI). In contrast, the maximally-activated rows under MOESI experience an average of 94.53%, 88.01%, and 85.78% coherence-induced ACTs, demonstrating that MOESI-prime eliminates coherence traffic as the dominant source of ACTs. While MESI's numbers are skewed by downgrade writebacks (which can yield subsequent demand reads), I find that coherence-induced ACTs are still the dominant source for the maximally-activated rows (85.29%, 74.85%, and 53.31%).

I additionally find that MOESI-prime's *second* maximally-activated row in the same bank during each benchmark's "worst-case" 64 ms window sees ACT rates decline by 29.99%, 29.07%, and 44.41% (2-, 4-, 8-nodes) on average compared to the maximally-activated row. The baselines' larger average decreases (MOESI: 64.50%, 56.04%, 69.07%; MESI: 67.86%, 55.84%, 75.45%) indicate that it is common for a single row to experience significantly more coherence-induced hammering than the rest within a bank.

MOESI-prime's increase in highest ACT rates for 4- and 8-node configurations is expected, and

still results in significant reductions over the MESI and MOESI baselines. These configurations (1) represent increasingly-strained scheduling scenarios (e.g., all sharing is inter-node in the 8-node configuration), (2) artificially-reduce directory cache size per node to keep the total amount constant, and (3) require the directory cache to cover a greater portion of remote memory (e.g., 7/8 of memory is remote for 8 nodes, compared to 1/2 for 2 nodes). Nonetheless, the remaining possibility of high maximum ACT counts (e.g., *fft* with 8 nodes, with 48.41% of these ACTs still coherence-induced directory reads/writes) shows (a) room for further improvement (e.g., via atomic directory read-modify-writes to yield 1 ACT instead of 2), and (b) the benefit of scheduling workloads across as few NUMA nodes as possible.

I conclude that MOESI-prime's mitigations for coherence-induced hammering are extremely effective at reducing highest ACT rates in non-malicious workloads.

4.6.1.2 Malicious Workloads

I find that both MESI and MOESI allow highest activation rates to surpass 500,000 to the shared cache lines' rows within 64 ms during *prod-cons* and *migra*. Conversely, MOESI-prime keeps highest ACT rates below 200 per 64 ms, a $>$2,500$\times$ improvement, and the hottest rows are *not* those of the shared cache lines (meaning MOESI-prime prevents hammering of the contended rows).

4.6.2 Performance

I depict MOESI-prime's and MOESI's per-benchmark execution speedup across 2-, 4-, and 8-node configurations in Table 4.2 (§4.6.2), normalized to respective MESI baselines. I find that MOESI-prime's mitigations for coherence-induced hammering yield negligible performance impact compared to MESI and MOESI (-0.51%–$+0.61\%$, depending on the configuration and baseline protocol).

MOESI-prime can *improve* performance in many workloads thanks to its elimination of unnecessary DRAM reads and writes. In particular, this elimination yields reduced contention for DRAM bandwidth and line-fill/writeback buffers.

Nonetheless, select workloads can experience slightly decreased performance under MOESI-prime for multiple reasons. First, an unnecessary speculative read or redundant write in the baselines—eliminated by MOESI-prime—may activate a row that will be used by a subsequent read, decreasing the subsequent read's latency via a row buffer hit. Second, a speculative read may prevent a switch to write scheduling in a DRAM controller, avoiding a bus-turnaround latency for subsequent reads. Third, MOESI-prime's increased usage of the directory cache to prevent hammering speculative reads can evict entries that would otherwise speed up remote snoops.

§4.6.2: MESI-Normalized Execution Speedup %						
	2-node		4-node		8-node	
Bench	MOESI	Prime	MOESI	Prime	MOESI	Prime
blacksc.	+0.01	-0.04	+0.00	+0.00	+0.01	+0.01
bodytra.	-0.73	+0.03	+0.00	-0.12	+0.03	-0.01
canneal	-3.97	-3.97	+0.09	+0.08	+0.03	+0.03
dedup	+8.32	+6.06	+10.77	-1.44	-1.13	-0.40
facesim	-0.86	+0.07	+0.02	-0.17	-0.02	-0.08
ferret	+6.36	+1.18	-0.85	+3.45	-3.50	-2.24
fluidan.	+0.20	+0.20	-0.27	-0.01	+0.58	+0.53
freqmine	+0.12	+0.11	+0.12	-0.05	-0.12	+0.04
raytrace	-0.55	-0.30	-0.36	-0.28	-0.08	+0.35
streamc.	+0.43	+0.02	+0.78	+0.76	-0.34	-0.22
swapti.	+0.00	-0.00	+0.00	+0.01	-0.59	-0.59
vips	-0.02	-0.08	+0.15	+0.09	+0.35	-0.04
barnes	+0.46	+2.63	+0.31	+0.63	-0.16	+0.18
cholesky	+1.70	+1.72	+0.41	+0.26	-1.48	-1.32
fft	-0.03	+0.14	+0.42	+0.19	+0.50	+0.47
lu_cb	+1.06	+1.37	-0.07	+2.02	+0.15	+0.15
lu_ncb	+1.20	+1.51	+0.67	-1.24	+0.21	+0.64
ocean_c.	+0.81	+0.07	+1.86	+1.79	+0.19	-4.51
ocean_n.	+0.22	-0.43	+1.74	-0.60	-0.02	-0.52
radiosi.	+0.59	+0.59	+0.12	-0.46	-0.35	-1.12
radix	+0.04	+0.19	+7.08	+7.92	+1.00	+1.21
water_n.	+0.01	+0.02	+0.35	+0.37	-0.15	-0.05
water_s.	-1.27	+0.00	+0.88	+0.85	+0.92	+0.88
AVG	+0.61	+0.48	+1.05	+0.61	-0.17	-0.29

§4.6.3: Power Saved			§4.6.4: 2n-Normalized Speedup		
MOESI	**Prime**	**Nodes**	**MESI**	**MOESI**	**Prime**
+0.00%	+0.22%	**2**	-	-	-
+0.06%	+0.12%	**4**	-0.52%	-0.04%	-0.31%
+0.02%	+0.06%	**8**	+0.18%	-0.60%	-0.55%

Table 4.2: Protocols' MESI-normalized execution speedups (§4.6.2), average DRAM power savings (§4.6.3), and 2-node- (2n-) normalized execution speedup (scalability, §4.6.4). Higher is better in each subtable. "Prime" is MOESI-prime.

Finally, we note that the performance of benchmarks such as *dedup* and *ferret* is particularly-sensitive to thread scheduling [23]. Given scheduling is altered both by different NUMA configurations and protocol timings, such sensitivity can lead to higher performance variability.

4.6.3 DRAM Power

I assess MOESI-prime's effects on DRAM power consumption using gem5's support for DRAMPower [42], comparing to 2-, 4-, and 8-node MOESI and MESI protocols in Table 4.2 (§4.6.3). I find that MOESI-prime's prevention of unnecessary DRAM reads and writes slightly improves average power consumption (0.03%–0.22%, depending on the ccNUMA configuration and baseline protocol).

4.6.4 Scalability

I measure each protocol's scalability by comparing its performance in all 4- and 8-node configurations to its 2-node baseline in Table 4.2 (§4.6.4). Each protocol exhibits negligible (within $\pm 1\%$) differences in scalability across evaluated configurations. I conclude that MOESI-prime offers similar scalability to MESI and MOESI.

4.7 Discussion

4.7.1 Broader Applicability

Coherence-induced hammering occurs during commodity workload execution on broadcast and memory directory Intel ccNUMA protocols, with AMD documentation [54] indicating similar coher-ence-induced speculative DRAM reads. Thus, such hammering applies to numerous commodity protocols. As Intel, AMD, and ARM deploy chiplet architectures for increased scalability and yield [3, 25, 116, 222], the chiplets in even a *single* socket will form a ccNUMA system, requiring careful design to avoid coherence-induced hammering. Additionally, given heterogeneous coherence [5, 238, 278] can be architected similar to ccNUMA (e.g., with accelerators as remote nodes), MOESI-prime's mitigations could extend beyond the realm of traditional ccNUMA.

4.7.2 Limitations of a Writeback Directory Cache

Recall that the directory cache uses a write-on-allocate policy (§4.3.3), where snoop-**A**ll (potentially dirty on a remote) is written to the memory directory upon allocation. Given such writes are a source of coherence-induced hammering, a writeback directory cache might appear to be an easy solution.

However, while MOESI-prime's **M'** and **O'** states *prevent* redundant directory writes, a writeback directory cache can at-best delay/reduce them. Capacity evictions of entries for (would-be)

M'/O' lines would still result in unnecessary writes—consuming DRAM cycles, bandwidth, and power—and could still be abused by a malicious adversary to hammer.

Furthermore, the write-on-allocate policy ensures that directory cache entries can be silently evicted (or even detectably-corrupted) without loss of correctness, as the backing memory directory entry is guaranteed to be in (conservatively-correct) **A** and can thus be used instead. On the other hand, a writeback directory cache eliminates this guarantee, requiring additional on-die area for error correction (and writeback) logic.

As evidence that a writeback directory cache alone is insufficient to prevent coherence-induced hammering, "writeback" MOESI yields significantly higher (worse) maximum ACT rates than "write-on-allocate" MOESI-prime across the PARSEC 3.0 and SPLASH-2x workloads. On average, "writeback" MOESI increases maximum ACT rates by 159.56%, 104.71%, and 75.01% (2-, 4-, and 8-nodes). For the maximally-activated workload in each configuration, the increases are 140.47%, 100.39%, and 55.00%, respectively.

Nonetheless, because MOESI-prime only prevents *redundant* (unnecessary) directory writes, a writeback directory cache's deferral of *initial* (necessary) directory writes can complement MOESI-prime's ability to reduce worst-case ACT rates. On average, combining MOESI-prime with a writeback directory cache decreases (improves) maximum ACT rates by 3.69%, 0.57%, and 5.15% (2-, 4-, and 8-nodes). For the maximally-activated workload in each configuration, the decreases are 2.50%, 14.48%, and 15.25%, respectively.

4.7.3 Considerations for Other Hammering

MOESI-prime mitigates the reliability and security threat of coher-ence-induced hammering, but other forms of hammering remain. To our knowledge, all other existing hammering patterns [53, 59, 75, 91, 92, 98, 124, 125, 131, 147, 152, 167, 180, 218, 233, 240, 241, 250, 269, 288, 299, 300, 325] use some combination of repeated flush instructions, set conflicts, or DMAs in order to bypass system caches and thereby repeatedly access DRAM. While these forms of hammering need to be mitigated, they are of a different nature than coherence-induced hammering. In particular, these other patterns are not known to arise in commodity workloads, currently only posing a security (not a reliability) threat. More importantly, MOESI-prime's mitigations for coherence-induced hammering are complementary to mitigations for other current and future hammering patterns.

As a case in point, Cojocar et al. [51] exploit a hammering phenomenon related to the speculative reads found in ccNUMA protocols (§4.3.4). In particular, they hammer using memory directory reads caused by repeated *flushes* of the same invalid cache line(s). Upon receiving a flush for an invalid cache line, the home agent may read the memory directory state to check for remote copies (which must also be flushed). Thus, by repeating this pattern, one can hammer on

applicable ccNUMA platforms.

This "repeated flush" technique could be considered a malicious combination of flush-based and coherence-induced hammering. The pattern is only known to occur in malicious code, and would be mitigated by flush-specific Rowhammer defenses (e.g., virtualizing or throttling `clflush` behavior). In contrast, the coherence-induced hammering introduced in this paper (1) occurs in commodity workloads and (2) does *not* require `clflush` capabilities.

4.8 Related Work

Rowhammer. Rowhammer bit flips were disclosed in 2014 on DDR3 DRAM [152] and followed by attacks across a variety of DRAM technologies, such as DDR4, LPDDR4/5, and HBM. Prior work [140, 141, 142] has also explored the related phenomenon of data-dependent DRAM failures. While existing attacks require carefully-crafted instruction sequences, coherence-induced hammering is the first hammering shown to occur in commodity workloads.

DDR4 and newer DRAM includes *target row refresh* (TRR) as a mitigation. However, attacks have bypassed TRR to flip bits [75, 98, 125, 233, 240, 300]. Recent work [52, 147, 261] shows that newer DRAM is increasingly susceptible to Rowhammer, and that proposed mitigations [144, 171, 337] will incur increasing performance overhead with rising susceptibility (i.e., decreasing MACs, §4.3). Follow-up state-of-the-art mitigations [21, 199, 236, 256, 336] are consistent with this finding. In contrast, while MOESI-prime only prevents coherence-induced hammering, it has negligible impact on performance, and decreases the frequency at which these MAC-dependent defenses would be engaged for commodity workloads.

ccNUMA Systems. Scale-Out ccNUMA [78] reduces remote DRAM latencies by replicating remote data in local DRAM. Other performance optimizations include a cache-line aware interface for performance tuning ccNUMA systems [248], feedback-driven page placement [197, 198], NUMA-optimized locks [33, 64], faster barriers [47], and speculative lock elision [247]. As coherence/consistency must always be maintained, these optimizations could benefit from MOESI-prime's prevention of coherence-induced hammering in high-performance systems.

ccNUMA Coherence Protocols. State-of-the-art ccNUMA protocols are inspired by the DASH architecture's directory protocol [173]. AMD [174] and Improved-MOESI [6] propose an "always migrate" ownership policy similar to MOESI-prime's "greedy local" policy, except MOESI-prime does not migrate ownership from the local node when possible. Other work proposes mechanisms (e.g., coherence states) to optimize producer-consumer [48] and migratory [58, 284] sharing. MOESI-prime complements these techniques, preventing such sharing from hammering DRAM.

Protocol Generation. ProtoGen [234] and HieraGen [235] automatically generate correct-by-design protocols from stable state specifications. To our knowledge, no support yet exists to

automatically generate memory directory ccNUMA protocols.

4.9 Conclusion

In this work, I have provided novel evidence of *coherence-induced hammering* in commodity workloads, the first hammering found to occur in non-malicious code. Given rising susceptibility to Rowhammer, I have designed MOESI-prime, a ccNUMA protocol that prevents identified sources of such hammering, retains Intel's state-of-the-art scalability, improves average DRAM power, and negligibly-affects average performance—even improving the performance of many workloads. As Rowhammer susceptibility continues to rise, solutions that avoid unnecessary row activations such as MOESI-prime will ensure continued reliability and security in the cloud.

CHAPTER 5

Siloz: Leveraging DRAM Isolation Domains to Prevent Inter-VM Rowhammer

5.1 Introduction

Cloud providers host virtual machines (VMs) from multiple tenants atop the same physical machine, while providing per-VM isolation across various metrics [77, 251]. To provide per-VM performance isolation, providers use a rich set of technologies across hardware resources (e.g., CPU affinity [187], SR-IOV [67], and memory bandwidth allocation [118]). Although providers can also use a growing set of methods to provide per-VM security isolation (e.g., CPU enclaves [57], cache partitioning [155], and memory encryption [8, 117]), providers lack practical means to provide strong isolation in one of the most significant cloud server resources: DRAM.

In particular, today's servers interleave (spread) data from multiple tenants across different DRAM banks, ranks, and channels to maximize the memory-level parallelism afforded by these structures [192,286]. Unfortunately, sharing these structures without careful consideration exposes co-located VMs to security and reliability threats [237,271], including Rowhammer bit flips [152]. To combat these threats, I envision a future in which cloud providers can leverage *DRAM isolation domains* that provide DRAM isolation capabilities in line with those of other hardware resources.

The goal of this work is to enable cloud providers to take the first step toward practically managing DRAM as a set of isolated domains. To achieve this goal, I propose the use of DRAM *subarrays* for isolation. DRAM consists of many subarrays that are natural isolation boundaries of DRAM cells [153]: cells in one subarray cannot disturb cells in another [52]. While DRAM does not expose subarrays today, software can easily determine subarray boundaries (§5.4.1). By using subarrays, I show that inter-VM Rowhammer can be prevented on *today's* cloud servers without sacrificing performance.

