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

Kevin Loughlin
University of Michigan

Stefan Saroiu
Microsoft

Alec Wolman
Microsoft

Yatin A. Manerkar
University of Michigan

Baris Kasikci
University of Michigan

ABSTRACT

Prior work shows that Rowhammer attacks—which flip bits in DRAM via frequent activations of the same row(s)—are viable. Adversaries typically mount these attacks via instruction sequences that are carefully-crafted to bypass CPU caches. However, we discover a novel form of hammering that we refer to as *coherence-induced hammering*, caused by Intel's implementations of cache coherent non-uniform memory access (ccNUMA) protocols. We 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, we 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 ±0.61% of MESI and MOESI) and average DRAM power (0.03%–0.22% improvement) across evaluated ccNUMA configurations.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; **Systems security**; • **Hardware → Hardware reliability**.

KEYWORDS

Rowhammer, Security, Reliability, Coherence Protocol

ACM Reference Format:
Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. 2022. MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527427

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527427

1 INTRODUCTION

The threat of Rowhammer [61] 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 [20, 22, 25, 30, 38, 39, 41, 48, 49, 51, 58, 61, 65, 70, 84, 88, 94, 95, 101, 108, 111–114, 119] 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, we 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, we offer the first evidence of Rowhammer's additional *reliability* threat.

Using DDR4 DRAM access traces, we 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 [58, 88, 107]), it is paramount to revisit ccNUMA protocol design before already-vulnerable mitigations [25, 30, 41, 49, 94] are overwhelmed.

We 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 [85], 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 [85].

Unfortunately, we 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 [24, 110], 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 [78] 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 we will show, these phenomena *cannot be prevented by a conventional MOESI protocol.*

Accordingly, in this work, we 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.

We 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**), we 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 (§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.

We evaluate MOESI-prime in gem5 [9, 73], using a full-system configuration that models a major cloud provider's production hardware. We demonstrate that MOESI-prime prevents identified sources of coherence-induced hammering in both malicious and non-malicious workloads. Additionally, we prove that baseline MESI/MOESI protocols can be transformed into MOESI-prime protocols without loss of correctness. Finally, we show that MOESI-prime's prevention of unnecessary reads and writes has negligible effect on average performance (within ±0.61% of MESI and MOESI baselines) and average DRAM power (0.03%–0.22% improvement) across PARSEC 3.0 [123] and SPLASH-2x [117] in 2-, 4-, and 8-node ccNUMA configurations.

In summary, we make the following contributions:

- Using DDR4 memory access traces—collected from commodity benchmarks on a major cloud provider's production hardware—we discover *coherence-induced hammering*, the first hammering found to occur in non-malicious code.
- We identify hammering sources in Intel ccNUMA protocols.
- We 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.
- We 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 [72].

2 BACKGROUND

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 [61] 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 [22]) and target row refresh (TRR [30], 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 [22] and TRR [25, 30, 49, 94]. Alternative mitigations yield a range of security-performance trade-offs and are not known to be deployed [4, 6, 7, 12, 14, 16, 29, 34, 36, 58–61, 63, 66, 69, 71, 77, 91, 105, 114, 115, 118, 121, 122].

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 [44], or core complex/chiplet [102]) 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 [11, 100], 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).

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 [42]—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 [23]—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 [85], 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 [74], we discuss how MOESI's elimination of downgrade writebacks is critical in preventing a source of coherence-induced hammering in §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 [42, 83]) bandwidth, both Intel and AMD have opted to reduce snoop traffic by defaulting to directory ccNUMA protocols since at least 2017 [23, 47, 64, 78].

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 [79, 120]. 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.

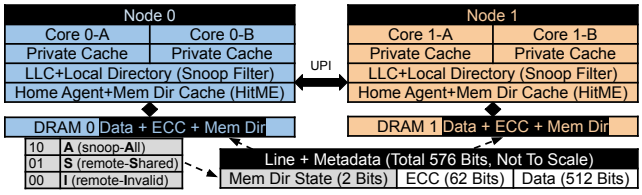


Figure 1: A simplified Intel Skylake ccNUMA system. Each line maps to a home agent that maintains coherence across nodes using distributed state. Local state is stored in the LLC+local directory (snoop filter). Remote state is stored in a line's *memory directory* bits. Select remote state is also stored in an on-die directory cache (HitME [80]) to reduce snoop latency.

