Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention

Michael G. Bechtel, Heechul Yun
University of Kansas, USA.
{mbechtel, heechul.yun}@ku.edu

***Abstract*—In this paper we investigate the feasibility of denial-of-service (DoS) attacks on shared caches in multicore platforms. With carefully engineered attacker tasks, we are able to cause more than 300X execution time increases on a victim task running on a dedicated core on a popular embedded multicore platform, regardless of whether we partition its shared cache or not. Based on careful experimentation on real and simulated multicore platforms, we identify an internal hardware structure of a non-blocking cache, namely the cache writeback buffer, as a potential target of shared cache DoS attacks. We propose an OS-level solution to prevent such DoS attacks by extending a state-of-the-art memory bandwidth regulation mechanism. We implement the proposed mechanism in Linux on a real multicore platform and show its effectiveness in protecting against cache DoS attacks.**

***Keywords*-Denial-of-Service Attack, Shared Cache, Multicore, Real-time sytems**

I. Introduction

Efficient multi-core computing platforms are increasingly demanded in embedded real-time systems, such as cars, to improve their intelligence while meeting cost, size, weight and power constraints. However, because many tasks in such a system may have strict real-time requirements, unpredictable timing of multicore platforms due to contention in the shared memory hierarchy is a major challenge [13]. Moreover, timing variability in multicore platforms can potentially be exploited by attackers.

For example, as cars are becoming more connected, there are increasing possibilities for an attacker to execute malicious programs inside of a car's computer [10], such as via downloaded 3rd party apps. Even if the computer's runtime (OS and hypervisor) strictly partitions cores (and memory) to isolate these potentially dangerous programs from critical tasks, as long as they share the same multicore computing platform, an attacker may still be able to cause significant timing influence to the critical tasks—simply by accessing shared hardware resources, such as a shared cache, at a high-speed, effectively mounting *denial-of-service (DoS) attacks*.

On modern multicore processors, non-blocking caches [24] are commonly used, especially as shared last-level caches, to support concurrent memory accesses from multiple cores. However, access to a non-blocking cache can be blocked when any of the cache's internal hardware buffers become full, after which the cache will deny any further requests from the CPU—even if the request is actually a cache-hit—until the internal buffers become available again [2], [35]. On a shared cache, such blocking globally affects *all* cores because none of the cores can access the cache until it is unblocked, which can take a long time (e.g., hundreds of CPU cycles) as it may need to access slow main memory. Therefore, if an attacker can intentionally induce shared cache blocking, it can cause significant timing impacts to other tasks on different cores. This is because every shared cache hit access, which would normally take a few cycles, would instead take a few hundreds cycles until the cache is unblocked.

While most prior works on cache isolation focused on cache space partitioning, either in software or in hardware [9], [16]–[19], [25], [27], [37], [43], [44], these cache partitioning techniques are ineffective in preventing such shared cache blocking because, even if the cache space is partitioned, the cache's internal buffers may still be shared. A recent study [39] experimentally showed the ineffectiveness of cache partitioning in preventing shared cache blocking on a number of out-of-order multicore platforms. Specifically, when the miss-status-holding-registers (MSHRs) [24]—which track outstanding cache-misses—of a shared cache are exhausted, consequent cache blocking can cause substantial—reportedly up to 21X—task execution time increases, even when the task runs alone on a dedicated core with a dedicated cache partition and its working-set fits entirely in its cache partition [39].

In this paper, we first experimentally investigate the feasibility and severity of cache DoS attacks on shared caches on contemporary embedded multicore platforms: five quad-core CPU implementations, out of which four are in-order and one is out-of-order architecture. Our first main finding is that extreme shared cache blocking can occur not only in out-of-order processors, as suggested in [39], but also in *in-order* processors. In fact, we observe the most severe execution time increase—*up to 346X* (times, not percent)—on a popular in-order architecture based quad-core platform (Raspberry Pi 3). This is surprising because it was believed that simpler in-order architectures are less susceptible to memory contention than out-of-order architectures [38], [39].

We use a cycle-accurate full system simulator (Gem5 [8] and Ramulator [22] for CPU and memory, respectively) to identify possible causes of such severe timing impacts of cache DoS attacks under various microarchitectural configurations. Our findings include: (1) eliminating MSHR contention alone is not sufficient to mitigate potential cache DoS attacks because another cache internal hardware structure, namely the *writeback buffer* [35]—which is used to temporarily store evicted cache-lines—in a shared cache can be an important

additional source of cache blocking; (2) the combination of a small cache writeback buffer and the presence of aggressive hardware prefetchers can cause severe writeback buffer contention, and subsequent cache blocking.

We propose an OS-level solution to mitigate shared cache DoS attacks that target cache writeback buffers. Our solution is based on MemGuard [48], which is a Linux kernel module that regulates (throttles) each core's maximum memory bandwidth usage to a certain threshold value at a regular interval (e.g., 1ms) using a hardware performance counter. Our extension is to apply two separate regulations—one for read (cache-line refills) and one for write (cache write-backs) memory traffic—for each core. This allows us to set a high threshold value for read bandwidth while setting a low threshold value for write bandwidth. This mitigates writeback buffer DoS attacks with minimal performance impacts for normal, non-attacker applications, which are typically more sensitive to read performance [14], [35].

Our solution is implemented on a real multicore platform and evaluated against cache DoS attack programs that generate very high write traffic (to overflow shared cache writeback buffers.) The results show that it is effective in preventing such cache DoS attacks while minimizing the throughput loss of the prior memory bandwidth throttling approach.

This paper makes the following **contributions**:

- We experimentally demonstrate the feasibility of shared cache DoS attacks on a number of contemporary embedded multicore platforms. In particular, we show that even relatively simple in-order multicore architectures can also be highly affected by such microarchitectural attacks.
- We provide detailed microarchitectural analysis as to why these cache DoS attacks are effective in modern multicore architectures featuring non-blocking caches and hardware prefetchers. In particular, we identify the writeback buffer of a shared cache as a potential attack vector that enables shared cache DoS attacks.
- We propose an OS-level solution to mitigate DoS attacks targeting a shared cache's writeback buffer. The proposed OS solution is shown to be effective in mitigating such attacks. We also provide it as open-source[1].

The remainder of this paper is organized as follows: Section II provides necessary background information on non-blocking caches and hardware prefetchers. Section III defines the threat model. Section IV shows the feasibility of shared cache DoS attacks on contemporary embedded multicore platforms. Section V validates the problem of writeback buffer induced shared cache blocking using a simulated multicore platform. Section VI presents our software solution to counter shared cache DoS attacks. We discuss related work in Section VII and conclude in Section VIII.

II. Background

In this section, we provide background on non-blocking caches and hardware prefetchers.

[1]https://github.com/mbechtel2/memguard.