5.1.1 This Paper: Mitigating Inter-VM Rowhammer

Inter-VM Rowhammer is a glaring example of today's lack of DRAM isolation; a VM's frequent activations ($\approx$ accesses) of the same DRAM rows—"hammering"—can flip bits in nearby rows used by another VM or the host. Bit flips can cause data loss [152], machine check exceptions [53], denial-of-service [124], side channels [50, 167], and system subversion [269].

Despite deployed hardware mitigations [53, 75, 125, 160], cloud systems remain vulnerable to inter-VM hammering. In fact, recent work [185] shows that malicious and *commodity* cloud workloads already activate rows at rates exceeding today's Rowhammer thresholds. As these thresholds continue to decrease with process scaling [52, 147, 261], Rowhammer poses an *increasing* threat to security and reliability.

While cloud providers could use software to mitigate inter-VM hammering, state-of-the-art defenses incur high memory/performance overhead or contain significant gaps in protection. SoftTRR [347] and CTA [322] do not scalably-generalize beyond page table protection, leaving other data vulnerable. ANVIL [13] is susceptible to DMA-based Rowhammer [299]. "Guard row" mitigations [27, 31, 161]—where a set of guard rows are reserved as protection buffers between normal rows—require $\geq 50\%$ extra DRAM per protected region and thus only scale to protect small quantities of data.

Given the limitations of existing software Rowhammer mitigations—coupled with the dearth of hardware DRAM isolation support—cloud providers lack practical means to mitigate inter-VM Rowhammer. Thus, I introduce Siloz, a hypervisor that uses *subarray group* memory management to prevent inter-VM hammering with negligible performance effect. Specifically, Siloz integrates subarray group isolation, bank-level parallelism, and extended page table (EPT) integrity for efficient protection against inter-VM hammering.

Siloz's key insight is that subarray-based Rowhammer isolation—where prior work [52] shows that Rowhammer is ineffective across subarray boundaries—can co-exist with *bank-level parallelism*. Bank-level parallelism is the finest-grained access parallelism exposed by modern DRAM, offering $> 18\%$ execution time improvement [286]. As such, Siloz enables high performance alongside per-VM Rowhammer isolation by partitioning DRAM into fine-grained *subarray groups* of $\approx$ 1.5 GB each (depending on memory geometry), formed from a subarray per each of a socket's banks. Thus, a VM using one or more subarray groups can allocate memory across every bank, yet isolated to specific subarrays.

To conveniently manage subarray groups, Siloz builds on existing non-uniform memory access (NUMA) support. Siloz abstracts subarray groups as *logical* NUMA nodes, enabling robust memory management, while maintaining compatibility with *physical* NUMA performance optimizations (e.g., Siloz can use same-socket subarray groups for lower latency).

Notably, Siloz's ability to enforce subarray group isolation relies on EPT integrity; because

EPTs uniquely define the host physical addresses that VMs can access, a malicious VM could induce bit flips in even its *own* EPTs to access another domain [269]. While emerging Intel and AMD hardware offer support for EPT integrity checks [8, 117], Siloz can also protect against EPT bit flips on legacy systems. Namely, Siloz exploits the insight that all EPTs can fit in $< 0.001\%$ of DRAM rows are thus amenable to supplemental guard row protection without significant cost. By novelly accounting for server DRAM addressing alongside prior guard row techniques [27, 31, 161], Siloz limits DRAM overheads for EPT protection to just 32 8 KB rows per bank ($\approx 0.024\%$ of a 1 GB bank).

I evaluate Siloz's Linux/KVM [157] implementation on Intel Skylake servers based on a major cloud provider's configuration, demonstrating that Siloz prevents inter-VM hammering and EPT bit flips. I find that Siloz's combination of subarray group isolation and EPT protection has negligible effect on average performance (within $\pm 0.5\%$ of baseline Linux/KVM) across various cloud workloads [36, 55, 133, 162, 230], SPEC CPU 2017 [32], and PARSEC 3.0 [23, 344].

In summary, I make the following contributions:

- I present Siloz, a hypervisor that uses *subarray groups* to bring per-VM, in-DRAM isolation to the cloud, while preserving bank-level parallelism for high performance.
- To maintain isolation on hardware without emerging EPT integrity checks, Siloz places EPTs in designated guard-protected rows, using knowledge of DRAM addressing to securely limit reserved DRAM to $\approx 0.024\%$ of each bank.
- I show that Siloz offers cloud providers the first practical and comprehensive mitigation for inter-VM hammering, providing complete protection with negligible effect on average performance (within $\pm 0.5\%$ of baseline Linux/KVM).

Siloz's Linux/KVM implementation is open-source [182].

5.2 Background

In this section, I present background on server systems, DRAM, and Rowhammer as needed to understand Siloz.

5.2.1 Hypervisor Memory Management

State-of-the-art hypervisors like Linux/KVM [157] use hardware virtualization extensions (e.g., Intel VT-x [295] or AMD-V [136]) to map host memory to a VM's address space. The VM can then access the vast majority of its memory without performance-costly traps (VM exits) into the hypervisor.

The key difference between OS-level and hypervisor-level memory management is the extra layer of memory address translations included in hypervisor-provisioned mappings. For a standard process, the OS sets up multi-level page tables to translate between two types of addresses: virtual and physical. Hypervisors instead consider *three* types of addresses for a VM: (1) guest virtual addresses (GVAs, equivalent to standard virtual addresses), (2) guest physical addresses (GPAs, the VM's illusion of physical addresses), and (3) host physical addresses (HPAs, equivalent to standard physical addresses).

The guest OS uses multi-level page tables to translate GVAs to GPAs. Per convention on modern 64-bit systems, page tables are four levels for 4 KB pages, three levels for 2 MB pages, and two levels for 1 GB pages. Each level of the page table is itself implemented on a 4 KB page that holds 512 64-bit entries, each of which points to the next level of the table (or the final physical page) for a given mapping.

The hypervisor (i.e., host OS) uses similar, multi-level *extended page tables* (EPTs) to translate GPAs to HPAs. Guests cannot directly access EPTs, but guest behavior may indirectly cause accesses (e.g., hardware EPT walks).

5.2.2 Non-Uniform Memory Access (NUMA)

Cloud providers deploy large quantities of compute and memory per server for cost effectiveness and ease of management. To scale performance amidst large resource quantities, servers are often architected as non-uniform memory access (NUMA). A *NUMA node* conventionally refers to a combination of cores (e.g., a socket) and a local (near) memory pool that is faster to access than remote (far) memory; technically, a node may consist of only cores, only memory, or both.

The NUMA topology (i.e., the map of per-node resources) is typically reported to system software by server firmware [294]. System software uses the topology to improve performance via NUMA-aware resource management. For example, software may allocate memory from a core's local pool.

NUMA's key benefits are its abilities to decrease latencies for workloads using local memory and to reduce interference (e.g., memory traffic) among independent tasks on different nodes. Additionally, kernel NUMA support offers convenient abstractions to manage compute and/or memory resources.

5.2.3 Server DRAM (Micro)architecture

A server DRAM module is a hierarchically-organized set of DRAM cells, each of which encodes a single bit of information via high/low charge. Each module is typically attached to one of a set of CPU sockets, with the socket and its modules forming a conventional NUMA node (§5.2.2).

Because DRAM cell charges diminish over time, memory controllers and DRAM modules cooperate to periodically *refresh* the charges for data retention. In widely-deployed DDR4 [126] DRAM, cells are refreshed within 64 milliseconds.

As shown in Fig. 5.1, the DRAM module hierarchy is a set of *ranks*, each encompassing *banks*, each encompassing *subarrays*, which are each a row-column grid of cells. The DDR4 standard specifies that a rank holds up to 16 banks, and a row holds up to 8 KB of cells. Internal to server DRAM modules, each of a subarray's 8 KB rows is split into two half-rows across a rank's "A" and "B" sides, with each half-row simultaneously serving half of a given data request [51]. While a row's external representation (i.e., a single 8 KB structure) is sufficient to understand the majority of Siloz's design, I discuss the relevance of internal half-rows in §5.6.

Although many other (micro)architectural details are vendor-specific, a common server DRAM module is a dual-rank, 32 GB DIMM (dual in-line memory module). Given 32 GB split across 2 ranks and 16 banks/rank, a bank is 1 GB. Each bank is further divided among a vendor-specific number of subarrays [43, 52, 153, 189, 308], typically each consisting of 512–2048 rows [308]. For example, a commodity subarray size of 1024 8 KB rows [44, 145, 146, 153, 170] (as is the case for Siloz's evaluation server) yields 128 subarrays per 1 GB bank.

5.2.4 Accessing Data in DRAM

To read/write data in DRAM, a memory controller first translates the data's host physical address to a *media address* that identifies specific DRAM cells. The parallel in the CPU realm is a virtual-to-physical address mapping, which system software typically controls at page-sized granularity (§5.2.1). However, unlike software-defined virtual-to-physical mappings, physical-to-media mappings are fixed at boot via BIOS settings [103, 119] and applied at cache line granularity.

Given the data's media address, the controller first issues an *activate* (ACT) to the row containing the data. This command connects the row to its encompassing bank's *row buffer*, which can only be occupied by one row per bank at a time. The controller then issues a read or write command to an offset within the row buffer, completing the data access.

While accesses to a single bank are serialized, *different* banks can be accessed in parallel. Thus, commodity physical-to-media address mappings maximize throughput by interleaving (spreading) sequential cache lines across a conventional NUMA node's (e.g., socket's) banks, achieving *bank-level parallelism* for common access patterns [183, 286, 345].

5.2.5 Rowhammer

Rowhammer [152] is a silicon-level effect in DRAM where frequent ACTs (§5.2.4) of the same *aggressor* rows can flip bits in nearby *victim* rows due to electromagnetic interference. Specif-

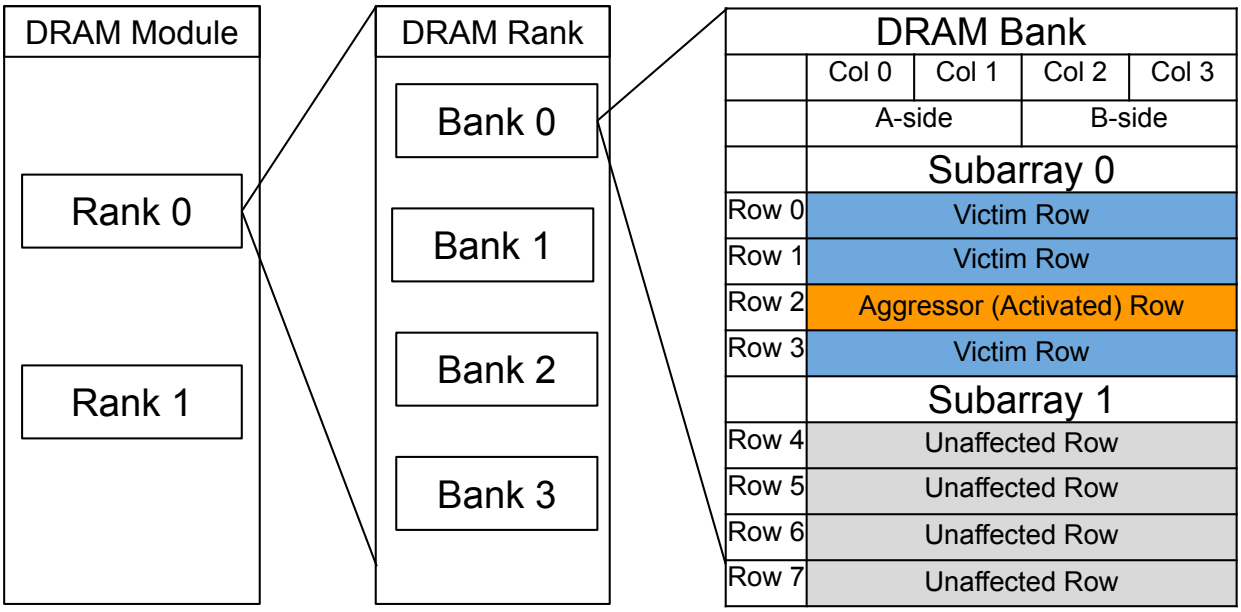

Figure 5.1: A simplified DRAM module hierarchy (§5.2.3) in the context of a DRAM row activation (§5.2.4) and Rowhammer (§5.2.5). A frequently-activated ("hammering") *aggressor* row may flip bits in *victim* rows in the same subarray.

ically, an ACT yields the side effects of (a) refreshing the charges in the activated row, but (b) potentially disturbing charges in *nearby* rows. When aggressor rows are activated at rates exceeding a Rowhammer threshold (varying across DIMMs), cumulative disturbance effects may flip bits in victim rows that have not been recently-refreshed/activated. As shown in Fig. 5.1, rows in the same subarray (§5.2.3) as the aggressor row(s) are potential victims, while rows in different subarrays are unaffected due to electric isolation (i.e., exclusive circuitry) [43, 52, 183, 330]; broadly, subarrays provide isolation boundaries for ACT-induced DRAM disturbance effects, including RowPress [190]. I limit discussion to Rowhammer for simplicity.

Rowhammer bit flips can cause data loss [152], machine check exceptions [53], denial-of-service [124], side channels [50, 167], and system subversion [269]. Recent work [185] shows that malicious and *commodity* workloads can yield ACT rates surpassing modern Rowhammer thresholds; other work [52, 147, 261] shows that thresholds are decreasing with process node scaling (i.e., susceptibility is increasing).

To mitigate Rowhammer, modern servers rely on error correction codes (ECC [19]) and target row refresh (TRR [75], an in-DRAM mitigation that refreshes a *subset* of victim rows ahead of schedule). While these mitigations have thus far proved effective for commodity workloads, carefully-crafted malicious workloads can still induce uncorrected bit flips despite ECC [53] and TRR [59, 75, 125, 160]. Furthermore, even *corrected* bit flips are a security concern, as they form a timing side channel that can leak a row's bit values to attackers with access to additional rows in

the same subarray [167].

Given the gaps in hardware mitigations and the goal of per-VM isolation, cloud providers can use software to supplementally mitigate *inter-VM hammering*, where one VM's hammering can flip bits in another VM or the host.

5.3 Limitations of Existing Software Defenses

In this section, I motivate the need for Siloz's prevention of inter-VM hammering by explaining how existing software mitigations fail to mitigate such hammering without significant performance/memory overheads, if at all. Broadly-speaking, existing software Rowhammer mitigations adopt one of three approaches: selectively-protecting data, detecting attacks in progress, or inserting guard row barriers.

Mitigation via Selective Data Protection. The first class of software mitigations opts to only protect a subset of data as a security-performance trade-off [322,347]. For instance, SoftTRR [347] periodically sets reserved bits in page table entries for neighbors of potential victim rows. Thus, accesses to the neighbors (potential aggressor rows) will trap into system software, which can refresh the victim rows before the aggressors surpass the Rowhammer threshold.

The key limitation of these defenses is that they only protect a small portion of a VM's data (e.g., page tables) for acceptable performance overheads. As I will show, Siloz protects *all* of a VM's data against another VM's hammering.

Mitigation via Attack Detection. The second class of software mitigations aims to detect Rowhammer against any data in the system and correspondingly stop the attack (e.g., by rate-limiting aggressor row activations or refreshing victim rows) [13,41,63]. For instance, ANVIL [13] uses performance counters to detect anomalies (e.g., frequent cache misses) that are potentially-indicative of a Rowhammer attack.

The key limitation of these approaches is that they can incur both performance-costly false positives and security-costly false negatives. For example, ANVIL must engage defenses at (increasingly-low) conservative thresholds, risks not detecting attacks/engaging defenses until after bits have flipped, and does not detect DMA-based Rowhammer [299]. In contrast, Siloz's subarray groups do not track hammering, but rather isolate each VM from the hammering of another; correspondingly, Siloz is agnostic to both the Rowhammer threshold and the form of hammering used to flip bits.

