Per-Bank Bandwidth Regulation of Shared Last-Level Cache for Real-Time Systems

Connor Sullivan
University of Kansas
Lawrence, Kansas USA
connor.sullivan13@ku.edu

Alex Manley
University of Kansas
Lawrence, Kansas USA
amanley97@ku.edu

Mohammad Alian*
Cornell University
Ithaca, New York USA
malian@cornell.edu

Heechul Yun
University of Kansas
Lawrence, Kansas USA
heechul.yun@ku.edu

***Abstract*—Modern commercial-off-the-shelf (COTS) multicore processors have advanced memory hierarchies that enhance memory-level parallelism (MLP), which is crucial for high performance. To support high MLP, shared last-level caches (LLCs) are divided into multiple banks, allowing parallel access. However, uneven distribution of cache requests from the cores, especially when requests from multiple cores are concentrated on a single bank, can result in significant contention affecting all cores that access the cache. Such cache bank contention can even be maliciously induced—known as cache bank-aware denial-of-service (DoS) attacks—in order to jeopardize the system's timing predictability.**

In this paper, we propose a per-bank bandwidth regulation approach for multi-banked shared LLC based multicore real-time systems. By regulating bandwidth on a per-bank basis, the approach aims to prevent unnecessary throttling of cache accesses to non-contended banks, thus improving overall performance (throughput) without compromising isolation benefits of throttling. We implement our approach on a RISC-V system-on-chip (SoC) platform using FireSim and evaluate extensively using both synthetic and real-world workloads. Our evaluation results show that the proposed per-bank regulation approach effectively protects real-time tasks from co-running cache bank-aware DoS attacks, and offers up to a 3.66× performance improvement for the throttled benign best-effort tasks compared to prior bank-oblivious bandwidth throttling approaches.

I. Introduction

Modern commercial-off-the-shelf (COTS) multicore processors are equipped with sophisticated memory hierarchies that support a high degree of memory-level parallelism (MLP). Because memory accesses often take significantly longer than actual computation, enabling high MLP across all levels of the memory hierarchy is crucial for achieving high performance in modern multicore architectures.

To facilitate high MLP, shared last-level caches (LLCs) are often organized into multiple banks that can be independently accessed in parallel. For instance, the LLC of the ARM Cortex-A72 processor has two independent tag banks, each of which is further divided into four data banks [1]. Such a multi-bank cache design maximizes parallelism and throughput in accessing the cache, and is widely adopted in high-performance multicore architectures [1]–[5], including those that are used in safety-critical embedded real-time systems in automotive and aviation domains [6], [7].

While most prior work on shared cache for real-time systems has focused on cache space partitioning, multiple studies have shown that partitioning cache space alone does not guarantee temporal isolation in accessing the cache [8]–[12]. In particular, it has been shown that the performance of a multi-bank cache can degrade significantly when requests to the cache are unevenly distributed across the banks. In the worst-case scenario, when all requests are concentrated on a single cache bank, severe contention can arise. Such bank conflicts can disrupt the system's temporal predictability and be leveraged as cache bank-aware denial-of-service (DoS) attacks [12].

To mitigate shared cache bank contention, the prior study [12] suggested a software-based cache bandwidth throttling approach as a potential solution, which is based on MemGuard [13] and uses hardware performance counters to monitor and regulate the LLC access bandwidth of the offending cores (those that generate excessive parallel requests to the LLC). However, such a software-based bandwidth throttling solution severely impacts the performance of the throttled cores. To provide sufficient isolation for the protected real-time task, it reportedly incurs up to 300× slowdown of the throttled tasks [12], which may be unacceptable overhead for many applications. While hardware-based memory bandwidth throttling solutions [14]–[16], if used for LLC bandwidth throttling, can potentially reduce the overhead of software-based throttling, their effectiveness is still fundamentally limited because they are not aware of cache banks when regulating bandwidth, which makes them overly pessimistic.

In this paper, we propose per-bank bandwidth regulation of shared LLCs for predictable and efficient use of the shared cache in multicore SoCs for real-time systems. Our approach is motivated by the observation that the worst-case bank contention arises when cache accesses are concentrated on a single cache bank rather than distributed across the banks. As such, instead of throttling bandwidth to the entire shared LLC, we apply bandwidth throttling on a per cache-bank basis to only throttle accesses when there is a bank conflict. This effectively multiplies the permissible cache access bandwidth of best-effort tasks without compromising the isolation benefits of bandwidth throttling to the protected real-time tasks.