A. Non-blocking Cache

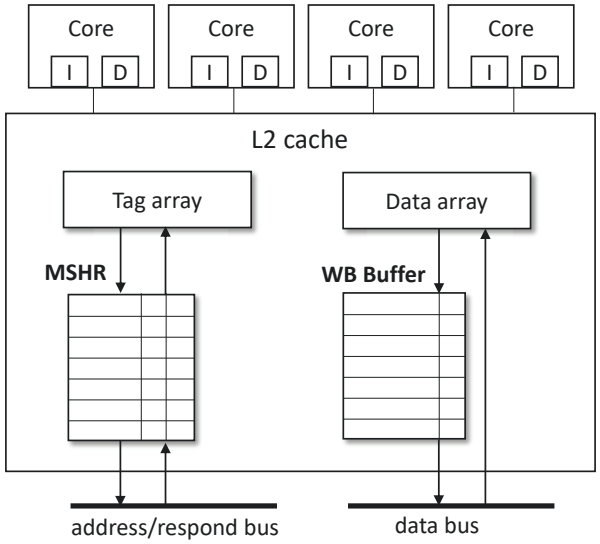


Fig. 1: Internal organization of a shared L2 cache. Adopted from Figure 11.10 in [35].

Modern processors employ non-blocking caches to improve cache-level parallelism and performance. On a non-blocking cache, access to the cache is allowed even while it is still processing prior cache miss(es). This non-blocking access capability is essential in a multicore processor, especially for the shared last-level cache, to achieve high performance. Figure 1 shows the internal structure of a non-blocking shared L2 cache of a multicore processor, which depicts its two major hardware structures: *Miss-Status-Holding-Register (MSHR)* and *WriteBack (WB) buffer.*

When a cache-miss occurs on a non-blocking cache, the miss related information is recorded on a MSHR entry. The MSHR entry is cleared once the corresponding cache-line is fetched from the lower-level memory hierarchy (e.g., DRAM controller). In the mean time, the cache can continue to serve concurrent memory requests from the CPU cores or other higher-level caches (e.g., L1 caches). A non-blocking cache can support multiple outstanding cache-misses, although the degree to which this can occur depends on the size of the MSHR structure.

On the other hand, the writeback buffer is used to temporarily store evicted (dirty) cache-lines from the cache while the corresponding demand misses are serviced (cache-line refills). Because cache-line refills (reads from DRAM) are generally more important to application performance, delaying the writebacks reduces bus contention and, therefore, improves performance. In this way, a non-blocking cache can support concurrent access to the cache efficiently most of the time.

Note, however, that when *either* the MSHRs or writeback buffer of a non-blocking cache becomes full, the entire cache is blocked—i.e., it no longer accepts any further requests—until after free entries in both the MSHRs and writeback buffer are available, at which point the cache is said to be unblocked.

Unfortunately, this can take a long time because access to DRAM may take hundreds of CPU cycles, which could take even longer if the DRAM controller is congested. When a shared cache is blocked due to either MSHR or writeback buffer exhaustion, it affects *all* cores. If a task frequently accesses the shared cache, even if the accesses are all cache-hits, the task may still suffer massive execution time increase if the cache is blocked most of the time.

B. Hardware Prefetcher

Cache prefetching is a technique used to reduce cache miss penalties by preemptively loading memory blocks that are likely to be accessed in the near future into the cache. Due to the high cost of a cache miss, successful prefetching can significantly improve performance. Therefore, modern processors often employ multiple hardware prefetchers alongside the caches. A hardware prefetcher monitors access to a cache and predicts future memory addresses based on detected memory access patterns. It then generates a set number of requests that are stored in its internal queue before being sent to the cache. This number is called the prefetcher's *degree* or *depth*. Next-line and stride based prefetchers are the most common. However, while generally effective, prefetching can also incur unnecessary cache-line refills due to mis-predictions, evict useful cache-lines, pollute the cache, and generally add more pressure to the memory hierarchy, which in turn can lead to increased cache blocking as we will show in Section IV.

III. Threat Model

We assume the victim and the attacker are co-located on a multicore processor as shown in Figure 2. We assume the multicore processor has a shared last-level cache as well as per-core private caches. We assume that the runtime (OS and/or hypervisor) provides core and memory isolation between the attacker and the victim. In other words, the attacker cannot run on the same core as the victim and cannot directly access the victim's memory. We henceforth refer to the core that the victim runs on as the victim core, whereas the core that the attacker runs on is the attacker core. In addition, we assume that the runtime can partition cache space between the attacker core and the victim core by means of page coloring [16], [46].

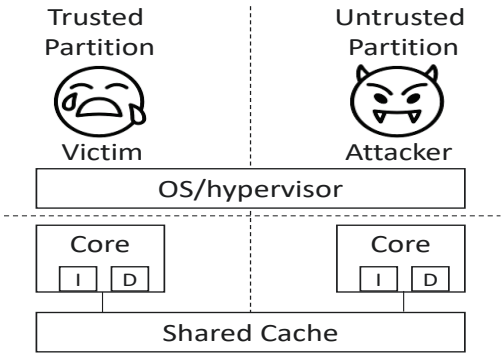


Fig. 2: Threat model.[2]

The only capability the attacker has is to run non-privileged program(s) on the attacker core. In this paper, the attacker's main goal is to generate timing interference to the tasks running on the victim core. On a car, for example, the attacker's goal may be to delay execution of real-time control tasks running on a victim core so that they miss their deadlines, potentially resulting in a crash. The attacker intends to achieve this goal by mounting denial-of-service (DoS) attacks on shared hardware resources, primarily the shared cache.

IV. Shared Cache DoS Attacks on Embedded Multicore Platforms

In this section, we experimentally evaluate the feasibility and significance of cache denial-of-service (DoS) attacks on contemporary embedded multicore platforms.

A. Cache DoS Attack Code

The main objective of the cache DoS attack code is to generate as many concurrent cache-misses on the target cache as quickly as possible. As discussed in Section II, concurrent cache-misses can exhaust available MSHR and writeback buffer resources, and thereby induce cache blocking.

```
for (i = 0; i < mem_size; i += LINE_SIZE)
{
        sum += ptr[i];
}
```

(a) Read attack (BwRead)

```
for (i = 0; i < mem_size; i += LINE_SIZE)
{
        ptr[i] = 0xff;
}
```

(b) Write attack (BwWrite)

Fig. 3: Memory attacks. `LINE_SIZE` = a cache-line size.

Figure 3 shows read and write attack code snippets. The read attack code simply iterates over a single one-dimensional array, at every cache-line (LINE_SIZE) distance, and sums them up. Because *sum* is allocated on a register by the compiler, the code essentially keeps generating memory *load* operations, which may always miss the target cache if the array size is bigger than the cache size. These missed loads will stress the cache's MSHR.