Mitigation via Guard Rows. Several proposed software Rowhammer mitigations [27, 31, 161] place *guard rows* between isolation domains (e.g., user-kernel or different processes). These mitigations exploit the fact that Rowhammer only affects data in nearby rows (§5.2.5), reserving guard rows as protection barriers between "normal" rows. If hammering occurs in the normal rows,

it can only flip bits in guard rows, which are unused or contain supplementally-protected data.

The key limitation of these mitigations is that they inherently waste DRAM; the guard rows cannot be used as normal rows. While I will show (§5.5.4) that these DRAM overheads can be acceptable when protecting small quantities of isolation-enforcing data against bit flips, protecting arbitrary data incurs impractical overheads (i.e., $\geq 50\%$ extra DRAM per protected region [161], where DRAM is the dominant hardware cost in many cloud environments [313]).

Furthermore, because guard rows still share circuitry with normal rows, increasing Rowhammer susceptibility requires increasing quantities of guard rows for mitigation. For instance, ZebRAM's [161] 50% DRAM overhead at 1 guard row per normal row rises to 80% at a modern requirement of 4 guard rows per normal row on server DIMMs [52, 261].

Finally, in addition to wasting DRAM, existing guard row mitigations do not account for standardized [126] and vendor-specific [52] server DRAM address mappings that affect row proximity (and hence, which rows must serve as guard rows).

As I will show (§5.6), Siloz use of subarray groups to prevent inter-VM hammering (a) allows $\approx 98.5\%$–100% of DRAM to be used as normal rows, depending on server features and subarray size (b) offers fundamental, silicon-level Rowhammer isolation, and (c) accounts for server DRAM addressing in both normal rows and any potential guard rows.

Key Takeaways. Existing software mitigations fail to provide efficient and effective mitigation of inter-VM hammering to cloud providers. As I will show in contrast, Siloz's subarray group isolation comprehensively protects each domain against inter-VM hammering at negligible overhead.

5.4 Subarray Group Primitive

In this section, I introduce the *subarray group* that Siloz uses as a memory management primitive. Recall that each DRAM bank is composed of a set of row-column subarrays, where Rowhammer is ineffective across subarray boundaries (§5.2.5). Accordingly, the key motivation behind subarray groups is that different VMs occupying disjoint subarray(s) cannot directly hammer each other. I first describe the structure of subarray groups in DRAM (§5.4.1) before detailing how system-level pages map to subarray groups (§5.4.2).

5.4.1 Subarray Groups in DRAM

While each individual subarray offers a unit of Rowhammer isolation, Siloz opts to provide isolation via *subarray groups*, defining subarray group s as a collection of the s^{th} subarray from

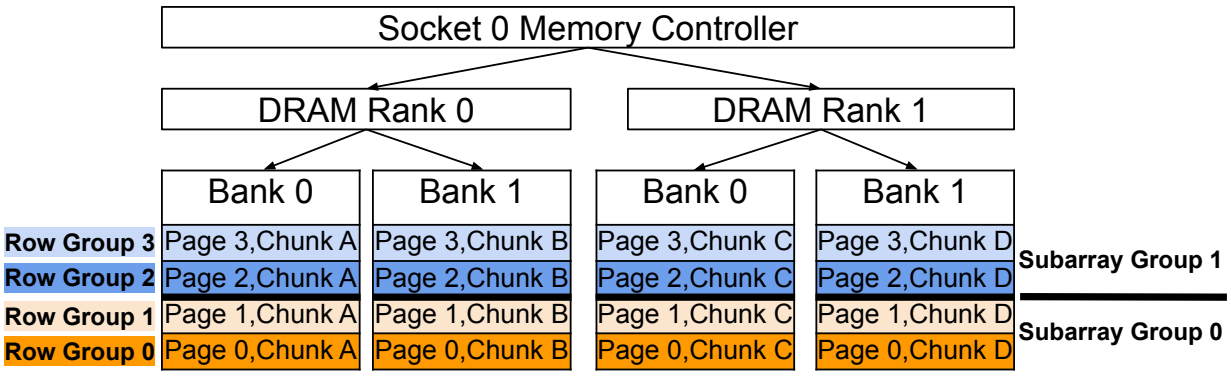

Figure 5.2: *Subarray groups* in a DRAM hierarchy (§5.4.1). Ascending physical pages are mapped to ascending *row groups*—and by extension, subarray groups—in a physical node (§5.4.2). For simplicity, I depict 2 rows per subarray, 1 page per row group, and a monotonically-ascending mapping.

each bank in a physical[1] NUMA node (e.g., socket). As motivation behind Siloz's use of subarray groups, I consider the challenges of isolation to a single subarray. In particular, allocating a page of memory on a single subarray on modern servers is not practical—if even feasible—due to (a) physical-to-media address mappings that interleave individual pages across a physical node's banks to achieve bank-level parallelism (§5.2.4), and (b) the performance impact of eliminating such parallelism (e.g., $> 18\%$ for some workloads [286]), if such an option is supported in BIOS/-firmware.

Overcoming these challenges, Siloz' subarray groups' composition from 1 subarray per bank in a physical node maintains high throughput and is compatible with physical-to-media address mappings. As depicted in Fig. 5.2, for a subarray size of r rows, subarray group 0 is comprised of rows $[0, r)$ in each of a physical node's banks (i.e., *row groups* $[0, r)$), subarray group 1 of row groups $[r, 2r)$, and so on. While I have not observed heterogeneously-sized subarrays in Siloz's evaluation platform, subarray group composition can be trivially-adjusted to account for proposed heterogeneity [43, 170, 280, 308] and heterogeneity observed in other DRAM modules [221].

A subarray group's size is thus the product of a server's number of banks per physical node, rows per subarray, and row size. Of these factors, banks per physical node and row size are already reported to system software. While microarchitectural subarray sizes are *not* reported in the DDR4 standard [126], I have confirmed with a major cloud provider that DRAM vendors can share subarray sizes with them.

Even without the cooperation of DRAM vendors, one can infer a module's subarray sizes using prior methodologies [52, 329]. I apply the mFIT [52] methodology—which uses a regular pattern

[1]We refer to conventional NUMA nodes (§5.2.2) as *physical* nodes to distinguish them from Siloz's *logical* nodes (§5.5.2).

Banks/Physical Node	Rows/Subarray	Row Size	SG Size
192	1024	8 KB	1.5 GB

Table 5.1: Subarray group (SG) size is the product of banks per physical node, rows per subarray, and row size; SG size derivation for Siloz's evaluation server is pictured.

of failed Rowhammer attacks to deduce subarray size—to Siloz's evaluation server and observe failed Rowhammer attacks every 1024 rows. Thus, I infer a subarray size of 1024 rows, consistent with sizing used in prior work [44, 145, 146, 153, 170].

Given the server's 192 banks/physical node and 8 KB/row, the 1024-row subarray size yields a subarray *group* size of 1.5 GB, as shown in Table 5.1 (192 banks/physical node $*$ 1024 rows/subarray $*$ 8 KB/row). For subarray sizes in the modern range of 512–2048 rows [308], the subarray group size would linearly-increase from 0.75 GB to 3 GB. I compare the effects of managing different subarray group sizes in §5.7.4.

5.4.2 Mapping Pages to Subarray Groups

Subarray group isolation can only work if entire *pages* map to the same subarray group(s). This is because hypervisors—including Siloz—provision memory to VMs at the granularity of pages (§5.2.1), meaning that a VM is only isolated if its pages reside in the same exclusive subarray group(s).

We therefore detail how commodity physical-to-media addressing maps all 2 MB and 4 KB pages to a single subarray group, enabling isolation. I then discuss isolation of 1 GB pages, which poses an additional challenge.

2 MB and 4 KB Pages. Given 2 MB alignment in commodity subarray group sizes (with handling of exceptional cases discussed in §5.6), I exploit the insight that 2 MB and 4 KB pages map to a single subarray group on servers that adopt a generally-ascending physical-to-media address mapping. For instance, to a first approximation of Intel's Skylake-based server mappings, row groups (§5.4.1) are populated in ascending order by ascending page numbers, shown in Fig. 5.2. Assuming one page per row group for visualization, page 0 maps to row group 0, page 1 to row group 1, and so on.

Considering the finer details of Intel's mapping, increasing page numbers do not *monotonically*-ascend through all row groups, but still result in a layout where 2 MB and 4 KB pages map to the same subarray group (maintaining subarray group isolation capabilities). Specifically, every n rows groups are populated in *alternating ascending* fashion by two individually-contiguous physical address ranges A and B, where $n = 16$ based on the memory geometry of Siloz's evaluation server (and 16 row groups is 24 MB of memory: 8 KB/row $*$ 16 rows/bank $*$ 192 banks/socket).

Row groups $[0, n)$ are populated by the first chunk of range A, row groups $[n, 2n)$ by the first chunk of range B, row groups $[2n, 3n)$ by the second chunk of range A, and so on—until repeating with new ranges at a second, 768 MB-aligned mapping "jump". Crucially, because these chunks align with and encompass entire 2 MB pages, subarray group isolation remains possible.

1 GB Pages. The aforementioned address "jump" at 768 MB-aligned addresses means that 1 GB (1024 MB) pages do *not* inherently map to a single subarray group. However, by constructing *sets* of consecutive subarray groups totaling 3 GB in size (e.g., 2 sets of 1024-row subarray groups, each 1.5 GB), I find that 1/3 of 1 GB physical address ranges occupy the same set of subarray groups, enabling isolation of associated pages. The remaining 2/3 of memory can be allocated as 2 MB or smaller pages to preserve isolation.

5.5 Siloz Hypervisor Design

In this section, I present the design of a hypervisor, Siloz, built to provide efficient inter-VM Rowhammer protection by placing VMs in private subarray group(s). We first detail Siloz's policy for inter-VM isolation via subarray groups (§5.5.1) and how Siloz introduces *logical* NUMA nodes to manage this policy (§5.5.2). I then describe a subarray group's lifetime from host boot to shutdown (§5.5.3). Finally, I discuss integrity protection for the extended page tables (EPTs) that Siloz uses to enforce subarray group isolation (§5.5.4).

For convenient concrete examples, I discuss Siloz and its subarray groups in the context of a Linux/KVM baseline hypervisor, Siloz's evaluation server—a dual-socket, 192 DRAM banks/-socket (i.e., physical node), major cloud provider-based Intel Skylake configuration—-and a commodity subarray size of 1024 rows [44, 145, 146, 153, 170] (as found on Siloz's evaluation server, resulting in a subarray group size of 1.5 GB, §5.4). However, Siloz's design principles generalize to other hypervisors, memory geometries, subarray sizes, and—given similar physical-to-media address mappings—CPU vendors.

5.5.1 Subarray Group Isolation: Goal and Policy

Siloz's goal is to isolate each VM and the host from inter-VM hammering. Accordingly, Siloz places each VM and the host into private subarray groups, such that the effects of hammering are restricted to one's own domain. Siloz correspondingly classifies each subarray group as either *host-reserved* (usable by the host) or *guest-reserved* (usable by exactly one VM).

Siloz decides whether to allocate a page from a particular host- or guest-reserved subarray group based on the page's *mediated* or *unmediated* classification as henceforth defined. If a VM can directly access the page (e.g., without a VM exit), the page is unmediated and should be

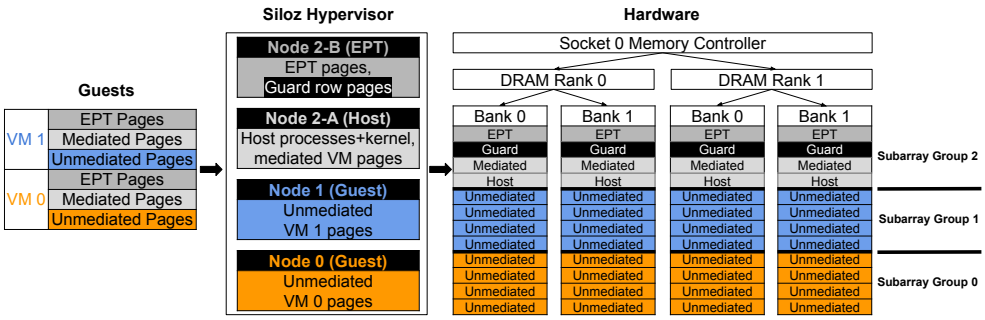

Figure 5.3: Siloz prevents inter-VM hammering by placing specific pages in host- or guest-reserved subarray groups, based on whether a VM has *unmediated* access to the pages (§5.5.1). Siloz abstracts subarray groups as logical NUMA nodes (§5.5.2) for convenient memory management throughout system lifetime (§5.5.3). Because extended page tables (EPTs) enforce subarray group isolation, Siloz supplementally ensures EPT integrity using emerging hardware extensions [8, 117] or guard rows (§5.5.4).

allocated from one of the VM's subarray groups; otherwise, the page is *mediated* and should be allocated from a host-reserved subarray group.

Intuitively, *unmediated* pages include those mapped into the VM's address space that will not cause a VM exit for some access type (e.g., guest RAM, guest ROM due to unmediated reads, and select MMIO pages). More specifically, Siloz classifies pages based on their existing QEMU memory type [62], indicating which access types trigger exits, if any.

The rationale behind Siloz's policy is that a VM can trivially-hammer memory to which it has unmediated access, and that memory should therefore be contained to the VM's subarray group(s) to maintain isolation. Conversely, theoretical "confused deputy" hammering (i.e., maliciously-exiting into the hypervisor in a manner that tricks the host into hammering on behalf of the VM) is a comparatively-difficult—and undemonstrated—attack vector. Thus, host-mediated pages are already relatively-guarded against use for Rowhammer. More importantly, should such confused deputy hammering ever prove feasible, the required VM exit means that the host could easily apply its own mitigation for this hammering (e.g., rate-limiting exit-induced memory accesses).

For clarity, I defer discussing Siloz's handling of EPT pages—which are uniquely both (a) not mapped into the guest's address space but (b) still indirectly-accessible to the guest via unmediated hardware EPT walks (§5.2.1)—to §5.5.4.

5.5.2 Subarray Groups as Logical NUMA Nodes

To conveniently manage subarray group isolation, Siloz introduces the concept of *logical* NUMA nodes, where a logical NUMA node consists of at least one subarray group. Siloz thus builds on robust and mature kernel support for *physical* nodes (e.g., sockets, §5.2.2) to manage subarray

groups.

While logical nodes are managed via similar NUMA primitives, they are distinct from physical nodes; when two logical nodes contain DRAM from the same socket—namely, two different subarray groups—they do *not* exhibit remote NUMA latencies between each other. Siloz nonetheless preserves physical NUMA optimizations via a mapping from each logical node to its physical node. For instance, when possible, a VM is constructed from subarray groups (logical nodes) co-located on same socket (physical node) for lower latency.

Logical nodes corresponding to guest-reserved subarray groups are classified as guest-reserved nodes. Guest-reserved nodes comprise all but one logical node per socket and are *memory-only* (i.e., no directly-associated compute resources, §5.2.2). This design, coupled with a Linux *control group* [68] that limits memory allocations to specific nodes [138], prevents Siloz from using guest-reserved nodes unless requested by a KVM-privileged process (§5.5.3). Such memory-only nodes are similar to the concept of zNUMA nodes [176].

The remaining logical nodes correspond to host-reserved subarray groups and are hence classified as host-reserved nodes. Unlike guest-reserved nodes, host-reserved nodes are associated with both subarray group(s) *and* their corresponding socket's cores. Again coupled with a Linux control group, this design restricts Siloz to host-reserved nodes by default for both memory allocations and scheduling decisions.