As shown in Fig. 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 [80]: *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 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 [24, 110]).

To avoid this repeated penalty, each home agent uses an on-die memory directory cache [80, 82] (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.

3 COHERENCE-INDUCED HAMMERING

In this section, we describe how we discovered sources of *coherence-induced hammering* in Intel's ccNUMA protocols. These phenomena are the first examples known to cause *commodity* workloads to exhibit dangerously-high row ACT rates. We 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 [50]) before any bits may flip in victim row(s). Recent work [21, 58, 107] 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.

3.1 Introduction and Methodology

We 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 [26] 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.

We run Ubuntu Linux 20.04 with KVM [62] 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, we provide evidence of hammering in two different cloud workloads (*memcached* [52] and *terasort* [87]), based on internal benchmarks used by the cloud provider. We show that PARSEC 3.0 [123] and SPLASH-2x [117] benchmarks exhibit similar behavior in §6.

For *memcached* and *terasort*, we record at least 10 seconds of execution per trace, given bus analyzer storage limits. We 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. We 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 we can only record traffic to 1 DIMM out of the many DIMMs used by a workload in a production machine.

3.2 Source #1: Downgrade Writebacks

To determine the root cause(s) of hammering in commodity workloads, we conducted further analysis of the DDR4 access traces. We 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.

	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 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.

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 [85], incurred in MESI-based protocols (i.e., Intel server protocols [37, 53]). At a high level, the writeback occurs when a dirty line is shared with another cache, such as producer-consumer sharing [19] (repeated writer-reader).

For instance, in Fig. 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 (§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** → **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, we pinned our workloads to a single node—so that downgrade writebacks would go to the node's LLC, not DRAM—and recorded new traces. As shown in Fig. 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, we 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, we 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

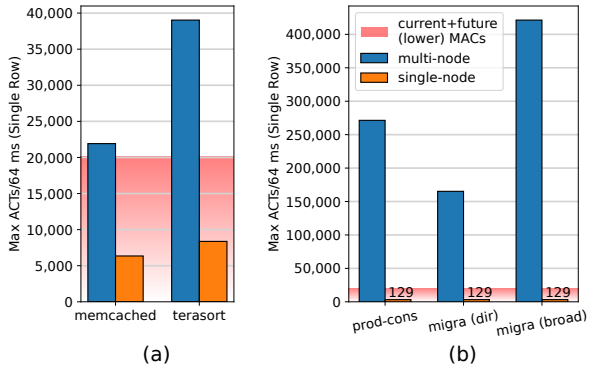


Figure 3: Activation (ACT) rates on a major cloud provider's production hardware for (a) commodity benchmarks and (b) worst-case micro-benchmarks. In both cases, dirty sharing across NUMA nodes yields ACTs in excess of current Rowhammer thresholds (MACs).

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, we 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 (§2.1). We 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. 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× 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, we 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.

3.3 Source #2: Memory Directory Writes

To determine if downgrade writebacks are the *only* source of coherence-induced hammering on Intel ccNUMA servers, we 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 [24, 110], 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**, we 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 **M** → **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 §6 confirm this reasoning.

As with *prod-cons*, we run *migra* with the threads scheduled on different nodes and on the same node. We 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. 3(b) shows that the multi-node experiment *still hammers* (165,233 ACTs). Furthermore, we 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.

We discovered that others had also reported unusually-high DRAM writes on Intel Skylake servers [46], and suspected memory directory (§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 [78, 79] (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 [80] (akin to write-through), even directory cache allocations immediately incur DRAM writes.

We 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. We found that the *write*-based hammering was eliminated when executing *migra (broad)* across NUMA nodes. We 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.

3.4 Source #3: Speculative Reads

While we no longer observed write-based hammering, we instead observed *read*-based hammering caused by the same lines in *migra (broad)*—421,360 ACTs in Fig. 3(b). In fact, we 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, we disabled all prefetchers listed in the BIOS, but still observed repeated reads of the lines. Thankfully, prior work [46, 78] 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), we 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 (§2.3). Thus, we infer that the remaining reads appear to indicate directory cache *misses*. Because our micro-benchmark only migrates two lines between the nodes, we find it unlikely that the misses arise from set conflicts in the directory cache (i.e., conflict misses).