On the other hand, the write attack code performs the same iteration over an array but, instead of reading each array entry, it updates them, thereby generating memory *store* operations. On a writeback cache, each missed store will trigger two memory transactions: one memory read (cache-line fill) and one memory write (writeback of the evicted cache-line). Therefore, these missed stores will stress both the MSHRs and writeback buffer of a cache.

[2]The icons are by icons8: https://icons8.com/

Platform	Raspberry Pi 3	Odroid C2	Raspberry Pi 2	Odroid XU4	
SoC	BCM2837	AmlogicS905	BCM2836	Exynos5422	
CPU	4x Cortex-A53 in-order 1.2GHz	4x Cortex-A53 in-order 1.5GHz	4x Cortex-A7 in-order 900MHz	4x Cortex-A7 in-order 1.4GHz	4x Cortex-A15 out-of-order 2.0GHz
Private Cache	32/32KB	32/32KB	32/32KB	32/32KB	32/32KB
Shared Cache	512KB (16-way)	512KB (16-way)	256KB (8-way)	512KB (16-way)	2MB (16-way)
Memory (Peak BW)	1GB LPDDR2 (8.5GB/s)	2GB DDR3 (12.8GB/s)	1GB LPDDR (8.5GB/s)	2GB LPDDR3 (14.9GB/s)	

TABLE I: Compared embedded multicore platforms.

We henceforth refer to the read attack as BwRead and the write attack as BwWrite. In addition, their array sizes are denoted in parentheses. For example, (LLC) denotes that the attacker's working-set is configured so that it always hits the shared last-level cache but misses the private L1 cache (i.e., less than 1/4 of the LLC cache size but bigger than the L1 cache size). On the other hand, (DRAM) denotes that the working-set size is bigger than the shared LLC size.

B. Embedded Multicore Platforms

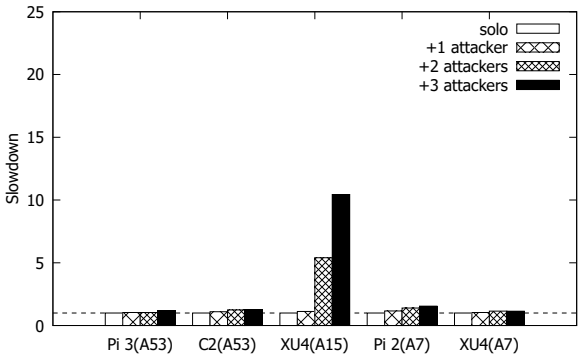
We evaluate the effectiveness of the attacks described above on four embedded multicore platforms: Raspberry Pi 3, Odroid C2, Raspberry Pi 2, and Odroid-XU4. Both the Raspberry Pi 3 and the Odroid C2 employ four Cortex A53 cores, while the Raspberry Pi 2 equips four Cortex A7 cores. The Odroid XU4 has four Cortex-A7 and four Cortex-A15 cores in a "big-little" [12] configuration. Note that Cortex-A15 is a sophisticated out-of-order design [4], while Cortex-A7 and Cortex-A53 are "simpler" in-order designs [5], [6]. In total, we compare five system configurations: four in-order and one out-of-order designs. All platform specifications can be seen in Table I.

C. Synthetic Workloads

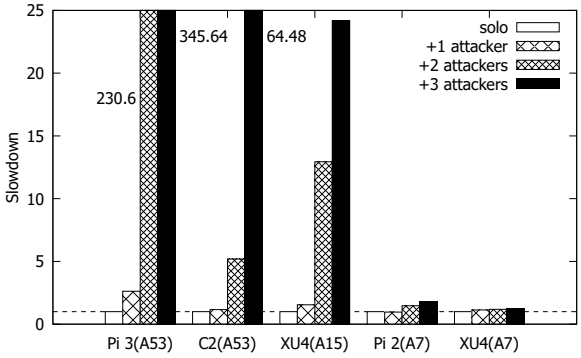
In this experiment, we use a BwRead (LLC) (Figure 3a) as a synthetic victim task and evaluate the feasibility and severity of shared cache DoS attacks.

The basic experiment setup is as follows: we first run the victim task on Core0 and measure its solo execution time. We then co-schedule an increasing number of attacker tasks on the other cores (Core1-3) and measure the response times of the victim task. For attackers, we use both BwRead (DRAM) and BwWrite (DRAM) to stress the L2 MSHR and writeback buffer, respectively.

Figure 4a shows the victim task's performance impact in the presence of BwRead (DRAM) attackers. Note first that Odroid-XU4's Cortex-A15, which is an *out-of-order* architecture based CPU, suffers considerable execution time increases from the read attackers, while the rest of the tested platforms, the four *in-order* architecture based ones, show little performance impacts. This result is consistent with the findings in [39], which suggested that an out-of-order core can generate many concurrent cache accesses and when they all miss the shared cache, due to the attacker's working set size being bigger than the cache, they can cause cache blocking when all MSHR entries are exhausted—i.e., MSHR contention. In [39], it is also suggested that *in-order* cores are less likely to suffer MSHR related cache blocking because an in-order core's ability to generate concurrent memory accesses is limited—that is, one memory access at a time.



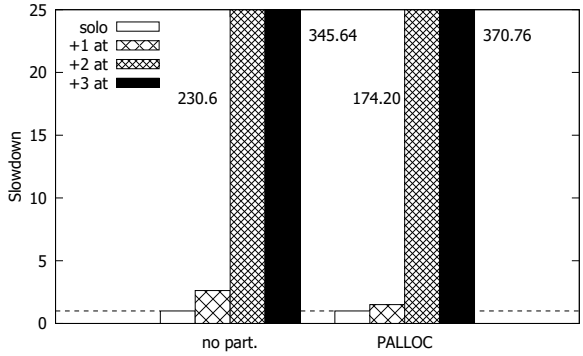
(a) Effects of read attackers: BwRead(DRAM)



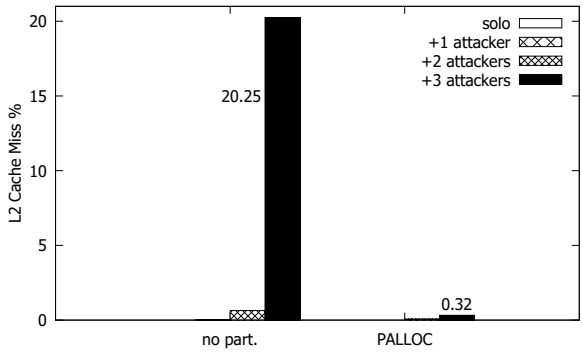
(b) Effects of write attackers: BwWrite(DRAM)

Fig. 4: Effects of memory attackers on a BwRead(LLC) victim. The attackers and victim run on their own dedicated cores.