5.5.3 Lifetime of a Subarray Group

Siloz calculates which physical pages map to which subarray groups during early boot, enabling isolation from boot until shutdown. The number of rows per subarray is passed as a boot parameter. To determine the physical-to-media address mapping (required to map physical addresses to subarray groups), Siloz uses its ports of existing drivers [102, 119] for such translations, modified to operate during early boot. Because physical-to-media mappings are fixed based on BIOS settings (§5.2.4), the calculated subarray group address ranges can be cached across boots in a bootloader or firmware.

Once the subarray group address ranges are loaded, Siloz augments existing NUMA topology parsing logic (§5.2.2) to (a) provision a logical node for each subarray group, and (b) store a mapping from the logical node to its corresponding physical node to preserve physical NUMA semantics (§5.5.2).

After the required nodes are in place and boot is complete, a privileged user can create a control group with exclusive access to available guest-reserved nodes. A QEMU [20] process, which manages a KVM VM, can then allocate memory on the guest-reserved nodes if the process (a) belongs to the control group, and (b) has KVM privileges. To request this memory, QEMU uses a

new UNMEDIATED flag in its mmap() calls for unmediated memory ranges; recall that mediation status is provided by existing QEMU memory types (§5.5.1). Upon parsing the flag, Siloz checks whether the requesting process is permitted to access guest-reserved nodes, and if so, allocates the memory from the appropriate nodes.

During VM execution, Siloz avoids potential overheads of managing a large number of nodes by identifying scenarios in which it is unnecessary to iterate over guest-reserved nodes, especially while holding locks. For instance, a guest-reserved node's free memory statistics do not change after VM boot and thus do not require periodic updates [285].

When a VM is shutdown/killed, its backing host memory is freed to the corresponding (logical) nodes' free pools per existing Linux semantics. However, the nodes' reservation remains valid until its encompassing control group is destroyed/modified by a privileged user. I note that there is no modification to host shutdown: the privileged shutdown routine is free to kill any process and its resources, ignoring otherwise active subarray group/logical NUMA constraints.

5.5.4 Extended Page Table (EPT) Integrity

EPTs pose a unique challenge to subarray group isolation. Because EPTs define the host physical addresses that a VM may access (§5.2.1), Siloz relies on EPT integrity to enforce subarray group isolation. Thus, unlike other VM data, Siloz's goal of per-VM Rowhammer isolation requires protection of a VM's *own* EPTs, not just inter-VM isolation; EPT bit flips must be prevented or detected-upon-use (integrity-checked).

Hardware-Based Protection. Emerging Intel and AMD servers support *secure EPT* [117], referred to as secure nested paging (SNP) by AMD [8]. With secure EPT, hardware automatically performs mapping integrity checks for EPT entries denoted as "secure", providing a convenient mechanism to supplement Siloz's subarray group isolation. While integrity checks only detect—not prevent—EPT corruption, they eliminate the key security threat of EPT bit flips: software cannot use a corrupted EPT to escape subarray group isolation.

Software-Based Protection. Because secure EPT is only beginning to emerge, an alternate solution is needed for legacy hardware. Here, Siloz exploits the insight that EPTs across all VMs account for a very small fraction of DRAM (i.e., $< 0.001\%$ in deployment conditions described shortly), lending themselves to protection via guard rows (§5.3).

A strawperson solution would reserve an entire subarray group for EPTs, placing n guard rows between each EPT row. Given 1024-row subarrays, EPTs and guard rows would jointly-occupy $\approx 0.78\%$ of DRAM. However, via insights about server DRAM addressing and VM deployment, Siloz can significantly reduce even this small DRAM overhead.

In particular, all EPTs can fit into a *single* row group per socket on Siloz's major cloud provider-

based server configuration due to several deployment conditions. First, because cloud providers do not typically share pages among VMs for security [176, 282, 305], the number of EPTs is bounded; each host page is mapped in at most one EPT. Second, allocating VMs in contiguous physical memory regions—made feasible by (a) each subarray group's contiguity, and (b) static guest memory allocation done for performance [176, 282, 305]—further reduces the number of EPTs; each last-level EPT can map 512 of the VM's contiguous pages (§5.2.1). Third, backing guests with 2 MB huge pages—again for performance [176, 282, 305]—reduces EPTs by a factor of 512 (§5.2.1).

In this environment (deployed by multiple major cloud providers [176, 282, 305]), the 512 entries in each last-level EPT page cumulatively map approximately 1 GB of DRAM, with higher-level EPT pages providing a negligible ($\approx 1/512$) increase in the total number of EPT pages. Since each bank is 1 GB, and a single 8 KB row in a bank holds two EPT pages, one row group per socket is sufficient to store all EPTs.

Thus, rather than allocating an entire subarray group for EPTs, Siloz reserves a contiguous block of b row groups in a designated subarray group. One row group at offset o in the block serves as the EPT row group, while the other $b - 1$ row groups serve as guard rows (roughly split above and below the EPT row group). The host or a VM can accordingly safely use remaining (non-reserved) rows in the subarray group.

In our implementation, Siloz specifically uses $b = 32$ and $o = 12$, which reserves just $\approx 0.024\%$ of DRAM for the combination of EPTs and guard rows. At a high level, the specific choices of $b = 32$ and $o = 12$ ensure that an EPT row has a sufficient number on guard rows on both sides to prevent bit flips, in spite of potential DIMM-internal half-row (§4.2.1) remaps affecting adjacency within 32-aligned blocks. I defer more detailed discussion of such remaps to §5.6.

To allocate EPTs from appropriate row groups, Siloz instruments the host KVM module's kmalloc() calls for EPT pages with a new GFP_EPT flag (get free page EPT). Siloz uses this flag in conjunction with the corresponding VM's control group to choose a row group block (implemented, like a subarray group, as a logical NUMA node) for the allocation.

To prevent guard rows from being used, Siloz offlines pages mapping to the guard rows during system initialization. I note that this behavior is simply an extension of Linux's existing capability to offline faulty memory pages [37].

5.6 Handling Media-to-Internal Mappings

Thus far, I have discussed DRAM row addressing (and by extension, subarray group isolation) in the context of the media addresses with which memory controllers access DRAM (§5.2.4). However, for completeness, it is important to consider potential differences in a server DIMM's

internal mapping of these media addresses, such that rows are indeed isolated to the expected subarrays. Thus, unlike prior software mitigations [13,27,31,63,161,299,322,347], Siloz accounts for various sources of potential row *remaps* in server DIMMs.

Row Repairs. DRAM vendors and cloud vendors can "repair" defective rows by remapping them to spare internal rows that are allocated during manufacturing [9, 49, 107, 108, 127, 129, 148, 148]. Notably, a row's remapped internal address is left up to the DRAM vendor and not exposed to the memory controller, which continues to use the same media address. Such repairs pose a threat to subarray group isolation if they are *inter*-subarray, wherein a defective row could be remapped to a spare row in a *different* subarray group [137].

While our experimental results (§5.7.1) have not yielded evidence of inter-subarray row repairs (e.g., many/all defective rows may be repaired using *intra*-subarray spare rows), Siloz can still mitigate the threat of inter-subarray row repairs. In the worst-case that a DIMM implementation only uses inter-subarray methods for its repairs, the pages mapping to these rows can simply be removed from the system's memory allocation pool to preserve isolation, as can already be done for failing memory [37]. I note that only a small portion of rows (e.g., 0.15% [52]) have been experimentally-observed to be remapped due to row repairs in server DIMMs, meaning little memory capacity would be lost with such a mitigation.

Vendor-Specific Address Scrambling. A subset of major DRAM vendors perform *row address scrambling* [52], transforming bits b_1 and b_2 of the row media address (where b_0 is the least significant bit) by XOR-ing each with b_3. While row scrambling can thus affect the internal *ordering* of a group of 8 rows (bit range $[b_0, b_2]$ encodes 8 rows, where b_1 and b_2 are potentially transformed), it does not affect the internal *contiguity* of these 8 rows; higher-order bits are unchanged.

Thus, for any DIMM whose subarray size is a multiple of 8 rows, there is no impact. In any remaining DIMMs, Siloz can remove pages mapping to the 8 row range potentially violating isolation at each subarray boundary from allocatable memory (similar to any inter-subarray repaired rows). For non-multiple-of-8 subarray sizes in the range $(512, 2048)$, this would impact between $\approx 1.56\%$ and $\approx 0.39\%$ of DRAM, respectively (linearly-decreasing with larger subarray sizes).

Standardized Address Mirroring and Inversion. DDR4 specifies [128] two other forms of row address transformations for a specific subset of signals: *mirroring* and *inversion*. Because both have similar ramifications for Siloz, I first describe each transformation as depicted in Table 5.2, before detailing how Siloz accounts for them. Again given a modern subarray size of 512–2048 rows [308], I consider transformations of row address bits in the range $[b_0, b_{10}]$ (encoding up to 2048 rows). Notably, transformations of $[b_0, b_{10}]$ and higher-order bits are mutually-independent in DDR4 [128].

Address mirroring works as follows: for a multi-rank DIMM, select pairs of address bits are mirrored (swapped) in *odd* ranks (red+orange columns in Table 5.2), while unmodified in even

Bit	Even Rank		Odd Rank	
	A-side	B-side	A-side	B-side
b_0	b_0	b_0	b_0	b_0
b_1	b_1	b_1	b_1	b_1
b_2	b_2	b_2	b_2	b_2
b_3	b_3	! b_3	b_4	! b_4
b_4	b_4	! b_4	b_3	! b_3
b_5	b_5	! b_5	b_6	! b_6
b_6	b_6	! b_6	b_5	! b_5
b_7	b_7	! b_7	b_8	! b_8
b_8	b_8	! b_8	b_7	! b_7
b_9	b_9	! b_9	b_9	! b_9
b_{10}	b_{10}	b_{10}	b_{10}	b_{10}

Table 5.2: DDR4 address mirroring and inversion [128] of lower-order row media address bits as a function of DIMM rank and "side" (half). Odd-rank addresses are mirrored (red+orange). B-side addresses are inverted (yellow+orange). Lightened colors denote transformed bits. "!" denotes boolean NOT.

ranks (white+yellow). Specifically, bit pairs ¡b_3, b_4¿, ¡b_5, b_6¿, and ¡b_7, b_8¿ are each mirrored on odd ranks (e.g., 0b**10**000—$b_4 = \mathbf{1}$, $b_3 = \mathbf{0}$—becomes 0b**01**000).

To understand address inversion, recall that each row is internally-split into two half-rows: the A-side and B-side half-rows (§4.2.1). Bits $[b_3, b_9]$ are inverted in B-side half-rows (yellow+orange), but not in A-side half-rows (white+red).

As with row address scrambling, inversion and mirroring pose a challenge to subarray group isolation only for certain subarray sizes. In particular, if the subarray size is a power-of-2 in the commodity range of 512–2048 rows, inversion and mirroring have no effect on subarray group isolation; for instance, the major vendor's DIMMs in Siloz's evaluation server are unaffected, given their 1024-row subarrays (§5.4).

For intuition on why power-of-2 sizes work so well, note that the n least-significant bits of a row media address encode the exact number of rows in a subarray of size 2^n. Given sizes of 512 ($n = 9, [b_0, b_8]$), 1024 ($n = 10, [b_0, b_9]$), and 2048 rows ($n = 11, [b_0, b_{10}]$), it is clear from Table 5.2 that these subarray size-aligned bit ranges are only transformed to different offsets *within* the same subarray, maintaining isolation.

For remaining potential subarray sizes in the commodity range—where inversion and mirroring can cause pages to be split across subarray boundaries—Siloz can still provide subarray group isolation by forming "artificial" subarray groups. In particular, Siloz can round the subarray size up to the nearest power-of-2, such that the rows in an artificial subarray group maintain the DIMM-internal contiguity property.

Because these artificial boundaries would not always align with true subarray boundaries that provide natural isolation, Siloz can instead enforce isolation across artificial bounds by adding n guard rows at the start of each artificial subarray, where $n = 4$ protects against bit flips observed on modern server DIMMs [52, 261]. Accounting for mappings on different ranks and sides, this

Parameter	Value
Host Machine	Dual-socket Intel Xeon Gold 6230 CPU @ 2.1 GHz; per-socket: 40 logical cores, 192 GB DDR4 DRAM (32 GB 2Rx4 DIMMs @ 2933 MHz, 192 total banks, 1024 8 KB rows per subarray)
Host OS+Kernel	Ubuntu 22.04+Linux/KVM 5.15 (generic configuration)
Guest OS+Kernel	Ubuntu 22.04+Linux 5.15 (generic configuration)

Table 5.3: Baseline system configuration. The host kernel is varied among the unmodified Linux/KVM baseline and Siloz.

would reserve between $\approx 1.56\%$ and $\approx 0.39\%$ of DRAM (again linearly-decreasing with larger subarray sizes). I note that this reservation would be *in place* of any potential reservations for address scrambling, since the artificial subarray size would be a multiple of 8.

Key Takeaways. While commodity power-of-2 subarray sizes integrate most easily, Siloz can support other potential subarray sizes by removing the small fraction of pages violating isolation from allocatable memory. Nonetheless, the aforementioned challenges highlight the benefit of hardware-software co-design in memory systems [153, 217], especially for optimal subarray group isolation. For instance, many limitations of the current DRAM hardware-software interface could be overcome by DRAM vendors directly exposing isolation domains such as subarray groups, providing architectural guarantees that would facilitate Siloz's widespread adoption.

5.7 Evaluation

We evaluate Siloz against a Linux/KVM [157] 5.15 (Ubuntu 22.04 LTS) baseline hypervisor on a major cloud provider-based Intel Skylake server configuration. Unless noted, I use default BIOS settings. Given 192 banks per socket (i.e., physical node) and 1024 8 KB rows per subarray, Siloz manages a subarray group size of $192 * 1024 * 8$ KB $= 1.5$ GB by default. I evaluate the effects of instead managing subarray sizes of 512 and 2048 rows (the lower and upper limits of modern subarray sizes [308]) in §5.7.4 and distinguish these variants as Siloz-512, Siloz-1024 (default), and Siloz-2048.

We use the same generic kernel configuration and boot parameters for the baseline and Siloz. Our system configuration is listed in Table 5.3.

To evaluate Siloz's security, I run an extended version of the Blacksmith [125] Rowhammer fuzzer to attempt bit flips.

To evaluate Siloz's effect on execution time, I run *redis+YCSB* [36, 55] and *terasort* from Hadoop [230] in line with related work [82]. I also run the SPEC CPU 2017 and PARSEC 3.0 benchmark suites used in related work [82, 161, 347].

To evaluate Siloz's effect on memory throughput, I run *memcached* [133] and SysBench *mySQL* [162], as well as Intel's Memory Latency Checker (MLC) v3.9a [306], which uses per-

Observed Bit Flips?	DIMM					
	A	B	C	D	E	F
Inside Subarray Group	yes	yes	yes	yes	yes	yes
Outside Subarray Group	NO	NO	NO	NO	NO	NO

Table 5.4: Siloz's contains bit flips to a hammering domain's subarray group (§5.7.1), preventing inter-VM hammering.

formance counters to measure throughput.

We run performance benchmarks in an unmodified Ubuntu 22.04 VM using KVM [157] acceleration with QEMU [20] v6.2.0 (i.e., Ubuntu 22.04's version). Select *mmap()* calls are modified to request memory from guest-reserved nodes (§5.5.3).

VMs are provisioned with all 40 logical cores from socket 0 and 4 GB of DRAM per logical core (160 GB total). Multi-threaded workloads are executed with a thread per logical core (40 total), except for PARSEC workloads, which require a power-of-2 number of threads and are thus executed with 32 threads. Guest memory is statically allocated, pinned, not oversubscribed, and backed by 2 MB host huge pages, as done by multiple major cloud providers [176, 282, 305].