Instead, we believe a phenomenon similar to a documented [23] 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 [80] 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, we 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, we conclude that coherence-induced speculative DRAM reads can occur in commodity broadcast *and* directory ccNUMA protocols, irrespective of the use of MESI or MOESI.

3.5 Why This Hammering is Problematic

Commodity workloads producing ACT rates known to induce bit flips [21, 58] is a significant cause for concern among cloud providers for several reasons. First, recent studies of data center reliability (e.g., Facebook [28] and Google [43]) 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 [58]. 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 [21, 58, 107], 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 [30, 41, 49, 88, 94, 114] already exploit as few as 3 rows simultaneously surpassing MACs in order to overwhelm existing mitigations (TRR, §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 [7, 77, 91, 105, 121] 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 [58, 71] 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.

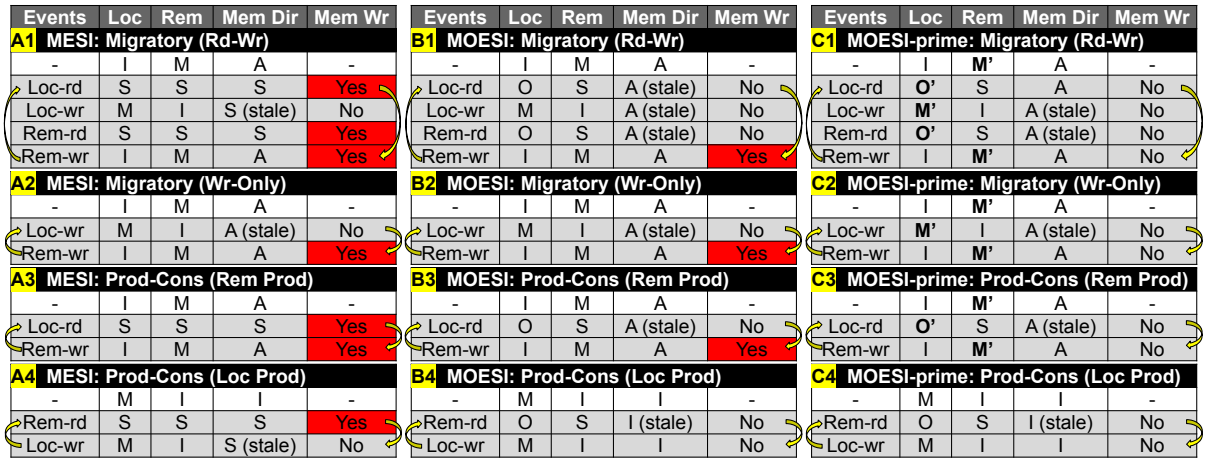
4 DESIGN OF MOESI-PRIME

In light of modern ccNUMA protocols' susceptibility to coherence-induced hammering, we 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. We build atop the MOESI states, given MESI's susceptibility to hammering downgrade write-backs (§3.2). We describe MOESI-prime's novel mechanisms to prevent both hammering directory writes (§4.1) as well as hammering speculative reads (§4.2), and discuss a safe protocol performance optimization (§4.3).

4.1 Preventing Hammering Directory Writes

As discerned in §3.3, hammering directory writes occur during repeated *dirty* sharing across nodes (i.e., sharing with at least one



Events	Loc	Rem	Mem Dir	Mem Wr
A1 MESI: Migratory (Rd-Wr)				
-	I	M	A	-
Loc-rd	S	S	S	Yes
Loc-wr	M	I	S (stale)	No
Rem-rd	S	S	S	Yes
Rem-wr	I	M	A	Yes
A2 MESI: Migratory (Wr-Only)				
-	I	M	A	-
Loc-wr	M	I	A (stale)	No
Rem-wr	I	M	A	Yes
A3 MESI: Prod-Cons (Rem Prod)				
-	I	M	A	-
Loc-rd	S	S	S	Yes
Rem-wr	I	M	A	Yes
A4 MESI: Prod-Cons (Loc Prod)				
-	M	I	I	-
Rem-rd	S	S	S	Yes
Loc-wr	M	I	S (stale)	No