Figure 4b, which uses BwWrite (DRAM) attackers instead, is therefore *surprising* because we observe extreme performance impacts on two recent in-order architecture based platforms, the Raspberry Pi 3 and Odroid-C2, both of which feature four ARM Cortex-A53 cores. In particular, we observe up to *346X slowdown* on the Raspberry Pi 3 platform, which is, to the best of our knowledge, the highest shared resource contention induced execution time increase ever reported in

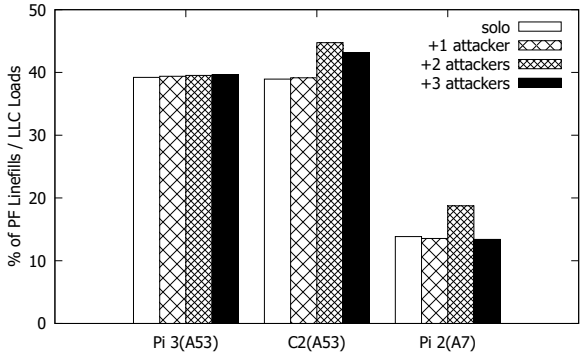


(a) Slowdown

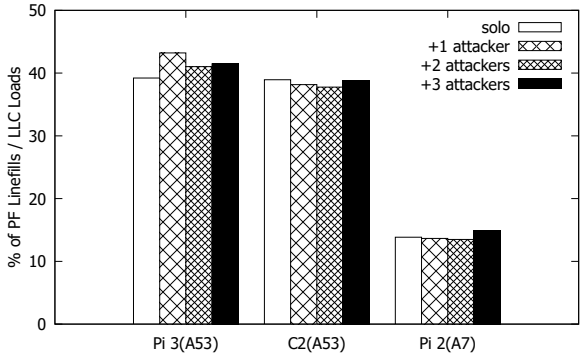


(b) L2 cache miss rate

Fig. 5: Effect of cache partitioning on Raspberry Pi 3. BwRead (LLC) victim vs. BwWrite (DRAM) attackers, w/o and with L2 cache partitioning.



(a) BwRead Co-runners



(b) BwWrite Co-runners

Fig. 6: Percentage of prefetcher linefills over LLC loads.

literature and it is much worse than the slowdown observed on the out-of-order Cortex-A15 in Odroid-XU4. If MSHR contention was the cause of these extreme performance impacts we observed in the two in-order Cortex-A53 platforms, then we would also expect to see considerable performance impacts when the read attackers stressed the MSHRs in the previous experiment, which, however, was not the case.

D. Impact of Cache Partitioning

As reported in [39], we also find that cache partitioning does not help protect the victim's performance even when the victim's working-set size fits entirely in its given dedicated cache partition. Figure 5 shows the impact of cache partitioning. For cache partitioning, we use PALLOC [46], which implements a page coloring based kernel-level memory allocator, to equally partition the L2 cache among the cores in the partitioning setup. Note that the victim task, BwRead (LLC), suffers similar degrees of performance impacts regardless of whether partitioning is applied or not, even while the victim's L2 cache miss rate is significantly reduced with the cache partitioning—from 20.25% to 0.32%—in the presence of three write attackers. In other words, cache partitioning eliminates unwanted cache-line evictions from the attackers but it does not help provide cache performance isolation to the victim, which accesses a dedicated cache space partition.

E. Impact of Hardware Prefetcher

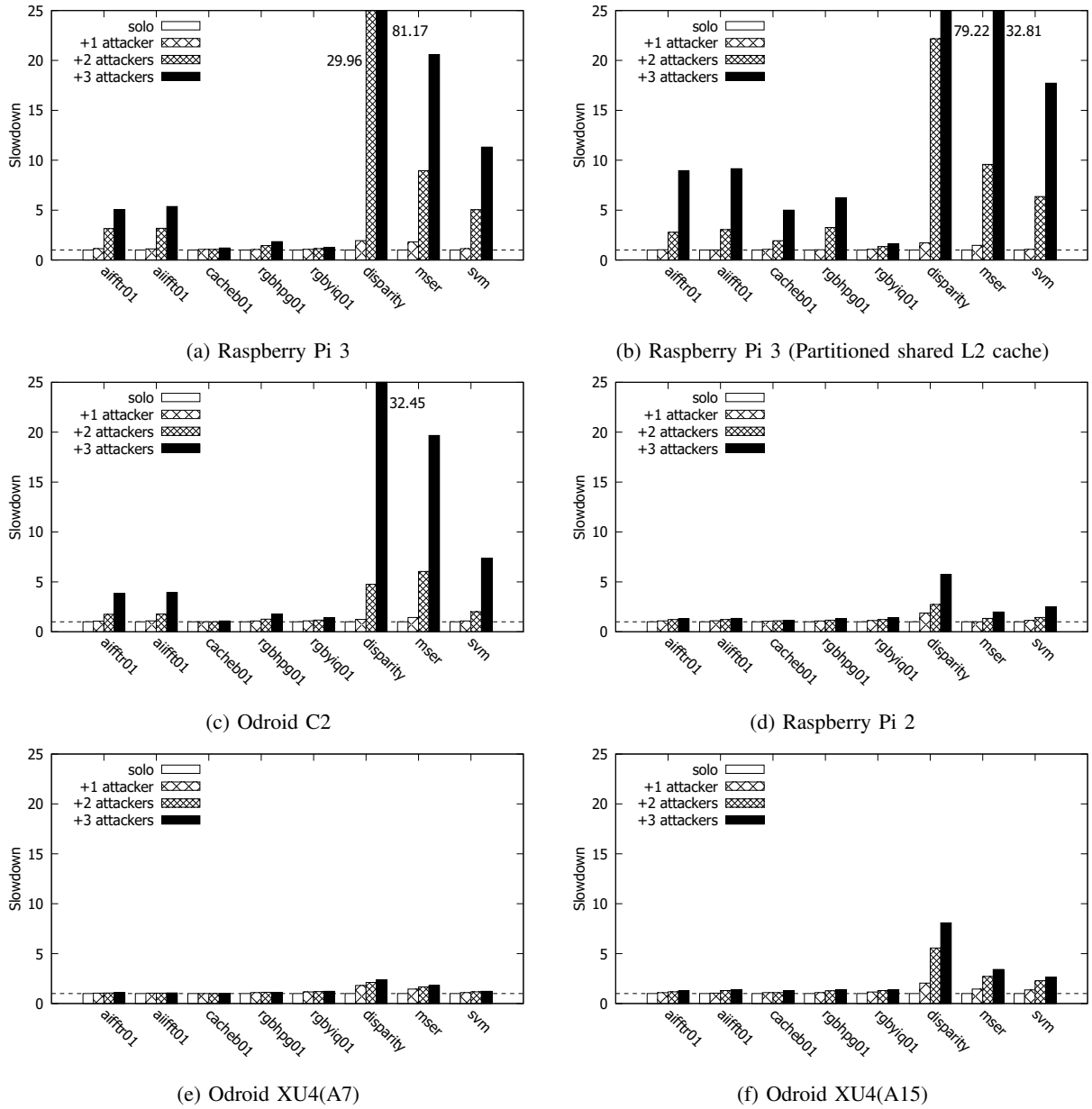
Another interesting observation in Figure 4b is that although both Cortex-A53 and Cortex-A7 are in-order core designs, they show dramatically different behaviors in the presence of the write attackers—Cortex-A53 shows extreme execution time increases while Cortex-A7 shows no significant execution time increase.