5.7.1 Security

We assess two aspects of Siloz's security. First, I determine whether Siloz can *contain* hammering to a domain's exclusive subarray group(s)—or alternatively put, whether Siloz can eliminate inter-VM bit flips. Second, I determine whether Siloz can prevent bit flips in designated rows (i.e., EPTs rows).

We generate bit flips in the baseline system using a modified version of Blacksmith [125] Rowhammer fuzzer (i.e., a fuzzer that attempts to find hammering patterns that induce bit flips despite state-of-the-art hardware mitigations), which I have extended to support server DIMMs. I then compare Blacksmith's effectiveness when running under Siloz.

Hammering Containment. I first pin Blacksmith to a subarray group in Siloz, where I only observe bit flips in the group, as expected. I left the system running for 24 hours, such that any bit flips potentially undetected by Blacksmith would be detected by ECC patrol scrubbing (which checks every row at least once per 24 hours). While I observed bit flips in all of the socket's DIMMs (and across ranks and banks in the DIMMs), *none* of these bit flips occurred outside of the subarray group (Table 5.4). Thus, I confirm Siloz's ability to contain hammering to provisioned subarray groups.

EPT Bit Flip Prevention. To assess Siloz's ability to prevent bit flips in designated rows (e.g., EPT rows), I run Blacksmith with disjoint (a) groups of 32 consecutive logical rows protected according to Siloz's guard row-based mitigation, and (b) other groups of 32 rows unprotected in

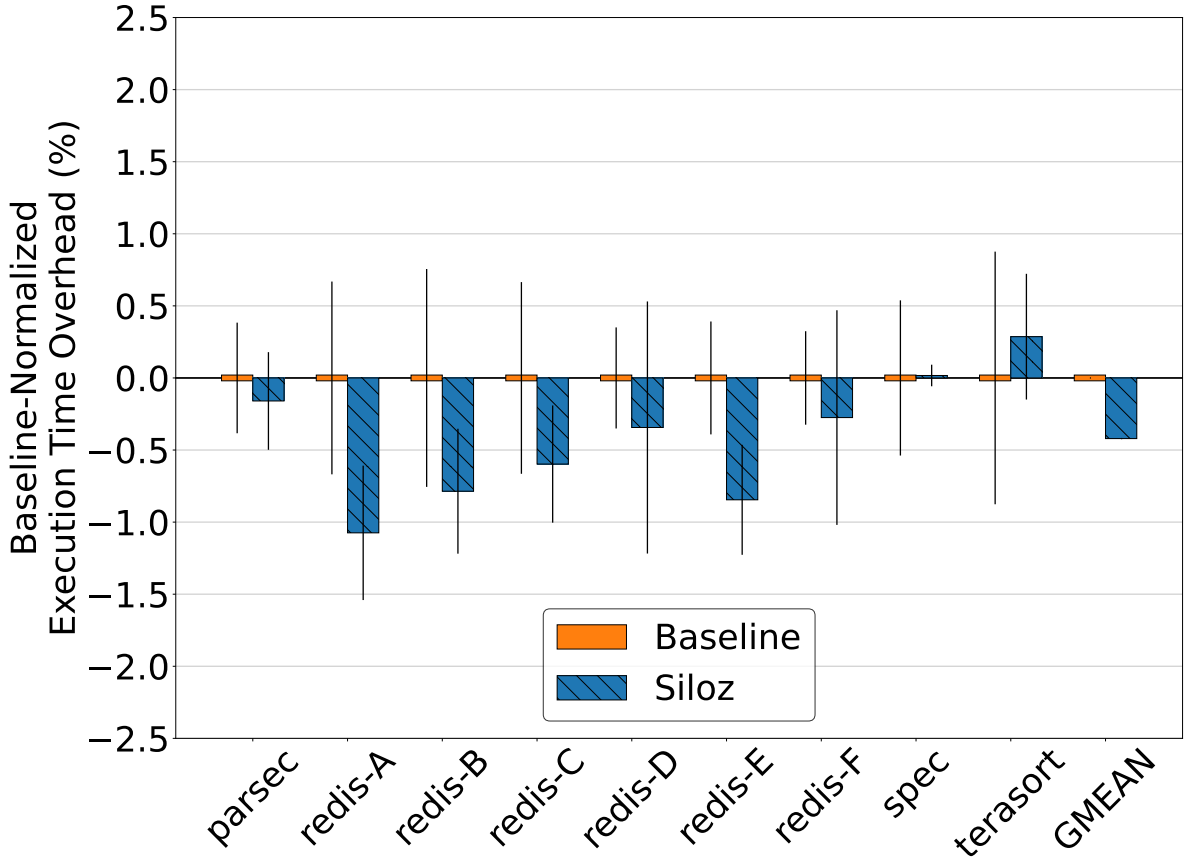

Figure 5.4: Baseline-normalized execution time (§5.7.2) for Siloz. Error bars depict 95% confidence intervals. Lower is better.

the same subarray group. As expected, I do not observe bit flips in the protected rows, while I *do* observe bit flips in the unprotected rows. I also note that all bit flips observed during our hammering containment tests were in *non*-EPT rows. Thus, I demonstrate Siloz's efficacy in *preventing* EPT bit flips.

5.7.2 Execution Time

We evaluate Siloz's effect on execution time against redis+YCSB, Hadoop terasort, and the SPEC CPU 2017 (SPECspeed) and PARSEC 3.0 benchmark suites. I include all six YCSB core workloads *A*–*F*. I omit PARSEC's supplemental network benchmarks due to occasional deadlock in the benchmarks on all kernels (i.e., including the unmodified baseline).

As shown in Fig. 5.4, I find that Siloz's geometric mean timing shows $< 0.5\%$ difference from baseline timing, demonstrating Siloz's negligible effect on execution time. Because Siloz only

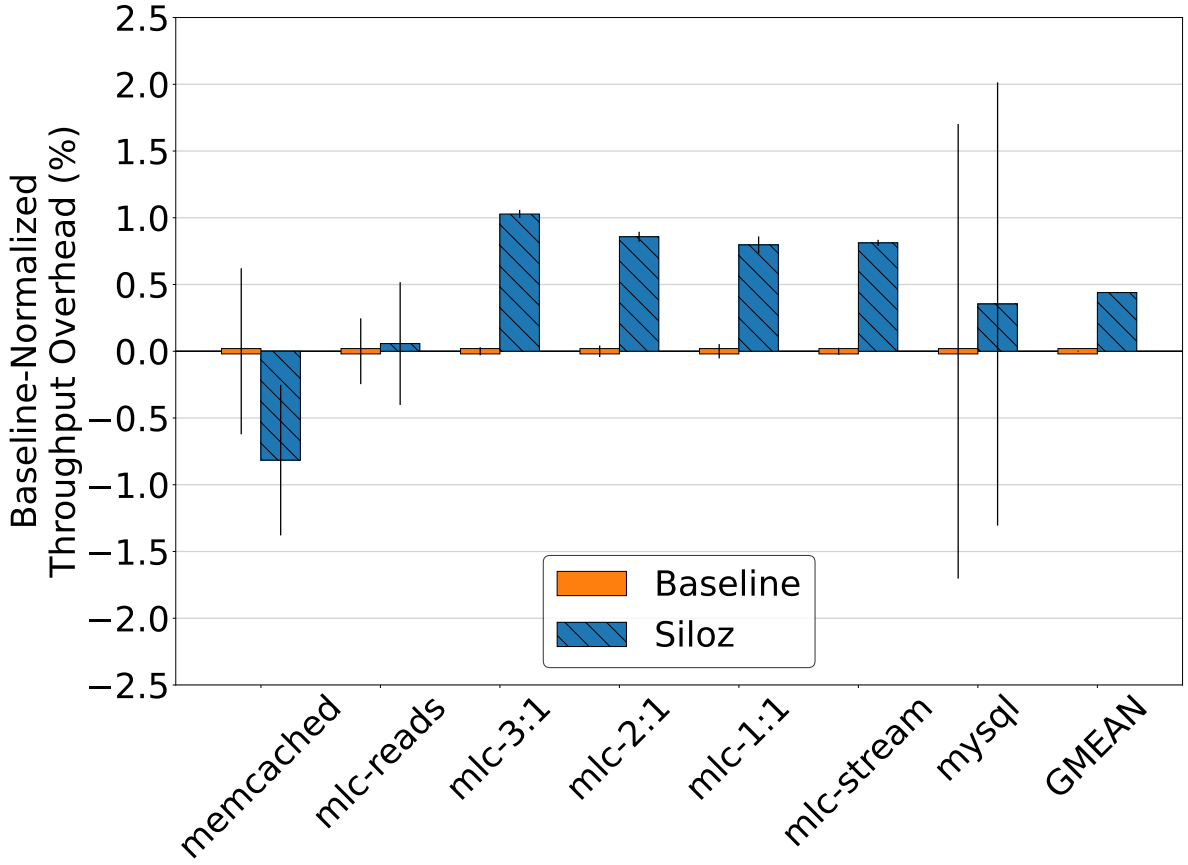

Figure 5.5: Baseline-normalized throughput (§5.7.3) for Siloz. Error bars depict 95% confidence intervals. Lower is better.

affects the location of *boot-time* allocations for each VM, one would not expect significant runtime effects, consistent with our results. Nonetheless, I consider potential sources of varied execution time for completeness.

Beyond the noise present in each benchmark, potential sources of execution time *improvement* for Siloz stem from Siloz's relatively-stringent NUMA locality enforcement. For instance, I found that when running *redis-B* in a slightly-modified baseline (specifically, with Siloz's constraints on EPT allocations, and no other changes), performance slightly improved. I note that better NUMA locality for EPTs is currently being reviewed for the Linux kernel [274].

Potential sources of execution time *worsening* for Siloz stem from Siloz's need to iterate over a greater quantity of (logical) NUMA nodes than the baseline, especially in I/O-bound workloads where the host kernel is more active. However, Siloz's subarray size sensitivity experiments (§5.7.4) indicate that noise is a more likely culprit.

5.7.3 Throughput

We measure Siloz's effects on memory throughput using memcached, Sysbench mySQL, and Intel MLC. MLC workloads are differentiated by all reads (*mlc-reads*), read:write ratios (*mlc-3:1*, *mlc-2:1*, and *mlc-1:1*), and a STREAM triad-like benchmark [204] (*mlc-stream*). As shown in Fig. 5.5, Siloz yields $< 0.5\%$ difference from baseline geometric mean throughput.

Factors affecting increases and decreases in throughput are similar to those affecting execution time (i.e., Siloz's more stringent NUMA enforcement and management of a greater number of NUMA nodes). Additionally, both bandwidth and execution time can be affected by address-dependent cache slice/set indexing functions [122, 201, 335]; specific addresses vary between the baseline and Siloz due to Siloz's subarray group address range constraints. However, because Siloz still manages contiguous regions much larger than those of commonly-optimized access patterns, it is unsurprising that there is no clear performance difference. Given that mean bandwidth and execution time are well-within the confidence intervals of individual benchmarks, I conclude that Siloz's mean differences are insignificant.

5.7.4 Subarray Size Sensitivity

While I are unaware of modern server DIMMs using the lower bound (512 rows) and upper bound (2048 rows) for conventional modern subarray sizes [308], I can nonetheless measure Siloz sensitivity to such sizes by modifying Siloz's *presumed* subarray size (passed as a boot parameter, §5.5.3).

In particular, because DDR standard access timings do not vary across subarrays [126], and varying the subarray size does not change the degree of bank-level parallelism available to each subarray group (§5.4), I can measure Siloz's performance on "artificial" subarray groups without loss of accuracy. I note that such artificial groups do *not* work for evaluating security without additional considerations (§5.6), because unlike access time and bank-level parallelism, isolation properties change across subarrays.

For clarity, I refer to the "original" Siloz variant run on our evaluation's server as Siloz-1024 (since the true subarray size is 1024 rows), and compare it to variants Siloz-512 and Siloz-2048. Notably, Siloz-512's smaller subarray group sizes means twice as many logical NUMA nodes as Siloz-1024 are needed to represent the correspondingly-larger number of subarray groups. Likewise, Siloz-2048's larger subarray group sizes require half as many nodes as Siloz-1024.

As shown in Fig. 5.6 (execution time) and Fig. 5.7 (throughput), I find $< 0.5\%$ geometric mean overheads for the performance of Siloz-512 and Siloz-2048 when normalized to that of Siloz-1024. The fact that there are no clear trends (nor significant differences) in mean timing and bandwidth as a function of subarray size is expected, given that subarray size does not effect DDR access

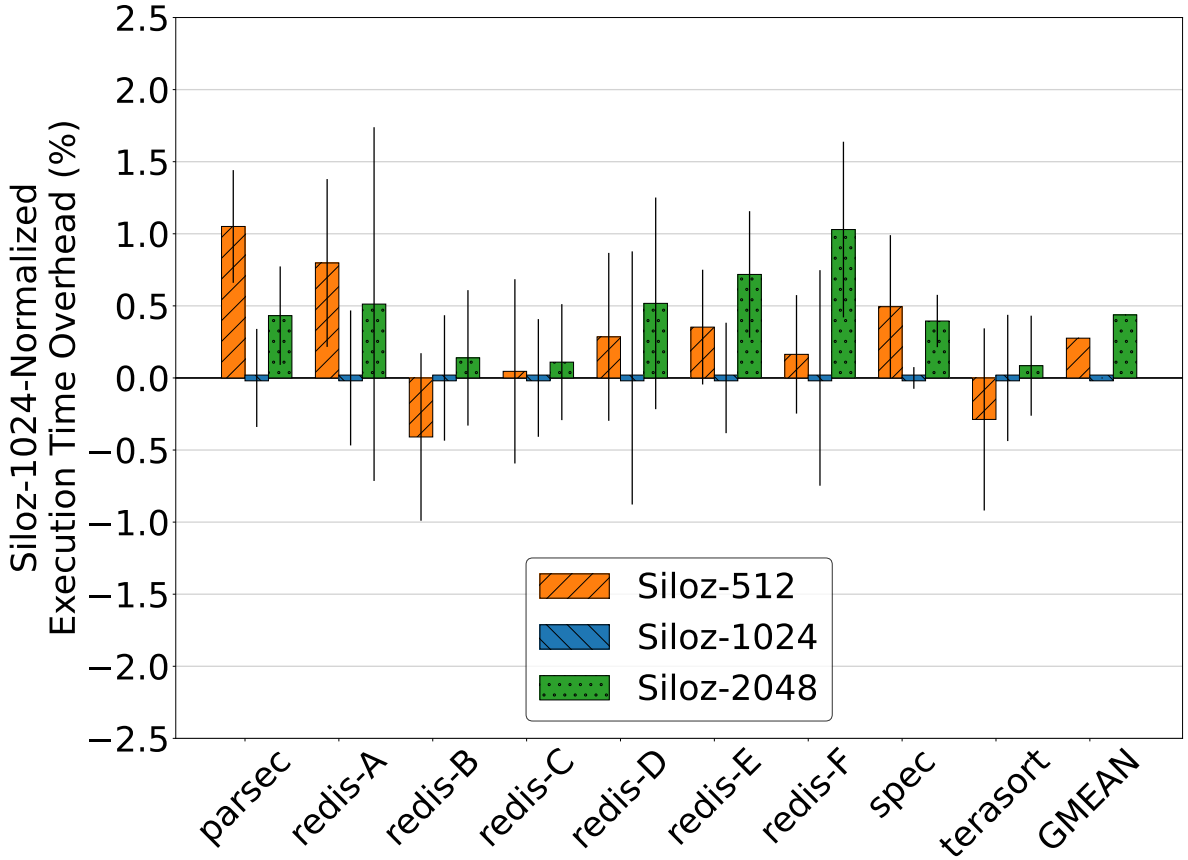

Figure 5.6: Siloz-1024-normalized execution time when varying from 512 to 2048 row groups per subarray group (§5.7.4). Error bars depict 95% confidence intervals. Lower is better.

times nor bank-level parallelism.