Events	Loc	Rem	Mem Dir	Mem Wr
B1 MOESI: Migratory (Rd-Wr)				
-	I	M	A	-
Loc-rd	O	S	A (stale)	No
Loc-wr	M	I	A (stale)	No
Rem-rd	O	S	A (stale)	No
Rem-wr	I	M	A	Yes
B2 MOESI: Migratory (Wr-Only)				
-	I	M	A	-
Loc-wr	M	I	A (stale)	No
Rem-wr	I	M	A	Yes
B3 MOESI: Prod-Cons (Rem Prod)				
-	I	M	A	-
Loc-rd	O	S	A (stale)	No
Rem-wr	I	M	A	Yes
B4 MOESI: Prod-Cons (Loc Prod)				
-	M	I	I	-
Rem-rd	O	S	I (stale)	No
Loc-wr	M	I	I	No

Events	Loc	Rem	Mem Dir	Mem Wr
C1 MOESI-prime: Migratory (Rd-Wr)				
-	I	**M'**	A	-
Loc-rd	**O'**	S	A	No
Loc-wr	**M'**	I	A (stale)	No
Rem-rd	**O'**	S	A (stale)	No
Rem-wr	I	**M'**	A	No
C2 MOESI-prime: Migratory (Wr-Only)				
-	I	**M'**	A	-
Loc-wr	**M'**	I	A (stale)	No
Rem-wr	I	**M'**	A	No
C3 MOESI-prime: Prod-Cons (Rem Prod)				
-	I	**M'**	A	-
Loc-rd	**O'**	S	A (stale)	No
Rem-wr	I	**M'**	A	No
C4 MOESI-prime: Prod-Cons (Loc Prod)				
-	M	I	I	-
Rem-rd	O	S	I (stale)	No
Loc-wr	M	I	I	No

Figure 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 §4.3.

writer). Thus, MOESI-prime's goal is to obviate the need for directory writes during repeated dirty sharing. We compare this approach to a writeback directory cache—which would at-best reduce the frequency of these (as we will show, unnecessary) writes—in §7.2.

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

4.1.1 Local-Remote Sharing. There are two basic forms of repeated dirty sharing: migratory [24, 110] (writer-writer) and producer-consumer [19] (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 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. We 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.

4.1.2 Remote-Remote and > 2-Node Sharing. Fig. 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 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.

4.2 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 (§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 [80]. 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 we will show in §6, MOESI-prime has negligible effect on performance versus MESI and MOESI baselines, despite this modest increase in contention.

We 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, we 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 [13, 93, 96, 97, 104, 116], which are complementary to MOESI-prime's protection against coherence-induced hammering.

4.3 Optimization: Greedy Local Ownership

Prior work [3, 81] 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. 2 (§2.3), the responder (C0) retains ownership (**M** → **O**) while the requestor (C1) enters **S**. Conversely, in AMD's "Always-Migrate" ownership policy [68], the responder C0 relinquishes ownership (**M** → **S**), and the requestor C1 acquires ownership (enters **O**).

We 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. 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 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 §6, we show that these protections prevent coherence-induced hammering across a broad range of non-malicious and malicious workloads.

5 PROTOCOL CORRECTNESS

In this section, we demonstrate that MOESI-prime's two key protocol extensions—the prime (**M'** and **O'**) states and directory cache modifications—preserve coherence. We 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.

We assume an Intel-like MESI baseline can be extended with the widely-used **O** state to form an otherwise-identical MOESI protocol. We thus reason about MOESI-prime in the context of a MOESI baseline. We also assume that the MOESI baseline correctly implements greedy local ownership (§4.3), noting that prior work [3, 68, 81] demonstrates the validity of denoting either the requestor or responder as the owner.

We 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 [72].

5.1 Correctness of M' and O' States

For this proof, we model ccNUMA systems as transition systems [5] 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. We 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, we 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, we 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, we 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 we 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.■

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 (§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 [80]), 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

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 [103] scheduling, RoCoRaBaCh [40] 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 1: gem5 simulation configuration.

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.

6 EVALUATION

We evaluate MOESI-prime in gem5 v21.1.0.2 [9, 73]. We extend an existing directory coherence protocol in the Ruby subsystem to model Intel's memory directory and directory cache. We 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, we use simple in-order cores atop detailed cache, coherence, and DRAM models. This configuration follows prior work [35], which demonstrates that out-of-order versus in-order execution does not significantly affect the memory system characterization of commodity workloads, given the detailed memory system model.