To better understand the root cause(s) of this difference, we compared cache related performance counter statistics of the Raspberry Pi 3 and the Odroid C2, both Cortex-A53 based, to the Raspberry Pi 2, which is based on older Cortex-A7.

The basic experiment setup is the same as above: The BwRead(LLC) victim task runs on Core 0 in the presence of an increasing number of BwRead(DRAM) or BwWrite(DRAM) attackers on the rest of the cores.

Our main finding is that all three processors we tested utilize hardware prefetchers on their L1 data caches, but the aggressiveness of the hardware prefetchers varies considerably between the platforms, specifically between the more recent Cortex-A53 and the older Cortex-A7.

Concretely, Figure 6 shows the fraction of the prefetch requests over all LLC memory accesses of the victim task. Note that the L1 prefetchers on the Pi 3 and C2 generate considerably more cache-line refills over total L1-D cache-line refills. On both platforms, the prefetchers account for $\sim$40% of the total cache refills, while the Pi 2's prefetcher only



(a) Raspberry Pi 3

(b) Raspberry Pi 3 (Partitioned shared L2 cache)

(c) Odroid C2

(d) Raspberry Pi 2

(e) Odroid XU4(A7)

(f) Odroid XU4(A15)

Fig. 7: EEMBC and SD-VBS benchmarks vs. BwWrite(DRAM) attackers. From left to right, *aifftr01, aiifft01, cacheb01, rgbhpg01, rgbyiq01* are EEMBC benchmarks and *disparity, mser, svm* are SD-VBS benchmarks. For SD-VBS benchmarks, we use *sqcif* inputs so that their working-sets can mostly fit in the shared L2 cache.

accounts for ∼13%. In other words, the hardware prefetcher of the Cortex-A53 in the Raspberry Pi 3 and Odroid C2 is more aggressive than that of the Cortex-A7 in Raspberry Pi 2.

Note that a L1 prefetcher's data prefetches may be issued concurrently with the core's demand request if the L1 data cache itself is a non-blocking cache. According to the Cortex-A53 documentation [6], its L1 data cache supports up to three outstanding cache-misses, suggesting that may indeed be the case. On the other hand, Cortex-A7's L1 data cache does not appear to support multiple outstanding cache-misses [5]. Thus, we believe that Cortex-A7's prefetcher may only be able to prefetch when the L1 data cache is not being used by the core. This difference in the L1 data cache's supported concurrent outstanding misses is important because cache DoS attacks require concurrent accesses to the shared L2 cache that overflows the cache's internal hardware buffers, namely the MSHRs and writeback buffer.

F. Impact on Real-World Applications

We also use a set of real-world benchmarks from the EEMBC [1] and SD-VBS [42] benchmark suites to investigate impacts of cache DoS attacks on real-world applications. The basic experiment setup is the same as before where we subject each of the tested benchmarks (the victim) to an increasing

number of BwWrite(DRAM) co-runners (the attackers) on different cores of the tested multicore platform.

Figure 7 shows the results. Note, first, that EEMBC benchmarks generally experience much less performance impact than SD-VBS benchmarks. This is because most EEMBC benchmarks do not frequently access the shared L2 cache due to their relatively smaller working-set sizes (which mostly fit in the L1 data cache), while the SD-VBS benchmarks access the shared L2 cache much more frequently due to their larger working-set sizes. Still, on the two Cortex-A53 based platforms, the Raspberry Pi 3 and Odroid C2, even the EEMBC benchmarks suffer up to 5.4X and 3.9X slowdown, respectively. More surprisingly, SD-VBS benchmarks suffer up to 81X slowdown on the Raspberry Pi 3 and up to 32X slowdown on the Odroid C2. In contrast, the Odroid-XU4's Cortex-A15 suffers relatively little as it experiences up to 8X slowdown, which is still significant.

Again, cache partitioning is ineffective in defending against these cache DoS attacks, as shown in Figure 7b. Instead, cache partitioning was actually detrimental when attacker BwWrite tasks were present as 7 of the 8 benchmarks suffered worse slowdowns with partitioning enabled (only the *disparity* benchmark had slightly improved performance).

Lastly, we also evaluate the impacts of BwRead (DRAM) attackers (the read attackers) on these real-world benchmarks. Although we do not include here, due to space considerations, as we observed in experiments using synthetic workloads (Section IV-C), the read attackers have much less performance impact (less than 8% in EEMBC and 2.3X in SD-VBS).

In short, we find that cache DoS attacks, especially the write attackers targeting cache writeback buffers, are highly effective in some in-order architecture based embedded multicore platforms, notably Cortex-A53 based ones. On the other hand, read attackers, which mainly target cache MSHRs, are not effective on the tested in-order multicores while they still have considerable timing impacts in out-of-order architecture based multicore platforms, as suggested in [39].

V. Understanding Shared Cache Blocking Due to Cache Writeback Buffer

In this section, we study writeback buffer induced shared cache blocking using a cycle accurate full system simulator. Specifically, we use Gem5 [8] and Ramulator [22] to model the CPU and the memory subsystem, respectively. Table II shows the baseline configuration we used here.

The simulated CPU we use is comprised of four cores. Each core has its own private (L1) instruction and data caches. The data cache (L1-D) is modeled as a non-blocking cache supporting up to three outstanding cache misses, as found in the Cortex-A53 [6]. The instruction cache (L1-I), on the other hand, supports up to one outstanding cache-miss. The instruction cache is then paired with a tagged prefetcher and the data cache is paired with a stride prefetcher, each of which has its own internal prefetch queue to hold prefetch addresses before they are sent to the respective cache. All cores have access to a single shared L2 cache which has the same queue structures as the L1 caches and, like the L1-D caches, is paired with a stride prefetcher. The shared L2 cache is then connected to a main memory controller (simulated by Ramulator [22]).

Note that we carefully configure the prefetchers and L1 data caches such that cache blocking *cannot* occur due to MSHR contention. That is, our L2 cache has a sufficient number of MSHRs to support up to 24 concurrent cache misses, which is enough to support 12 concurrent requests from the cores (their L1 data caches and prefetchers) and 8 prefetch requests from the L2 prefetcher. In other words, we removed the possibility of MSHR contention, as suggested in [39]. Thus, observed L2 cache blocking, if any, is not caused by MSHR exhaustion, but instead by writeback buffer exhaustion of the L2 cache.