Furthermore, the lack of a trend is further indicative that the number of NUMA nodes does not play a significant role in performance, and that the most likely source of performance differences among the baseline and Siloz variants is simply noise. In particular, if NUMA node iterations played a significant role in performance, one would expect the Siloz variant with the fewest nodes (Siloz-2048) to outperform the one with most nodes (Siloz-512), which is *not* the case.

5.8 Discussion

5.8.1 Memory Fragmentation

VMs (especially micro-VMs [1, 312]) may have finer-grained DRAM demands than Siloz's subarray group size for a given server configuration. Thus, provisioning an entire subarray group for

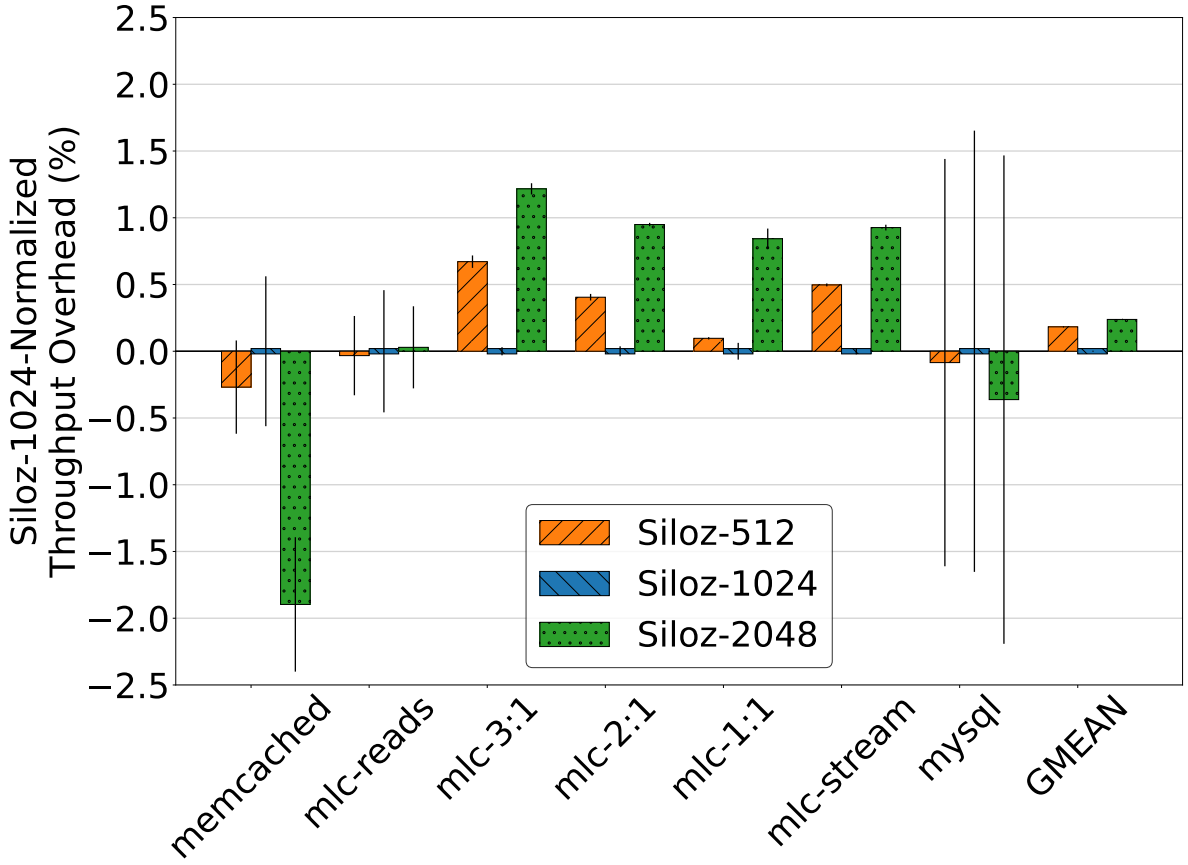

Figure 5.7: Siloz-1024-normalized throughput time when varying from 512 to 2048 row groups per subarray group (§5.7.4). Error bars depict 95% confidence intervals. Lower is better.

relatively small needs (e.g., a 1.5 GB subarray group to a VM needing 512 MB) risks wasting significant DRAM.

While major cloud providers often already offer VM sizing at similar granularity to Siloz [207, 282], there are nonetheless two key ways to mitigate potential fragmentation. First, multiple VMs may be safely provisionable in the same subarray group if they belong to the same trust domain (e.g., tenant). Second, the subarray group size may be adjustable on systems that support restricted interleaving (e.g., disabling interleaving across memory controllers, as done for sub-NUMA clusters [209]) to better align with VM needs; the subarray group size linearly decreases with the number of banks touched by each page (§5.4.1). In future systems, greater control over physical-to-media address mappings [39] could allow Siloz to make dynamic trade-offs for different VM sizes.

5.8.2 Considerations for Other DRAM Technologies

While today's DDR4 DRAM is already widely-deployed, DDR5 and HBM2 DRAM are being increasingly-deployed in servers and are still vulnerable to Rowhammer [151, 232]. I thus discuss how differences in DDR5 and HBM2 could impact Siloz's implementation.

First, memory controllers may use different physical-to-media address mappings for these modules, requiring updated versions of Siloz's drivers for DDR4 DIMMs [102, 119]. Second, DDR5 and HBM2 can provide greater bank-level parallelism than DDR4 by increasing the number of banks per rank (and hence, per physical node). Thus, the upper bound of Siloz's subarray group sizes could proportionally increase in these geometries, yielding coarser-grained memory management/provisioning (which can be offset using techniques described in §5.8.1).

In addition to these implementation effects, DDR5 standards actually ease subarray group isolation for non-power-of-2 subarray sizes. Specifically, DDR5 stipulates that any DIMM-internal address mirroring and inversion for signal routing and integrity (§5.6) must be *undone* upon arriving at each DRAM device [130], potentially to ease reasoning about DRAM faults/errors. Thus, Siloz would not have to create artificial subarray groups for non-power-of-2 subarray sizes in DDR5, as all devices use the same internal addresses.

5.8.3 Alternate EPT Protection

While Siloz provides EPT integrity using guard rows or emerging hardware extensions [8, 117], I emphasize that a variety of EPT bit flip mitigations can help to enforce Siloz's subarray group isolation. For instance, a state-of-the-art SoftTRR-like [347] *software refresh routine* could periodically refresh EPT rows to protect their values against bit flips.

We chose to use guard rows instead of a software refresh routine because of the difficulty of providing real-time guarantees in the Linux kernel [193], especially in many-core, generic production environments. I found that scheduling a software refresh routine to run every 1 ms (as would be required to protect EPT rows via periodic refresh [347]) did *not* consistently meet 1 ms deadlines between refreshes. Rather, I observed *at least* 1 ms intervals due to Linux scheduling semantics [290], even observing a period greater than *32 ms* between software refreshes (over 32 times a safe period).

To avoid scheduling delays, I instead ran the refresh routine immediately upon receipt of a periodic timer tick interrupt (i.e., during the interrupt request, rather than as its own task). I first found that I had to deactivate Linux optimizations that disable the timer tick on idle cores for power and performance savings [4, 263, 277, 314]. Then, despite forcing tick interrupts to unconditionally fire, I found that the interrupts could still be delayed or dropped for various reasons [4, 263, 277, 314], such as interrupts being disabled. These delayed/dropped tick interrupts resulted in missed

refresh timing deadlines, leaving EPTs vulnerable to bit flips even in the presence of redundant ticks, tick skew [99] across cores, and performance-costly real-time kernel patches.

Ultimately, real-time hardware is required to provide timing *guarantees* [193], which is not generally-practical for already-deployed cloud servers.

5.8.4 Broader Applicability to DRAM Isolation

Siloz uses logical NUMA nodes to manage subarray groups for Rowhammer isolation; however, cloud providers could use logical NUMA nodes to manage other units of DRAM isolation (e.g., banks, ranks, channels, or memory controllers). These units of isolation are attractive in that they could provide VMs with security isolation against additional DRAM timing [237] and power [50] side channels, as well as performance isolation (e.g., memory controller scheduling [212]).

The key challenge to extending Siloz's abstractions beyond subarray groups is the compatibility of physical-to-media address mappings. Given mappings that interleave a memory page across a physical node's banks (§5.2.4), these forms of isolation are not feasible in default configurations. However, extended addressing control (§5.8.1) could enable Siloz's application to these units, allowing cloud providers to offer a richer set of isolation options. Alternatively, modifying future DRAM to support subarray-level parallelism [153] could allow subarray groups themselves to offer such protection, mitigating forms of inter-domain DRAM interference not yet handled by the baseline or Siloz.

5.9 Related Work

Rowhammer attacks/analyses. Rowhammer bit flips were publicized in 2014 [152], spurring attacks/analyses [50, 51, 52, 53, 59, 72, 75, 91, 92, 98, 124, 125, 131, 147, 152, 152, 160, 160, 167, 180, 185, 190, 215, 218, 219, 221, 232, 233, 241, 250, 261, 269, 288, 289, 299, 300, 325, 328, 346]. These works motivate Siloz's Rowhammer protection.

Rowhammer mitigations. Beyond deployed-but-vulnerable mitigations discussed in §5.2.5, other hardware and software mitigations offer a range of security-performance trade-offs and are not known to be deployed [13, 16, 21, 27, 31, 41, 63, 73, 74, 81, 84, 96, 97, 106, 132, 134, 147, 149, 150, 151, 152, 161, 171, 175, 183, 185, 199, 200, 236, 242, 256, 262, 300, 308, 319, 322, 330, 336, 337, 348]. I analyze mitigations most-related to Siloz in §5.3. To our knowledge, Siloz is the first mitigation to offer comprehensive and high-performance protection against inter-VM hammering.

Other DRAM Side Channels. DRAMA [237] shows that DRAM accesses can leak information through timing side channels, such as bank conflicts. HammerScope [50] shows similar leakage through power side channels. Proposed mitigations [61, 272, 309, 350] are largely-based

on avoiding simultaneous contention for a DRAM resource (e.g., a bank). Combining these mitigations with Siloz's inter-VM Rowhammer protection is a potential avenue for future work.

Subarrays. mFIT [52] reverse engineers server DDR4 DRAM module subarray sizes and demonstrates that Rowhammer attacks are ineffective across subarrays, as asserted in prior work [183, 330]. X-ray [221] similarly infers subarray structure in select DDR4 and HBM2 modules. Siloz builds on these findings and insights to mitigate inter-VM hammering via subarray group isolation. Other work proposes using subarrays for efficient in-DRAM data movement [43, 308] and implementing subarray-level parallelism for DRAM activations [153] and refreshes [329], performance optimizations from which Siloz-isolated VMs may benefit. Furthermore, such forms of subarray-level parallelism could help Siloz additionally mitigate inter-VM performance interference (e.g., bank conflicts among different subarray groups) and associated timing side channels.

5.10 Conclusion

In this work, I have presented Siloz, a hypervisor that brings memory isolation domains to the cloud in the form of DRAM subarray groups. Already on today's hardware, Siloz provides VMs with efficient and comprehensive protection against inter-VM hammering. I hope that Siloz's effectiveness and practicality spur further development of memory isolation domains against additional forms of interference.

CHAPTER 6

Future Work and Conclusion

In light of emerging trends in Computer Science, my dissertation research offers three key directions for future work.

6.1 Microarchitectural Context Isolation for Performance

The benefits of microarchitectural context isolation can be extended beyond security and reliability into the realm of performance isolation. In particular, it is well-known that contexts sharing microarchitectural structures—including those exploited in both transient execution attacks and Rowhammer attacks—can degrade each other's performance [12, 214]. Therefore, isolating contexts in these structures offers the potential to improve performance, in addition to the security and reliability improvements provided in this dissertation.

For instance, in processor microarchitectures, a save-and-restore scheme for performance-critical [30] state could eliminate significant sources of context switching performance overheads, while also providing inter-context security isolation. In DRAM microarchitectures, rethinking the trade-offs between bank interleaving and row buffer locality in DRAM addressing [184] could limit various forms of timing interference, including bank conflicts that play a role in both Rowhammer attacks and performance degradation. Demonstrating microarchitectural context's ability to provide both security *and* performance isolation would spur its viability in commodity cloud systems.

6.2 Coherence-Induced Hammering in Emerging Architectures

MOESI-prime (chapter 4) shows that commodity coherence protocols in socket-based ccNUMA architectures use DRAM so liberally as to yield coherence-induced hammering during inter-node data sharing. Two emerging ccNUMA architectures could increase the likelihood of such hammering without careful design, therefore warranting further analysis and potential mitigations.

First, the advent of chiplet server architectures [3] (to a first approximation, sub-socket NUMA nodes) means that inter-node sharing is likely to *increase*. In particular, fewer cores per chiplet-based node (as compared to per socket-based node) may necessitate increased inter-node data sharing, potentially resulting in more frequent hammering.

Second, the rise of CXL servers [273]—which enable fine-grained, coherent data sharing among accelerators and the host—could further increase the likelihood and frequency of coherence-induced hammering. In fact, the CXL.cache protocol currently uses MESI coherence, which I have shown can cause hammering via downgrade writebacks without careful design (§4.3.2). The CXL.mem protocol includes optional support for a 2-bit memory directory, which I have likewise shown is a source of hammering without additional considerations (§4.3.3).

In addition to the security and reliability threats identified in MOESI-prime, coherence-induced hammering in these architectures could pose a greater threat to performance and energy consumption, given the potential increase in inter-node data sharing. Specifically, the minimal performance and energy impact of hammering observed in (relatively-rare) inter-socket sharing could rise with (relatively-frequent) inter-chiplet, host-accelerator, and accelerator-accelerator sharing. Thus, further research is needed to understand the presence, frequency, effects, and potential mitigation of coherence-induced hammering on these architectures.

6.3 (Silent) Data Corruption Beyond DRAM

While much of my work focuses on the phenomenon of data corruption/disturbances in DRAM, the threat of data corruption is likewise increasing in processors [14, 65, 101, 104, 270]; additional work predicts that corruption rates will similarly rise in other types of memory and storage devices [216]. Thus, circuit-level interference effects [28, 46, 69, 88, 166, 168, 246, 257, 275, 292] will be increasingly felt across a wide variety of computer hardware as the industry scales to denser (more susceptible) process nodes.

To mitigate this susceptibility, future research can compare and contrast the conditions leading to data corruption in various computer hardware (e.g., processors versus DRAM). One particularly interesting direction would be an analysis of data patterns leading to highest-likelihood of errors in vulnerable processors, similar to analysis of patterns in prior work on DRAM [52]. Given knowledge of such patterns, it may be possible to (a) increase the likelihood and speed of discovering circuit-level defects, and (b) decrease the likelihood of errors in production by biasing systems against using the worst-case data patterns.

6.4 Conclusion

In this dissertation, I have presented four first-author papers that offer efficient mechanisms to improve microarchitectural context isolation for cloud security and reliability. *DOLMA* (chapter 2) provides an effective mitigation against transient execution attacks, while mitigations proposed in *Stop! Hammer Time* (chapter 3), *MOESI-prime* (chapter 4), and *Siloz* (chapter 5) prevent various forms of Rowhammer attacks/behavior.

Together, my work demonstrates that microarchitectural context isolation in CPUs and DRAM can efficiently mitigate prominent hardware-based vulnerabilities in cloud systems. Microarchitectural context isolation's focus on the most crucial and expected form of cloud hardware isolation—inter-domain—enables efficient and effective protection, while avoiding the notoriously-challenging burden of preventing every possible hardware vulnerability at its source.