We 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 1.

We 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 (§4.3). We 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.

We run 8-thread (1 per core) benchmarks from the PARSEC 3.0 [123] and SPLASH-2x [117] suites, simulating the region-of-interest for each benchmark with the *simmedium* input size. We omit 3/26 benchmarks due to runtime errors on real hardware (*fmm* [106]) and use of unsupported x86-64 instructions in gem5 (*volrend* and *x264* [32, 33, 73]). We are unable to additionally simulate *memcached* and *terasort* (§3.1) due to lack of functional IP networking in gem5 for our configuration. For malicious workloads, we use producer-consumer (*prod-cons*, §3.2) and migratory sharing (*migra*, §3.3) micro-benchmarks that trigger coherence-induced hammering in the baseline protocols.

6.1 Highest Activation Rate

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

6.1.1 Non-Malicious Workloads. Fig. 5 depicts the highest ACT rates for each PARSEC 3.0 and SPLASH-2x benchmark, as well as the arithmetic mean per configuration.

We 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), we find that coherence-induced ACTs are still the dominant source for the maximally-activated rows (85.29%, 74.85%, and 53.31%).

We 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.

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

6.1.2 Malicious Workloads. We 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× improvement, and the hottest rows are *not* those of the

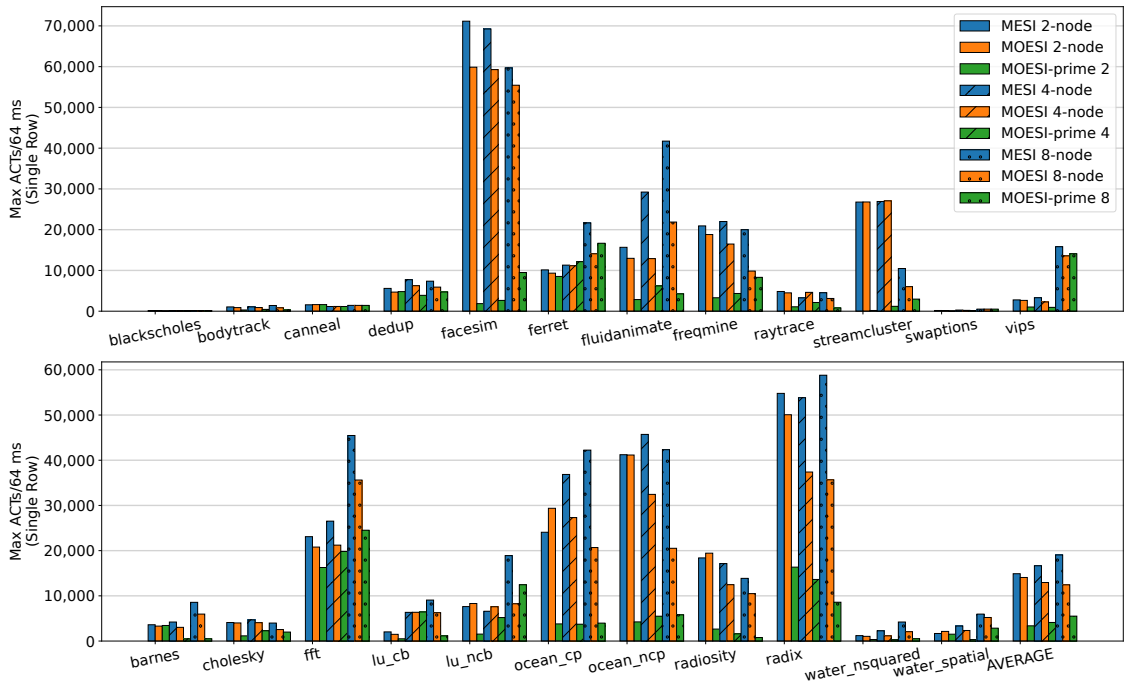


Figure 5: Highest ACT rates for PARSEC 3.0 [123] and SPLASH-2x [117] benchmarks across MESI, MOESI, and MOESI-prime.

shared cache lines (meaning MOESI-prime prevents hammering of the contended rows).