A. Effect of Hardware Prefetchers

In this experiment, we investigate the impacts of L1 and L2 prefetchers on the effectiveness of cache DoS attacks, targeting the cache writeback buffer. The experiment setup is the same as before—specifically, we run the BwRead (LLC) victim and three BwWrite (DRAM) attackers. We repeat the experiment in different L1-D and L2 prefetcher configurations.

Figure 8 shows the results. When the L1-D or L2 prefetchers are enabled (labeled 'L1D' and 'L2'), the performance of the victim task becomes noticeably worse as we observe more than 2X execution time increases. When we enable both L1-D and L2 prefetchers (labeled 'L1D/L2'), the result is more than 4X execution time increase, compared to the configuration where all prefetchers are disabled (labeled 'None').

In other words, enabling hardware prefetchers increases the victim's execution time due to increased L2 cache blocking, driven by increased cache writebacks initiated by the additional prefetch refill requests at the L2 cache.

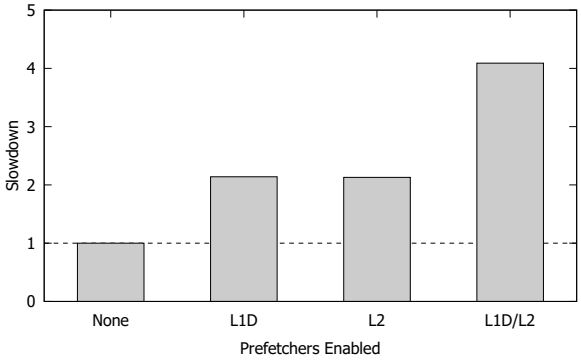


Fig. 8: Effects of hardware prefetchers.

B. Effect of Writeback Buffer Size

In this experiment, we explore the impacts of writeback buffer size of the shared L2 cache to the effectiveness of cache DoS attacks, targeting the writeback buffer. Specifically, we want to know if increasing the size of the L2 writeback buffer reduces L2 cache blocking, which in turn would improve the victim task's performance.

Core	Quad-core, 1.5 GHz, IQ: 96, ROB: 128, LSQ: 48/48
L1-I/D caches	Private 32 kB (2-way), Private 32 kB (4-way), MSHRs: 1 (I), 3 (D), Writeback Buffer: 1 (I), 3(D)
L1-D PF	Stride, Degree: 5, Queue size: 5
L2 cache	Shared 512 kB (16-way), MSHRs: 24, Writeback Buffer: 8, hit latency: 12, LRU
L2 PF	Stride, Degree: 8, Queue size: 8
DRAM controller	Read/write buffers: 64, open-adaptive page policy
DRAM module	DDR3@800MHz, 1 rank, 8 banks

TABLE II: Baseline simulation parameters for Gem5 and Ramulator.

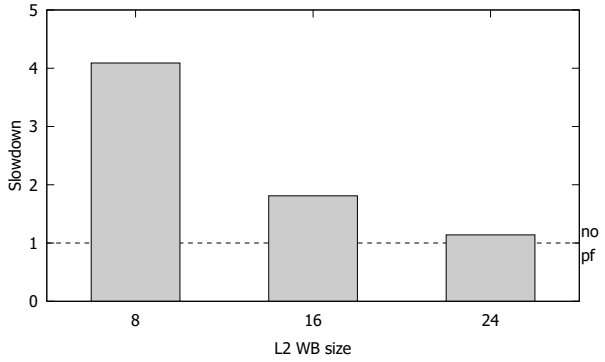


Fig. 9: Effect of the L2 cache writeback buffer size.

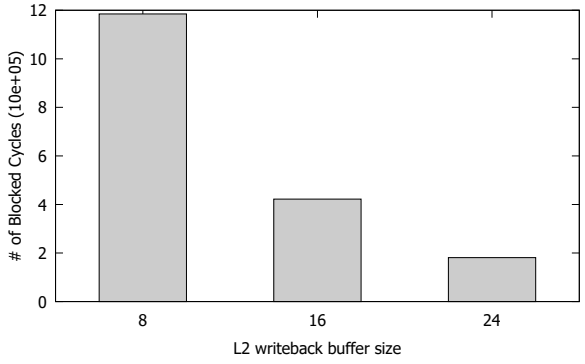


Fig. 10: Blocked cycles vs. the L2 writeback buffer size.

Figure 9 shows the results. As suspected, when we increase the size of the L2 writeback buffer, the performance of the victim task is improved accordingly. This is because the L2 cache's blocked time is decreased. Figure 10 shows the total number of cycles during which the L2 cache is blocked in the same experiment.

In summary, we find that the presence of hardware prefetchers and the size of the L2 cache writeback buffer are major factors affecting the platform's susceptibility to cache DoS attacks.

VI. OS-level Defense Mechanism Against Cache DoS Attacks

In this section, we present an OS-level solution to prevent denial-of-service attacks on the shared cache in a multicore platform, especially those targeting the cache writeback buffer.

Our solution is software-based and is built on top of an existing memory bandwidth throttling mechanism called MemGuard [48]. MemGuard uses per-core hardware performance counters to regulate (throttle) each core's maximum memory bandwidth usage. Specifically, it uses the *LLC miss* counter to calculate the amount of memory bandwidth consumed by each core. Prior studies show the effectiveness of memory bandwidth throttling in protecting real-time tasks [3], [7], [31], [32].

However, we find a significant limitation of using the LLC miss count as a sole means to measure and regulate memory bandwidth because it effectively treats both read and write misses as equal despite the fact that write misses may incur additional writeback traffic on a write-back cache. While the cache writebacks are typically not in the critical path and are processed opportunistically in both cache and DRAM controllers, as discussed in Section V, write-backs can block the cache when the writeback buffer is full. It can also delay cache-line refill operations if the memory controller cannot process backlogged DRAM writes in the background [47]. Thus, as shown in Section IV, we find that write intensive attackers are far more impactful than read intensive ones.

To address this limitation, we propose to extend MemGuard by utilizing an additional performance counter that measures the number of *LLC writebacks* in addition to the existing counter that monitors the LLC misses. By using the two counters, we can regulate both the number of LLC misses and writebacks separately. For example, we can throttle write intensive tasks more without affecting read intensive tasks by setting a low threshold for the writeback counter while setting a high threshold for the cache miss counter.

To demonstrate the effectiveness of this solution, we consider two different application scenarios.

In the first scenario, we investigate the impact of memory bandwidth throttling to application performance on the throttled core. First, we compare the baseline MemGuard and our modified version by applying them to throttle BwRead (DRAM) and BwWrite (DRAM) subject tasks in isolation (i.e., one task at a time). For the baseline MemGuard, we set the LLC miss threshold (read memory bandwidth) to 100 MB/s, and for our modified MemGuard, we set the LLC miss threshold (read) to 500 MB/s, while additionally setting the LLC writeback threshold (write) to 100 MB/s.