In fact, microarchitectural context isolation's combination of efficiency and effectiveness is precisely what renders it such an enticing mitigation option; such isolation naturally extends the architectural context isolation principles upon which the systems and architecture communities have built existing hardware and software, even improving performance in some cases. Just as architectural context isolation provides a viable path to security at the hardware-software interface, so too does microarchitectural context isolation in the hardware implementation. As emerging architectures and denser process nodes continue to push the reliability, security, and performance limits of computer hardware, microarchitectural context isolation techniques offer feasible options for combining reliable and secure operation with high performance.

BIBLIOGRAPHY

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] S. Ainsworth and T. M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2020.

[3] Paul Alcorn. New ucie chiplet standard supported by intel, amd, and arm. tomshardware.com/news/new-ucie-chiplet-standard-supported-by-intel-amd-and-arm, 2022.

[4] Abdullah Aljuhni, C Edward Chow, Amer Aljaedi, Shaji Yusuf, and Francisco Torres-Reyes. Towards understanding application performance and system behavior with the full dynticks feature. In *IEEE Computing and Communication Workshop and Conference (CCWC)*, 2018.

[5] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2018.

[6] Hesham Altwaijry and Diyab S Alzahrani. Improved-moesi cache coherence protocol. *Arabian Journal for Science and Engineering*, 2014.

[7] AMD. Speculative Store Bypass Disable, 2018. developer.amd.com/wp-content/resources/124441-AMD64-SpeculativeStoreBypassDisable-Whitepaper-final.pdf.

[8] AMD. AMD Secure Encrypted Virtualization (SEV), 2020. https://developer.amd.com/sev/.

[9] Rakesh Anigundi, Hongbin Sun, Jian-Qiang Lu, Ken Rose, and Tong Zhang. Architecture design exploration of three-dimensional (3d) integrated dram. In *International Symposium on Quality Electronic Design*, 2009.

[10] ARM. CP15 c9, cache lockdown support, 2018. https://developer.arm.com/documentation/ddi0406/cb/Appendixes/ARMv4-and-ARMv5-Differences/System-Control-coprocessor--CP15-support/CP15-c9--cache-lockdown-support.

[11] Krste Asanović and David A Patterson. Instruction sets should be free: The case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[12] Adam Auten, Tanishq Dubey, and Rohan Mathur. Dynamically improving branch prediction accuracy between contexts. *arXiv preprint arXiv:1805.00585*, 2018.

[13] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ACM SIGARCH Computer Architecture News (CAN)*, 2016.

[14] David F Bacon. Detection and prevention of silent data corruption in an exabyte-scale database system. *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects, IEEE (2022)*, 2022.

[15] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[16] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row hammer refresh command, 2015. US Patent 9,117,544.

[17] Kristin Barber, Anys Bacha, Langming Zhou, Yiyan Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *PACT*, 2019.

[18] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[19] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew Walton, and Vilas Sridharan. A systematic study of ddr4 dram faults in the field. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2023.

[20] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)*, 2005.

[21] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A complete in-dram rowhammer mitigation. In *Workshop on DRAM Security (DRAMSec)*, 2021.

[22] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[23] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[24] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News (CAN)*, 2011.

[25] Arijit Biswas. Sapphire rapids. In *IEEE Hot Chips Symposium (HCS)*, 2021.

[26] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[27] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks. In *ACM Asia Conference on Computer and Communications Security (Asia CCS)*, 2019.

[28] Mark T Bohr and Ian A Young. Cmos scaling trends and beyond. *IEEE Micro*, 2017.

[29] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[30] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[31] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security Symposium (USENIX Security)*, 2017.

[32] James Bucek, Klaus-Dieter Lange, et al. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018.

[33] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.

[34] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium (USENIX Security)*, 2019.

[35] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[36] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[37] Michael Andrew Carlton, Sean P Blanchard, and Nathan A Debardeleben. Improving memory error handling using linux. Technical report, Los Alamos National Lab., 2014.

[38] Chandler Carruth. *Speculative Load Hardening*. Google, 2018. llvm.org/docs/SpeculativeLoadHardening.html.

[39] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, et al. Impulse: Building a smarter memory controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 1999.

[40] Chris Celio, Jerry Zhao, Abraham Gonzalez, and Ben Korpan. Riscv-boom documentation, 2019.

[41] A. Chakraborty, M. Alam, and D. Mukhopadhyay. Deep Learning Based Diagnostics for Rowhammer Protection of DRAM Chips. In *IEEE Asian Test Symposium (ATS)*, 2019.

[42] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. drampower.info, 2012.

[43] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[44] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving dram performance by parallelizing refreshes with accesses. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[45] 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 *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2019.

[46] Jun Chen and Lei He. Determination of worst-case crosstalk noise for non-switching victims in ghz+ interconnects. In *Asia and South Pacific Design Automation Conference*, 2003.

[47] Liqun Cheng and John B Carter. Fast barriers for scalable ccnuma systems. In *IEEE International Conference on Parallel Processing (ICPP)*, 2005.

[48] Liqun Cheng, John B Carter, and Donglai Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[49] Keewon Cho, Wooheon Kang, Hyungjun Cho, Changwook Lee, and Sungho Kang. A survey of repair analysis algorithms for memories. *ACM Computing Surveys (CSUR)*, 2016.

[50] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. Hammerscope: Observing dram power consumption using rowhammer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[51] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[52] Lucian Cojocar, Kevin Loughlin, Stefan Saroiu, Baris Kasikci, and Alec Wolman. mfit: A bump-in-the-wire tool for plug-and-play analysis of rowhammer susceptibility factors. *Microsoft Tech Report*, 2021.

[53] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[54] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 2010.

[55] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *ACM symposium on Cloud computing*, 2010.

[56] Jonathan Corbet. *A page-table isolation update*. LWN, 2018. lwn.net/Articles/752621/.

[57] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[58] Alan L Cox and Robert J Fowler. Adaptive cache coherency for detecting migratory shared data. *ACM SIGARCH Computer Architecture News (CAN)*, 1993.

[59] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security Symposium (USENIX Security)*, 2021.

[60] Timothy J Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics division*, 1997.

[61] Peter W Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S Emer, and Mengjia Yan. Dagguise: mitigating memory timing side channels. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[62] The QEMU Project Developers. The memory api. qemu.readthedocs.io/en/latest/devel/memory.html, 2022.

[63] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-flip: Hardening ecc memory against rowhammer attacks. In *Network and Distributed System Security (NDSS) Symposium*, 2023.

[64] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. *ACM SIGPLAN Notices*, 2012.

[65] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. Detecting silent data corruptions in the wild. *arXiv preprint arXiv:2203.08989*, 2022.

[66] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[67] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 2012.

[68] Chris Down. 5 years of cgroup v2: The future of linux resource control. *USENIX Large Installation System Administration Conference*, 2021.

[69] Ahmed N Elkammar, Norman Scheinberg, and Srinivasa R Vemuru. Encoding to reduce crosstalk noise and power dissipation: A new closed formula. In *IEEE International Midwest Symposium on Circuits and Systems*, 2006.

[70] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[71] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, 2018.

[72] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *ACM SIGSAC conference on computer and communications security (CCS)*, 2022.

[73] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

[74] Ali Fakhrzadehgan, Prakash Ramrakhyani, Moinuddin K Qureshi, and Mattan Erez. Secddr: Enabling low-cost secure memories by protecting the ddr interface. *arXiv preprint arXiv:2209.00685*, 2022.

[75] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[76] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2019.

[77] Varun Gandhi and James Mickens. Rethinking isolation mechanisms for datacenter multi-tenancy. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[78] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccnuma: Exploiting skew with strongly consistent caching. In *ACM European Conference on Computer Systems (EuroSys)*, 2018.

[79] gem5. Architecture support. gem5.org/documentation/general-docs/architecture-support/, 2021.

[80] gem5. gem5-20 working status of benchmarks. gem5.org/documentation/benchmark-status/gem5-20, 2021.

[81] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. Armor: A run-time memory hot-row detector, 2015.

[82] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2019.

[83] Klaus v Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Iodine: Verifying constant-time execution of hardware. In *USENIX Security Symposium (USENIX Security)*, 2019.

[84] Hector Gomez, Andres Amaya, and Elkim Roa. DRAM row-hammer attack reduction using dummy cells. In *IEEE Nordic Circuits and Systems Conference (NORCAS)*, 2016.

[85] JR Goodman and HHJ Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004). *URL: https://www. cs. auckland. ac. nz/˜ goodman/TechnicalReports/MESIF-2009. pdf*, 2004.

[86] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium (USENIX Security)*, 2018.

[87] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *USENIX Security Symposium (USENIX Security)*, 2017.

[88] Ed Grochowski, David Ayers, and Vivek Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2002.

[89] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium (USENIX Security)*, 2017.

[90] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer, 2017.

[91] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[92] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[93] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[94] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. *arXiv preprint arXiv:2006.03841*, 2020.

[95] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. Simulating dram controllers for future system architecture exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[96] Hasan Hassan, Ataberk Olgun, A Giray Yaglikci, Haocong Luo, and Onur Mutlu. A case for self-managing dram chips: Improving performance, efficiency, reliability, and security via autonomous in-dram maintenance operations. *arXiv preprint arXiv:2207.13358*, 2022.

[97] Hasan Hassan, Minesh Patel, Jeremie S Kim, A Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2019.

[98] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[99] Red Hat. 2.14. reduce cpu performance spikes. https://access.redhat.com/documentation/en-us/red-hat-enterprise-linux-for-real-time/7/html/tuning-guide/reduce-cpu-performance-spikes.

[100] Andrew Hay. *MESIF Cache Coherence Protocol*. PhD thesis, ResearchSpace@ Auckland, 2012.

[101] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2023.

[102] Marius Hillenbrand. Physical address decoding in intel xeon v3/v4 cpus: A supplemental datasheet. *Karlsruhe Institute of Technology, Tech. Rep.*, 2017.

[103] Marius Hillenbrand, Mathias Gottschlag, Jens Kehne, and Frank Bellosa. Multiple Physical Mappings: Dynamic DRAM Channel Sharing and Partitioning. In *ACM Asia-Pacific Workshop on Systems (APSys)*, 2017.

[104] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[105] Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. The effect of numa tunings on cpu performance. In *Journal of Physics: Conference Series*, 2015.

[106] Seungki Hong, Dongha Kim, Jaehyung Lee, Reum Oh, Changsik Yoo, Sangjoon Hwang, and Jooyoung Lee. Dsac: Low-cost rowhammer mitigation using in-dram stochastic and approximate counting algorithm. *arXiv preprint arXiv:2302.03591*, 2023.

[107] Masashi Horiguchi and Kiyoo Itoh. *Nanoscale memory repair*. Springer Science & Business Media, 2011.

[108] Chih-Sheng Hou, Yong-Xiao Chen, Jin-Fu Li, Chih-Yen Lo, Ding-Ming Kwai, and Yung-Fa Chou. A built-in self-repair scheme for drams with spare rows, columns, and bits. In *2016 IEEE International Test Conference (ITC)*, 2016.

[109] Intel. Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Unit Reference Manual, 2014. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/xeon-e5-2600-v2-uncore-manual.pdf.

[110] Intel. Intel 64 and IA32 Architectures Performance Monitoring Events, 2017. https://software.intel.com/sites/default/files/managed/8b/6e/335279-performance-monitoring-events-guide.pdf.

[111] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*, 2018. software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

[112] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.

[113] Intel. Speculative Execution Side Channel Mitigations, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[114] Intel. Speculative Store Bypass, 2018. software.intel.com/security-software-guidance/advisory-guidance/speculative-store-bypass.

[115] Intel. Intel Server Board S2600 Family BIOS Setup User Guide, 2019. https://www.intel.com/content/dam/support/us/en/documents/server-products/Intel-Xeon-Processor-Scalable-Family-BIOS-User-Guide.pdf.

[116] Intel. Intel architecture day 2021, 2021. https://www.intel.com/content/www/us/en/newsroom/resources/press-kit-architecture-day-2021.html.

[117] Intel. Intel Trust Domain Extensions (Intel TDX), 2022. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[118] Intel. Introduction to memory bandwidth allocation. intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html, 2022.

[119] Intel. SKX EDAC Linux Driver. github.com/torvalds/linux/blob/master/drivers/edac/skx-base.c, 2022.

[120] Intel Community Forum. Skl - strange memory behavior, 2019.

[121] Intel Xeon Processor Scalable Memory Family. Uncore performance monitoring reference manual. *Intel Corporation*, 2017.

[122] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *IEEE Euromicro Conference on Digital System Design*, 2015.

[123] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Preventing Microarchitectural Attacks Before Distribution. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[124] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[125] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[126] JEDEC. *Double Data Rate 4 (DDR4) SDRAM Standard*, 2014.

[127] JEDEC. *Low Power Double Data Rate 4 (LPDDR4) SDRAM Standard*, 2017. JESD209-4B.

[128] JEDEC. *DDR4 Registering Clock Driver Definition (DDR4RCD02)*, 2019.

[129] JEDEC. *Double Data Rate 5 (DDR5) SDRAM Standard*, 2020.

[130] JEDEC. *DDR5 Registering Clock Driver Definition (DDR5RCD02)*, 2023.

[131] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In *ACM Asia Conference on Computer and Communications Security (Asia CCS)*, 2019.

[132] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Machine learning-based rowhammer mitigation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[133] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached design on high performance rdma capable interconnects. In *IEEE International Conference on Parallel Processing (ICPP)*, 2011.

[134] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. Csi: Rowhammer–cryptographic security and integrity against rowhammer. In *IEEE Symposium on Security and Privacy (S&P)*, 2023. To appear.

[135] David Kanter. The common system interface: Intel's future interconnect. *Real World Technologies*, 2007.

[136] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[137] Brent Keeth, R Jacob Baker, Brian Johnson, and Feng Lin. *DRAM circuit design: fundamental and high-speed topics*, volume 13. John Wiley & Sons, 2007.

[138] The kernel development community. Numa memory policy. `kernel.org/doc/html/latest/admin-guide/mm/numa-memory-policy.html`, 2022.

[139] Richard E Kessler. The alpha 21264 microprocessor. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1999.

[140] Samira Khan, Donghyuk Lee, and Onur Mutlu. Parbor: An efficient system-level technique to detect data-dependent failures in dram. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[141] Samira Khan, Chris Wilkerson, Donghyuk Lee, Alaa R Alameldeen, and Onur Mutlu. A case for memory content-based detection and mitigation of data-dependent failures in dram. *IEEE Computer Architecture Letters (CAL)*, 2016.

[142] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and mitigating data-dependent dram failures by exploiting current memory content. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[143] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *IEEE/ACM Design Automation Conference (DAC)*, 2019.

[144] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters (CAL)*, 2014.

[145] Jeremie Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. Solar-dram: Reducing dram access latency by exploiting the variation in local bitlines. In *IEEE International Conference on Computer Design (ICCD)*, 2018.

[146] Jeremie S Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-range: Using commodity dram devices to generate true random numbers with low latency and high throughput. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[147] Jeremie S Kim, Minesh Patel, A Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2020.

[148] Jooyoung Kim, Woosung Lee, Keewon Cho, and Sungho Kang. Hardware-efficient built-in redundancy analysis for memory with various spares. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[149] M. Kim, J. Choi, H. Kim, and H. Lee. An Effective DRAM Address Remapping for Mitigating Rowhammer Errors. *IEEE Transactions on Computers*, 2019.

[150] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. *arXiv preprint arXiv:2108.06703*, 2021.

[151] Woongrae Kim, Chulmoon Jung, Seongnyuh Yoo, Duckhwa Hong, Jeongjin Hwang, Jungmin Yoon, Ohyong Jung, Joonwoo Choi, Sanga Hyun, Mankeun Kang, et al. A 1.1 v 16gb ddr5 dram with probabilistic-aggressor tracking, refresh-management functionality, per-row hammer tracking, a multi-step precharge, and core-bias modulation for security and reliability enhancement. In *IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023.

[152] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2014.

[153] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2012.

[154] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[155] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[156] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[157] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Linux symposium*, 2007.