6.2 Performance

We depict MOESI-prime's and MOESI's per-benchmark execution speedup across 2-, 4-, and 8-node configurations in Table 2 (§6.2), normalized to respective MESI baselines. We 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.

Finally, we note that the performance of benchmarks such as *dedup* and *ferret* is particularly-sensitive to thread scheduling [8]. Given scheduling is altered both by different NUMA configurations and protocol timings, such sensitivity can lead to higher performance variability.

6.3 DRAM Power

We assess MOESI-prime's effects on DRAM power consumption using gem5's support for DRAMPower [17], comparing to 2-, 4-, and 8-node MOESI and MESI protocols in Table 2 (§6.3). We 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).

6.4 Scalability

We measure each protocol's scalability by comparing its performance in all 4- and 8-node configurations to its 2-node baseline in Table 2 (§6.4). Each protocol exhibits negligible (within ±1%) differences in scalability across evaluated configurations. We conclude that MOESI-prime offers similar scalability to MESI and MOESI.

7 DISCUSSION

7.1 Broader Applicability

Coherence-induced hammering occurs during commodity workload execution on broadcast and memory directory Intel ccNUMA protocols, with AMD documentation [23] indicating similar coherence-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 [1, 10, 45, 86], the chiplets in even a *single* socket will form a ccNUMA system, requiring careful design to avoid coherence-induced hammering. Additionally, given heterogeneous coherence [2, 92, 109] 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.

§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

§6.3: Power Saved			§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 2: Protocols' MESI-normalized execution speedups (§6.2), average DRAM power savings (§6.3), and 2-node- (2n-) normalized execution speedup (scalability, §6.4). Higher is better in each subtable. "Prime" is MOESI-prime.

7.2 Limitations of a Writeback Directory Cache

Recall that the directory cache uses a write-on-allocate policy (§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.

7.3 Considerations for Other Hammering

MOESI-prime mitigates the reliability and security threat of coherence-induced hammering, but other forms of hammering remain. To our knowledge, all other existing hammering patterns [22, 25, 30, 38, 39, 41, 48, 49, 51, 58, 61, 65, 70, 84, 88, 94, 95, 101, 108, 111, 113, 114, 119] 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. [20] exploit a hammering phenomenon related to the speculative reads found in ccNUMA protocols (§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.

8 RELATED WORK

Rowhammer. Rowhammer bit flips were disclosed in 2014 on DDR3 DRAM [61] and followed by attacks across a variety of DRAM technologies, such as DDR4, LPDDR4/5, and HBM. Prior work [54–56] 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 [30, 41, 49, 88, 94, 114]. Recent work [21, 58, 107] shows that newer DRAM is increasingly susceptible to Rowhammer, and that proposed mitigations [57, 66, 122] will incur increasing performance overhead with rising susceptibility (i.e., decreasing MACs, §3). Follow-up state-of-the-art mitigations [7, 77, 91, 105, 121] 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 [31] 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 [99], feedback-driven page placement [75, 76], NUMA-optimized locks [15, 27], faster barriers [18], and speculative lock elision [98]. 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 [67]. AMD [68] and Improved-MOESI [3] 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 [19] and migratory [24, 110] sharing. MOESI-prime complements these techniques, preventing such sharing from hammering DRAM.

Protocol Generation. ProtoGen [89] and HieraGen [90] automatically generate correct-by-design protocols from stable state specifications. To our knowledge, no support yet exists to automatically generate memory directory ccNUMA protocols.

9 CONCLUSION

In this work, we 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, we 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback, as well as Lucian Cojocar, Ishwar Agarwal, Daniel Berger, Tanj Bennett, Tim Cowles, Brett Dodds, Todd Farrell, Adam Grenzebach, Terry Grunzke, Mark Hill, Lisa Hsu, Todd Merritt, and Nicolai Oswald for many helpful discussions. Kevin Loughlin has been supported by an NSF Graduate Research Fellowship (award DGE 1256260) and a Google PhD Fellowship.