Figure 11 shows the results. In the baseline MemGuard, both BwRead and BwWrite tasks are limited to 100 MB/s as both read and write misses are treated equally. With our modification, however, the BwWrite task is limited to 100 MB/s, while the BwRead task's performance is increased to 500 MB/s, as we expected. This means that we can provide

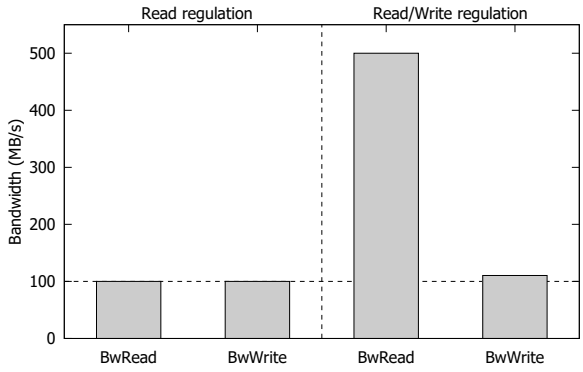


Fig. 11: Effect of read-only (MemGuard [48]) vs. separate read/write bandwidth regulation (Our approach).

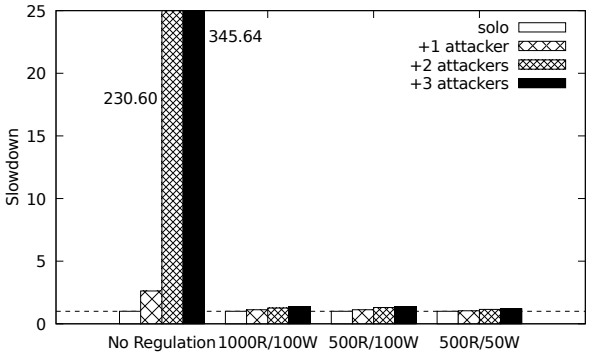


Fig. 13: Effect of read/write bandwidth regulation on protecting BwRead (LLC) victim from write DoS attacks.

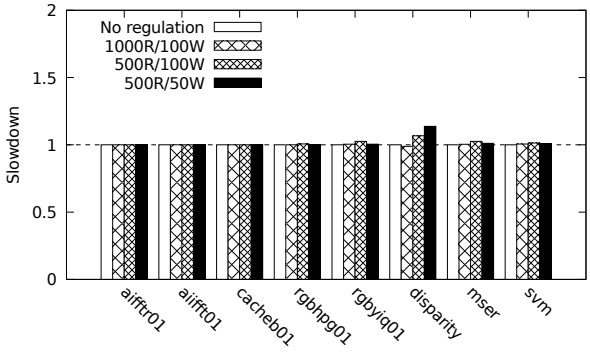


Fig. 12: Effect of read/write bandwidth regulation on the regulated non-attacker tasks.

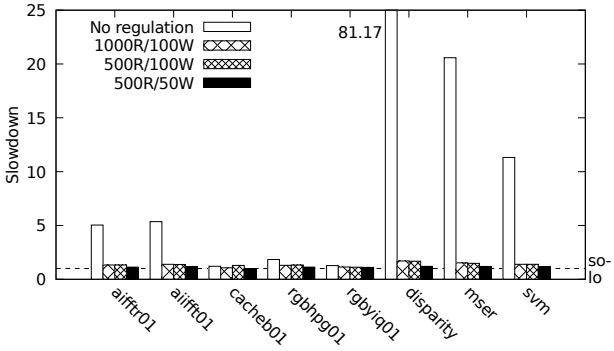


Fig. 14: Effect of read/write bandwidth regulation on protecting EEMBC and SD-VBS victim tasks from write DoS attacks.

the same degree of interference protection by heavily throttling write memory bandwidth, while allowing higher read memory bandwidth to the tasks running on the regulated cores.

In the next experiment, we use EEMBC and SD-VBS benchmarks and evaluate their performance impacts under the following three read/write throttling configurations: *1000R/100W*, *500R/100W*, and *500R/50W*. In 1000R/100W, we set 1000 MB/s threshold for LLC misses and 100 MB/s threshold for LLC writebacks. The other two configurations, 500R/100W and 500R/50W, are similarly defined.

Figure 12 shows the results. Note, first, that for most benchmarks, read/write throttling does not have any noticeable performance impact. At 1000R/100W, in particular, the performance impact of throttling is negligible. This is because the tested benchmarks are mostly not memory intensive and thus do not exceed the assigned throttling parameters. As we assign less read and write bandwidth, to 500R/100W and 500R/50W, some benchmarks, notably *disparity*, show performance impacts due to throttling, but the effects are relatively minor. Note that the effect of throttling may vary significantly depending on the application characteristics. In general, memory intensive applications, especially write intensive ones, will suffer the most, while read intensive ones will suffer much less under our proposed read/write regulation scheme. Fortunately, real-world applications are usually more dependent on read bandwidth rather than write bandwidth, which makes our approach attractive.

In the second scenario, we applied our modified MemGuard to protect against the write attackers on the Raspberry Pi 3 platform, which suffered the most severe interference (Section IV). We repeat the experiments in Section IV-C and Section IV-F with three different read/write throttling configurations to regulate the write attackers.

Figure 13 shows the results for the BwRead (LLC) victim task (c.f., Figure 4b in Section IV-C). As can be seen, applying write regulation on the attacker cores is effective in protecting the victim task, and decreasing the threshold can further improve performance. Concretely, the victim task's execution time increase is reduced from 345X down to 1.34X with a write threshold of 100MB/s and 1.2X with a write threshold of 50MB/s. In this experiment, increasing read regulation from 500MB/s to 1000MB/s has no effect because the attackers are throttled by the write regulation.

Figure 14 shows the results for the EEMBC and SD-VBS benchmark victim tasks in the presence of three write attackers (c.f., Figure 7a). Similar to the BwRead (LLC) experiment above, when write regulation is applied on attacker cores, the performance of the victim tasks improve. This is most noticeable in the SD-VBS benchmarks, such as the disparity benchmark whose WCET increase is reduced from 81.17X to

1.20X with the 500R/50W setup, showing the effectiveness of our read/write regulation approach in protecting against cache DoS attacks.

In summary, we show that our extended OS-level mechanism, which regulates read and write bandwidth separately, allows us to apply more efficient and targeted bandwidth regulation policies that can prevent cache DoS attacks, while minimizing the performance impact for the regulated cores with minimal performance impact to non-attacker tasks.