[158] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[159] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.

[160] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering from the next row over. In *USENIX Security Symposium (USENIX Security)*, 2022.

[161] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[162] Alexey Kopytov. Sysbench manual. *MySQL AB*, 2012.

[163] E. M. Koruyeh, S. Haji Amin Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[164] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT*, 2018.

[165] Akhilesh Kumar, Don Soltis, Irma Esmer, Adi Yoaz, and Sailesh Kottapalli. The new intel xeon scalable processor (formerly skylake-sp). In *IEEE Hot Chips Symposium (HCS)*, 2017.

[166] Sumeet S Kumar, Gokulraj Chandramohan, Willem van Driel, and GQ Zhang. Effects of crosstalk and simultaneous switching noise on high performance digital system packages. In *IEEE International Conference on Electronic Packaging Technology & High Density Packaging*, 2010.

[167] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[168] Patrik Larsson. di/dt noise in cmos integrated circuits. *Analog Design Issues in Digital VLSI Circuits and Systems: A Special Issue of Analog Integrated Circuits and Signal Processing, An International Journal*, 1997.

[169] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N Patt. Improving memory bank-level parallelism in the presence of prefetching. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[170] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[171] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2019.

[172] Jung-Bae Lee. Green Memory Solution. In *Samsung Electronics, Investor's Forum*, 2014.

[173] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *ACM SIGARCH Computer Architecture News (CAN)*, 1990.

[174] Kevin M Lepak, Vydhyanathan Kalyanasundharam, William A Hughes, Benjamin Tsien, and Greggory D Donley. Method and apparatus for accelerated shared data migration, 2014. US Patent 8,732,410.

[175] C. Li and J. Gaudiot. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*, 2019.

[176] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023. To appear.

[177] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[178] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[179] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (USENIX Security)*, 2018.

[180] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing Rowhammer faults through network requests. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[181] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA source code. github.com/efeslab/dolma, 2020.

[182] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz source code. github.com/efeslab/siloz, 2023.

[183] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Stop! hammer time: rethinking our approach to rowhammer mitigations. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[184] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Software-defined memory controllers: An idea whose time has come. In *Wild and Crazy Ideas (WACI) Session at ASPLOS*, 2022.

[185] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2022.

[186] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. MOESI-prime source code. github.com/efeslab/moesi-prime, 2022.

[187] Robert Love. Kernel korner: Cpu affinity. *Linux Journal*, 2003.

[188] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[189] Shih-Lien Lu, Ying-Chen Lin, and Chia-Lin Yang. Improving dram latency with dynamic asymmetric subarray. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[190] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[191] Andrei Lutas and Dan Lutas. Bypassing kpti using the speculative behavior of the swapgs instruction. *Bitdefender Whitepaper*, 2019.

[192] Sangkug Lym, Heonjae Ha, Yongkee Kwon, Chun-kai Chang, Jungrae Kim, and Matta Erez. Eruca: Efficient dram resource utilization and resource conflict avoidance for memory system parallelism. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[193] Michael M Madden. Challenges using linux as a real-time operating system. In *AIAA Scitech 2019 Forum*, 2019.

[194] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[195] Neethu Bal Mallya, Geeta Patil, and Biju Raveendran. Simulation based performance study of cache coherence protocols. In *International Symposium on Nanoelectronic and Information Systems*, 2015.

[196] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *WOOT*, 2019.

[197] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2006.

[198] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccnuma via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 2010.

[199] Michele Marazzi, Patrick Jattke, Solt Flavien, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[200] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. Rega: Scalable rowhammer mitigation with refresh-generating activations. In *IEEE Symposium on Security and Privacy (S&P)*, 2023. To appear.

[201] John McCalpin. Address hashing in intel processors. *UT Faculty/Researcher Works*, 2018.

[202] John McCalpin. Topology and cache coherence in knights landing and skylake xeon processors. *UT Faculty/Researcher Works*, 2018.

[203] John McCalpin. Directory structure in skylake server cpus. *Intel Community Forum*, 2020.

[204] John D McCalpin. Stream benchmark. *Link: www.cs.virginia. edu/stream/ref.html#what*, 1995.

[205] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[206] Micron. DDR4 SDRAM EDY4016A - 256Mb x 16, 2014. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb-ddr4-dram-2e0d.pdf.

[207] Microsoft. High performance computing vm sizes, 2023.

[208] Adrian C Moga, Malcolm Mandviwalla, Vedaraman Geetha, and Herbert H Hum. Allocation and write policy for a glueless area-efficient directory cache for hotly contested cache lines, 2013. US Patent 8,392,665.

[209] Srikanta Kumar Mohapatra et al. Authentication of sub-numa clustering effect on intel skylake for memory latency and bandwidth. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 2021.

[210] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Workshop on Memory Systems Performance and Correctness*, 2014.

[211] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. In *IEEE International Conference on Parallel Processing (ICPP)*, 2015.

[212] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium (USENIX Security)*, 2007.

[213] David Mulnix. Intel xeon processor scalable family technical overview. *Intel Corporation*, 2017.

[214] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[215] Koksal Mus, Yarkın Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering tls signing keys via rowhammer faults. *Cryptology ePrint Archive*, 2022.

[216] O. Mutlu and J. S. Kim. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[217] Onur Mutlu. Memory scaling: A systems architecture perspective. In *IEEE International Memory Workshop*, 2013.

[218] Onur Mutlu. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[219] Onur Mutlu, Ataberk Olgun, and A Giray Yağlıkçı. Fundamentally understanding and solving rowhammer. *arXiv preprint arXiv:2211.07613*, 2022.

[220] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2020.

[221] Hwayong Nam, Seungmin Baek, Minbok Wi, Michael Jaemin Kim, Jaehyun Park, Chihun Song, Nam Sung Kim, and Jung Ho Ahn. X-ray: Discovering dram internal structure and error characteristics by issuing memory commands. *IEEE Computer Architecture Letters (CAL)*, 2023.

[222] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. Sapphire rapids: The next-generation intel xeon scalable processor. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

[223] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. Eddie: Em-based detection of deviations in program execution. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2017.

[224] Ian Neal. Lapidary: Crafting more beautiful gem5 simulations. medium.com/@iangneal/lapidary-crafting-more-beautiful-gem5-simulations-4bc6f6aad717, 2019.

[225] Ian Neal. Lapidary: creating beautiful gem5 simulations. github.com/efeslab/lapidary, 2019.

[226] NIST NVD. Cve-2018-3639. nvd.nist.gov/vuln/detail/CVE-2018-3639, 2018.

[227] NIST NVD. Cve-2018-3640. nvd.nist.gov/vuln/detail/CVE-2018-3640, 2018.

[228] NIST NVD. Cve-2018-3693. nvd.nist.gov/vuln/detail/CVE-2018-3693, 2018.

[229] NIST NVD. Cve-2019-11135. nvd.nist.gov/vuln/detail/CVE-2019-11135, 2019.

[230] J Norris. Package org.apache.hadoop.examples.terasort, 2013.

[231] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[232] Ataberk Olgun, Majd Osseiran, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, Onur Mutlu, et al. An experimental analysis of rowhammer in hbm2 dram chips. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[233] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S Kim, and Onur Mutlu. A deeper look into rowhammer's sensitivities: Experimental analysis of real dram chipsand implications on future attacks and defenses. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[234] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2018.

[235] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2020.

[236] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[237] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium (USENIX Security)*, 2016.

[238] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[239] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[240] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing half-double: New hammering technique for dram rowhammer bug. *Google Security Blog*, 2021.

[241] Rui Qiao and Mark Seaborn. A new approach for Rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016.

[242] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2022.

[243] Moinuddin K Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[244] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2019.

[245] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security and Privacy (S&P)*, 2021. To appear.

[246] Chittarsu Raghunandan, KS Sainarayanan, and MB Srinivas. Area efficient bus encoding technique for minimizing simultaneous switching noise (ssn). In *IEEE International Symposium on Circuits and Systems*, 2007.

[247] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2001.

[248] Sabela Ramos and Torsten Hoefler. Cache line aware optimizations for ccnuma systems. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015.

[249] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in numa multicore systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[250] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium (USENIX Security)*, 2016.

[251] Kaveh Razavi and Animesh Trivedi. Stratus: Clouds with microarchitectural resource management. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[252] TIRIAS Research. Second generation amd epyc processor enhanced cache and memory architecture, 2020.

[253] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News (CAN)*, 2000.

[254] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security Symposium (USENIX Security)*, 2021.

[255] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An undo approach to safe speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[256] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[257] KS Sainarayanan, JVR Ravindra, and MB Srinivas. Minimizing simultaneous switching noise (ssn) using modified odd/even bus invert method. In *IEEE International Workshop on Electronic Design, Test and Applications (DELTA)*, 2006.

[258] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. Ghost loads: what is the cost of invisible speculation? In *ACM CF*, 2019.

[259] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2019.

[260] Ciro Santilli. PARSEC Benchmark . github.com/cirosantilli/parsec-benchmark/, 2020.

[261] Stefan Saroiu, Alec Wolman, and Lucian Cojocar. The price of secrecy: How hiding internal dram topologies hurts rowhammer defenses. In *International Reliability Physics Symposium (IRPS)*, 2022.

[262] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[263] Stijn Schildermans, Kris Aerts, Jianchen Shan, and Xiaoning Ding. Paratick: Reducing timer overhead in virtual machines. In *IEEE International Conference on Parallel Processing (ICPP)*, 2021.

[264] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, 2019.

[265] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *Network and Distributed System Security (NDSS) Symposium*, 2020.

[266] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[267] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS*, 2019.

[268] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing of registers: Reviving foreshadow. *arXiv preprint arXiv:2008.02307*, 2020.

[269] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. *Black Hat*, 2015. See also http://googleprojectzero.blogspot.co/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[270] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519*, 2021.

[271] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2018.

[272] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[273] Debendra Das Sharma, Robert Blankenship, and Daniel S Berger. An introduction to the compute express link (cxl) interconnect. *arXiv preprint arXiv:2306.11227*, 2023.

[274] Vipin Sharma. Numa aware page table's page allocation. *LWN*, 2022.

[275] Jih-Sheng Shen, Pao-Ann Hsiung, and Kuei-Chung Chang. A novel spatio-temporal adaptive bus encoding for reducing crosstalk interferences with trade-offs between performance and reliability. In *IEEE Asia-Pacific Computer Systems Architecture Conference*, 2008.

[276] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[277] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. Getting maximum mileage out of tickless. In *Linux Symposium*. Citeseer, 2007.

[278] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[279] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM stronger against row hammering. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[280] Young Hoon Son, O Seongil, Yuhwan Ro, Jae W Lee, and Jung Ho Ahn. Reducing memory access latency with asymmetric dram bank organizations. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2013.

[281] SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. spec.org/cpu2017/.

[282] Androski Spicer. Deep dive on amazon ec2, 2017.

[283] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480*, 2018.

[284] Per Stenström, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. *ACM SIGARCH Computer Architecture News (CAN)*, 1993.

[285] Brian K Tanaka. Monitoring virtual memory with vmstat. *Linux Journal*, 2005.

[286] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. Improving bank-level parallelism for irregular applications. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[287] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[288] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against Rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[289] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[290] Linus Torvalds et al. Linux source code. https://github.com/torvalds/linux, 2023.

[291] C. Trippel, D. Lustig, and M. Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[292] S-W Tu, Y-W Chang, and J-Y Jou. Rlc coupling-aware simulation and on-chip bus encoding for delay reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.

[293] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. Google, 2018. support.google.com/faqs/answer/7625886.

[294] Unified Extensible Firmware Interface UEFI. Advanced configuration and power interface specification. *ACPI. INFO, Roseville*, 2013.

[295] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 2005.

[296] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium (USENIX Security)*, 2018.

[297] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[298] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[299] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *ACM SIGSAC conference on computer and communications security (CCS)*, 2016.

[300] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[301] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum to RIDL: Rogue In-flight Data Load, 2019. mdsattacks.com/files/ridl-addendum.pdf.

[302] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[303] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *cacheoutattack.com*, 2020.

[304] Ricardo A Velásquez, Pierre Michaud, and André Seznec. Selecting benchmark combinations for the evaluation of multicore throughput. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[305] Kirtana Venkatraman. Virtual machine memory allocation and place-ment on azure stack. *Microsoft Azure*, 2019.

[306] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. Intel memory latency checker v3.9a. *Intel*, 2021.

[307] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE TSE*, 2019.

[308] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, and O. Mutlu. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[309] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[310] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. Detect DRAM disturbance error by using disturbance bin counters. *IEEE Computer Architecture Letters (CAL)*, 2019.

[311] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. Reinforce Memory Error Protection by Breaking DRAM Disturbance Correlation Within ECC Words. In *IEEE International Conference on Computer Design (ICCD)*, 2019.

[312] Zicheng Wang. Can "micro vm" become the next generation computing platform?: Performance comparison between light weight virtual machine, container, and traditional virtual machine. In *IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*, 2021.

[313] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: transparent memory offloading in datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[314] Frederic Weisbecker. Status of linux dynticks. In *Workshop on Operating Systems Platforms for Embedded Real-Time applications-OSPERT13*. Citeseer, 2013.

[315] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[316] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical Report*, 2018.

[317] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *arXiv preprint arXiv:1912.11523*, 2019.

[318] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium (USENIX Security)*, 2019.

[319] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. Shadow: Preventing row hammer in dram with intra-subarray row shuffling. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2023.

[320] D Hyuk Woo and HH Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[321] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH Computer Architecture News (CAN)*, 1995.

[322] Xin-Chuan Wu, Timothy Sherwood, Frederic T Chong, and Yanjing Li. Protecting page tables from Rowhammer attacks using monotonic pointers in DRAM true-cells. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[323] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH Computer Architecture News (CAN)*, 2003.

[324] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[325] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium (USENIX Security)*, 2016.

[326] W. Xiong and J. Szefer. Leaking information through cache lru states. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[327] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks. *arXiv preprint arXiv:2005.13435*, 2020.

[328] A Giray Yağlıkçı, Haocong Luo, Geraldo F De Oliviera, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S Kim, Lois Orosa, and Onur Mutlu. Understanding rowhammer under reduced wordline voltage: An experimental study using real dram devices. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[329] A Giray Yağlikçi, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. Hira: Hidden row activation for reducing refresh latency of off-the-shelf dram chips. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[330] Abdullah Giray Yağlıkçı, Jeremie S Kim, Fabrice Devaux, and Onur Mutlu. Security analysis of the silver bullet technique for rowhammer prevention. *arXiv preprint arXiv:2106.07084*, 2021.

[331] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[332] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2017.

[333] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[334] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[335] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *Cryptology ePrint Archive*, 2015.

[336] A. Giray Yağlikçi, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[337] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating Row-hammering based on memory Locality. In *IEEE/ACM Design Automation Conference (DAC)*, 2019.

[338] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *Network and Distributed System Security (NDSS) Symposium*, 2019.

[339] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2020.

[340] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[341] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT): A Formal Analysis. *Technical Report*, 2019.

[342] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. STT source code. `github.com/cwfletcher/stt`, 2020.

[343] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[344] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec 3.0: A multicore benchmark suite with network stacks and splash-2x. *ACM SIGARCH Computer Architecture News (CAN)*, 2017.

[345] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000.

[346] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[347] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Nepal Surya, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. SoftTRR: Protect Page Tables Against RowHammer Attacks using Software-only Target Row Refresh. *USENIX Annual Technical Conference (ATC)*, 2022.

[348] Ziyuan Zhang, Meiqi Wang, Wencheng Chen, Han Qiu, and Meikang Qiu. Mitigating targeted bit-flip attacks via data augmentation: An empirical study. In *International Conference on Knowledge Science, Engineering and Management*, 2022.

[349] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[350] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. Camouflage: Memory traffic shaping to mitigate timing attacks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.