REFERENCES

[1] Paul Alcorn. 2022. New UCIe Chiplet Standard Supported by Intel, AMD, and Arm. tomshardware.com/news/new-ucie-chiplet-standard-supported-by-intel-amd-and-arm.
[2] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.
[3] Hesham Altwaijry and Diyab S Alzahrani. 2014. Improved-moesi cache coherence protocol. *Arabian Journal for Science and Engineering* (2014).
[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ACM SIGARCH Computer Architecture News (CAN)*.
[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.
[6] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. 2015. Row hammer refresh command. US Patent 9,117,544.
[7] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. 2021. Panopticon: A Complete In-DRAM Rowhammer Mitigation. In *Workshop on DRAM Security (DRAMSec)*.
[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News (CAN)* (2011).
[10] Arijit Biswas. 2021. Sapphire Rapids. In *IEEE Hot Chips Symposium (HCS)*.
[11] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[12] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks. In *ACM Asia Conference on Computer and Communications Security (Asia CCS)*.
[13] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.
[14] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security Symposium (USENIX Security)*.
[15] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*.
[16] A. Chakraborty, M. Alam, and D. Mukhopadhyay. 2019. Deep Learning Based Diagnostics for Rowhammer Protection of DRAM Chips. In *IEEE Asian Test Symposium (ATS)*.
[17] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. drampower.info.
[18] Liqun Cheng and John B Carter. 2005. Fast barriers for scalable ccnuma systems. In *IEEE International Conference on Parallel Processing (ICPP)*.
[19] Liqun Cheng, John B Carter, and Donglai Dai. 2007. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[20] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. 2020. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *IEEE Symposium on Security and Privacy (S&P)*.
[21] Lucian Cojocar, Kevin Loughlin, Stefan Saroiu, Baris Kasikci, and Alec Wolman. 2021. mFIT: A Bump-in-the-Wire Tool for Plug-and-Play Analysis of Rowhammer Susceptibility Factors. *Microsoft Tech Report* (2021).
[22] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *IEEE Symposium on Security and Privacy (S&P)*.
[23] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *ACM/IEEE International Symposium on Microarchitecture (MICRO)* (2010).
[24] Alan L Cox and Robert J Fowler. 1993. Adaptive cache coherency for detecting migratory shared data. *ACM SIGARCH Computer Architecture News (CAN)* (1993).

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

[26] Timothy J Dell. 1997. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics division* (1997).

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

[28] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. *arXiv preprint arXiv:2102.11245* (2021).

[29] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. 2022. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

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

[31] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *EuroSys*.

[32] gem5. 2021. Architecture Support. gem5.org/documentation/general_docs/architecture_support/.

[33] gem5. 2021. gem5-20 Working Status of Benchmarks. gem5.org/documentation/benchmark_status/gem5-20.

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

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

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

[37] JR Goodman and HHJ Hum. 2004. 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).

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

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

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

[41] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[42] Andrew Hay. 2012. *MESIF Cache Coherence Protocol.* Ph. D. Dissertation. ResearchSpace@ Auckland.

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

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

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

[46] Intel Community Forum. 2019. SKL - strange memory behavior.

[47] Intel Xeon Processor Scalable Memory Family. 2017. Uncore Performance Monitoring Reference Manual. *Intel Corporation* (2017).

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

[49] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *IEEE Symposium on Security and Privacy (S&P)*.

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

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

[52] 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. 2011. Memcached design on high performance rdma capable interconnects. In *IEEE International Conference on Parallel Processing (ICPP)*.

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

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

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

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

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

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

[59] M. Kim, J. Choi, H. Kim, and H. Lee. 2019. An Effective DRAM Address Remapping for Mitigating Rowhammer Errors. *IEEE Trans. Comput.* (2019).

[60] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2021. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. *arXiv preprint arXiv:2108.06703* (2021).

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

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

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

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

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

[66] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[67] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. *ACM SIGARCH Computer Architecture News (CAN)* (1990).

[68] Kevin M Lepak, Vydhyanathan Kalyanasundharam, William A Hughes, Benjamin Tsien, and Greggory D Donley. 2014. Method and apparatus for accelerated shared data migration. US Patent 8,732,410.

[69] C. Li and J. Gaudiot. 2019. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*.

[70] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. 2020. Nethammer: Inducing Rowhammer faults through network requests. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.

[71] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. 2021. Stop! Hammer time: rethinking our approach to rowhammer mitigations. In *Workshop on Hot Topics in Operating Systems (HotOS)*.