VII. Related Work

In a real-time system, the ability to guarantee predictable timing is highly important. However, it is difficult to achieve predictable timing in a multicore platform due to shared hardware resources, such as cache and main memory. Therefore, much research effort has been focused on analyzing and controlling the timing impacts of these shared hardware resources in multicore platforms. For shared caches, most prior works focused on cache space partitioning by using various OS or hardware mechanisms [9], [16]–[19], [25], [27], [37], [43], [44]. Recently, however, Valsan et al. experimentally showed that cache space partitioning does not necessarily guarantee cache performance isolation in non-blocking caches used in modern multicore processors [39]. The authors then identified miss-status-holding-registers (MSHRs), which are the cache's internal buffers to track outstanding cache-misses, as the source of the observed timing increases in a number of out-of-order multicore platforms. In essence, they identified cache MSHRs as a denial-of-service (DoS) attack vector. In contrast, our work shows that even relatively simple in-order multicore platforms are not immune to cache DoS attacks and identifies an additional internal hardware structure of a non-blocking cache, namely the writeback buffer, as another DoS attack vector.

Several prior studies investigated various DoS attack vectors in multicore. Moscibroda et al. examined DoS attacks on memory (DRAM) controllers [28]. They found that the commonly used FR-FCFS [33] scheduling algorithm, which may re-order memory requests to maximize throughput, is vulnerable to DoS attacks. They suggested "fair" memory scheduling as a solution. Many subsequent papers proposed various fair memory scheduling algorithms in DRAM controllers [21], [29], [30], [36]. Keramidas et al. studied DoS attacks on cache space and proposed a cache replacement policy that allocates less space to such attackers (or cache "hungry" threads) [15]. Woo et al. investigated DoS attacks on cache bus (between L1 and L2) bandwidth, main memory bus (front-side bus) bandwidth, and shared cache space, on a simulated multicore platform [45]. In contrast, our work demonstrates the feasibility of shared cache DoS attacks on real multicore platforms and identifies an internal hardware structure of non-blocking caches as a DoS attack vector. Furthermore, we present an OS-based solution on a real multicore platform that prevents identified cache DoS attacks.

Recently, microarchitectural timing attacks [11] have gained significant attention, both from the public and the research community, in the wake of the Meltdown, Spectre, and Foreshadow attacks [23], [26], [40]. In general, these timing attacks aim to gain secret information through externally observable timing differences in accessing microarchitectural resources such as cache. In contrast, DoS attacks on microarchitectural resources, which we focus on in this paper, aim to directly influence performance (timing) of the victim applications or cores. The Rowhammer attack [20] is another kind of attack targeting hardware. It exploits a reliability failure mode in modern DRAM hardware where repeatedly and quickly accessing certain DRAM locations can result in bit flips in nearby memory locations. Successful attacks that break OS memory isolation boundaries have been demonstrated in servers and mobile devices [34], [41]. As safety-critical embedded real-time systems are becoming more connected, such as we are already seeing in cars, we believe that software attacks targeting computer hardware are increasingly important areas that need attention from both the real-time and security research communities.

VIII. Conclusion

In this paper, we investigated the feasibility and severity of DoS attacks on shared caches in multicore platforms. From careful experiments on a number of contemporary embedded multicore platforms, we observed surprisingly severe execution time increases—up to 346X slowdown—on some of the tested platforms. In particular, we found that two recent in-order architecture, ARM Cortex-A53, based multicore platforms are especially more susceptible to write-intensive cache DoS attacks than more complex out-of-order architecture based multicore platforms.

From detailed micro-architectural analysis using a cycle-accurate full system simulator, we identified the shared cache's writeback buffer as a possible DoS attack vector, in addition to the previously known cache MSHR, that could have caused the behaviors we observed on the real platforms.

We propose a software (OS) solution to mitigate cache DoS attacks targeting the shared cache's writeback buffer. Our solution implements a separate read and write memory bandwidth regulation mechanism to effectively counter write intensive cache DoS attacks while minimizing performance impacts to read heavy normal applications. Our solution is implemented in Linux and shown to be effective to counter the shared cache DoS attacks.

As future work, we plan to investigate OS and architecture support mechanisms to withstand different types of micro-architectural attacks (e.g., cache timing attacks [23], [26], Rowhammer attacks [20]) in the context of safety-critical real-time systems.

Acknowledgements

This research is supported by NSF CNS 1718880, CNS 1815959, and NSA Science of Security initiative contract #H98230-18-D-0009.

REFERENCES

[1] Eembc benchmark suite. http://www.eembc.org.

[2] Memory system in gem5. http://www.gem5.org/docs/html/gem5MemorySystem.html.

[3] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *Real-Time Systems Symposium (RTSS)*, 2018.

[4] ARM. *Cortex-A15 Technical Reference Manual, Rev: r4p0*, 2011.

[5] ARM. *Cortex-A7 Technical Reference Manual, Rev: r0p5*, 2012.

[6] ARM. *Cortex-A53 Technical Reference Manual, Rev: r0p4*, 2014.

[7] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun. DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture (HPCA)*, pages 340–351. IEEE, 2005.

[10] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92. San Francisco, 2011.

[11] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. pages 1–37, 2016.

[12] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[13] A. Hamann. Industrial challenges: Moving from classical to high performance real-time systems. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2018.

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[15] G. Keramidas, P. Petoumenos, S. Kaxiras, A. Antonopoulos, and D. Serpanos. Preventing denial-of-service attacks in shared cmp caches. In *International Workshop on Embedded Computer Systems*, pages 359–372, 2006.

[16] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[17] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Real-Time Systems (ECRTS)*, pages 80–89. IEEE, 2013.

[18] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759, 2017.

[19] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 111–122. IEEE, 2004.

[20] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[21] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *International Symposium on Microarchitecture (MICRO)*, pages 65–76. IEEE, 2010.

[22] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 2016.

[23] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint*, 2018.

[24] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *International Symposium on Computer Architecture (ISCA)*, pages 81–87. IEEE Computer Society Press, 1981.

[25] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium (RTAS)*, pages 213–224. IEEE, 1997.

[26] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint*, 2018.

[27] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[28] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. USENIX, 2007.

[29] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture (MICRO)*, pages 146–160. IEEE, 2007.

[30] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *International Symposium on Computer Architecture (ISCA)*, volume 36, pages 63–74. IEEE Computer Society, 2008.

[31] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[32] R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[33] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[34] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[35] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

[36] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2013.

[37] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture (HPCA)*. IEEE, 2002.

[38] T. Ungerer et al. parMERASA–Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *Digital System Design (DSD)*, pages 363–370, 2013.

[39] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, and K. Leuven. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, pages 991–1008, 2018.

[41] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[42] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64. IEEE, 2009.

[43] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[44] A. Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, 2(3):315–327, 1994.

[45] D. H. Woo and H. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[46] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[47] H. Yun, R. Pellizzoni, and P. Valsan. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 184–195, 2015.

[48] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.