[72] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. 2022. MOESI-prime source code. github.com/efeslab/moesi-prime.

[73] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[74] Neethu Bal Mallya, Geeta Patil, and Biju Raveendran. 2015. Simulation based performance study of cache coherence protocols. In *International Symposium on Nanoelectronic and Information Systems*.

[75] Jaydeep Marathe and Frank Mueller. 2006. Hardware profile-guided automatic page placement for ccNUMA systems. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*.

[76] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Parallel and Distrib. Comput.* (2010).

[77] Michele Marazzi, Patrick Jattke, Solt Flavien, and Kaveh Razavi. 2022. PROTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *IEEE Symposium on Security and Privacy (S&P)*.

[78] John McCalpin. 2018. Topology and Cache Coherence in Knights Landing and Skylake Xeon Processors. *UT Faculty/Researcher Works* (2018).

[79] John McCalpin. 2020. Directory Structure in Skylake Server CPUs. *Intel Community Forum* (2020).

[80] Adrian C Moga, Malcolm Mandviwalla, Vedaraman Geetha, and Herbert H Hum. 2013. Allocation and write policy for a glueless area-efficient directory cache for hotly contested cache lines. US Patent 8,392,665.

[81] Daniel Molka, Daniel Hackenberg, and Robert Schöne. 2014. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Workshop on Memory Systems Performance and Correctness*.

[82] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. 2015. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. In *IEEE International Conference on Parallel Processing (ICPP)*.

[83] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. *Intel Corporation* (2017).

[84] Onur Mutlu. 2017. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[85] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. 2020. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* (2020).

[86] 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. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *IEEE International Solid-State Circuits Conference (ISSCC)*.

[87] J Norris. 2013. Package org.apache.hadoop.examples.terasort.

[88] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S Kim, and Onur Mutlu. 2021. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chipsand Implications on Future Attacks and Defenses. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[89] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. 2018. ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[90] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. 2020. HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[91] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[92] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[93] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P)*.

[94] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. 2021. Introducing half-double: New hammering technique for dram rowhammer bug. *Google Security Blog*.

[95] Rui Qiao and Mark Seaborn. 2016. A new approach for Rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[96] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[97] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[98] Ravi Rajwar and James R Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[99] Sabela Ramos and Torsten Hoefler. 2015. Cache line aware optimizations for cc-NUMA systems. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[100] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. 2013. Optimizing virtual machine scheduling in NUMA multicore systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[101] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium (USENIX Security)*.

[102] TIRIAS Research. 2020. Second Generation AMD EPYC Processor Enhanced Cache and Memory Architecture.

[103] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. *ACM SIGARCH Computer Architecture News (CAN)* (2000).

[104] Gururaj Saileshwar and Moinuddin Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium (USENIX Security)*.

[105] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. 2022. 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)*.

[106] Ciro Santilli. 2020. PARSEC Benchmark . github.com/cirosantilli/parsec-benchmark/.

[107] Stefan Saroiu, Alec Wolman, and Lucian Cojocar. 2022. The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses. In *International Reliability Physics Symposium (IRPS)*.

[108] Mark Seaborn and Thomas Dullien. 2015. 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.

[109] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache coherence for GPU architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[110] Per Stenström, Mats Brorsson, and Lars Sandberg. 1993. An adaptive cache coherence protocol optimized for migratory sharing. *ACM SIGARCH Computer Architecture News (CAN)* (1993).

[111] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Defeating software mitigations against Rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.

[112] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. 2022. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *IEEE Symposium on Security and Privacy (S&P)*.

[113] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *ACM SIGSAC conference on computer and communications security (CCS)*.

[114] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[115] 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. 2020. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[116] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium (USENIX Security)*.

[117] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH Computer Architecture News (CAN)* (1995).

[118] Xin-Chuan Wu, Timothy Sherwood, Frederic T Chong, and Yanjing Li. 2019. 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)*.

[119] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium (USENIX Security)*.

[120] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy (S&P)*.

[121] 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. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[122] Jung Min You and Joon-Sung Yang. 2019. MRLoc: Mitigating Row-hammering based on memory Locality. In *ACM/IEEE Design Automation Conference (DAC)*.

[123] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC 3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News (CAN)* (2017).