We implement the proposed per-bank throttling capability as an extension to an open-source hardware memory bandwidth

[*] This work was conducted while affiliated with the University of Kansas.

regulator [16] on a RISC-V system-on-chip (SoC) platform using Xilinx UltraScale+ VCU118 FPGA [17] and FireSim [18]. We evaluate the effectiveness of the proposed approach in providing temporal isolation to the real-time victim tasks in the presence of cache bank-aware DoS attacks. We then demonstrate the efficiency benefits of per-bank regulation over prior approaches that throttle the aggregate bandwidth of all banks globally. We show that per-bank regulation can effectively protect victim tasks from the attack while providing best-effort tasks with up to a 3.66× performance improvement over the prior bank-oblivious regulation scheme.

In summary, we make the following contributions:

- We propose per-bank bandwidth regulation on shared LLC to effectively and efficiently defend against potential cache bank contention attacks (regardless of whether malicious or benign).
- We present a prototype hardware design, which can be integrated into any RISC-V SoC that supports the standard TileLink interconnect, and analyze its ability to prevent cache bank-aware DoS attacks.
- We implement our design on a realistic cycle-exact, FPGA-accelerated full-system simulator, and evaluate its performance improvements over prior bank-oblivious regulation approaches. We also provide our design as open-source*.

The remainder of the paper is organized as follows. Section II provides the necessary background. Section III defines the threat model. Section IV motivates the need for per-bank regulation. We present our proposed per-bank regulation design in Section V and the evaluation results in Section VI. We discuss related work in Section VII and conclude in Section VIII.

II. Background

In this section we provide the necessary background on multi-banked caches, cache bank-aware DoS attacks, and bandwidth regulation methods.

A. Multi-Bank Cache Organization

The shared cache of a modern multicore processor is often composed of multiple independent banks (sometimes referred to as slices [19]), which can be accessed in parallel. This multi-bank cache organization facilitates high MLP, which is crucial for high-performance multicore processors. In a multi-bank cache, a mapping function determines the bank from a given physical address. The mapping function can be as simple as using a subset of the memory address bits.

Figure 1 depicts the multi-bank LLC organization of the ARM Cortex-A72 [1]. Note that it is comprised of two independent tag banks, each of which is further divided into four sub-banks called data banks. The tag banks are completely independent, allowing for two separate LLC accesses to be serviced in parallel. Likewise, the data banks allow for further interleaving of accesses. To index the tag and data banks, physical address bits 4, 5 and 6 are used. Bit 6 indexes between the two tag banks, with bits 4 and 5 being used to index the data banks within each tag bank. For 64 byte cache lines, each line is split into four sub-lines of 16 bytes that are striped across the data banks.

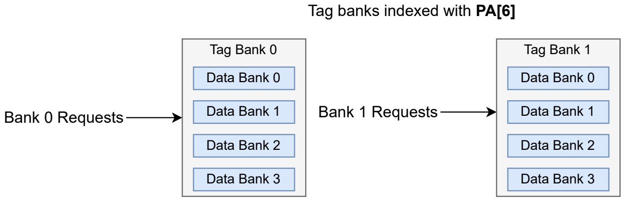


Fig. 1: ARM Cortex-A72 LLC Organization [1].

It is important to note that these bits (4, 5 and 6) are in the lower 12 bits of an address, within the page offset. This means that these bits can be fully controlled from the user space without the need for elevated privileges or huge pages. With this understanding, an attacker can direct memory accesses to specific banks, opening the door for potential DoS attacks [12].

B. Cache Bank-Aware DoS Attack

The feasibility of cache bank-aware DoS attacks was first demonstrated in a recent study [12] on both ARM Cortex-A57 [2] and Cortex-A72 [1] cores. Under the threat model described in Section III, the study shows that by saturating a single cache bank of the shared L2 cache with many parallel requests, an attacker can cause up to a 10× cross-core slowdown on a victim task. This slowdown occurs even when the victim is running on a dedicated core in isolation, accessing a dedicated L2 cache (space) partition by means of page coloring. As the cache space is partitioned between the victim and the attacker, it demonstrates that the slowdown is not caused by cache evictions. Furthermore, it also shows that the contention occurs at the bank level—not at the bus level—as no slowdown is observed when the victim and attacker target separate cache banks. Lastly, the worst-case slowdown occurs when both the victim and the attacker access the same cache bank, suggesting that the cache bank bandwidth becomes the bottleneck in such a situation.

C. Cache Bandwidth Regulation

To mitigate cache bank-aware DoS attacks, the prior study [12] proposed a software-based cache bandwidth regulation method, LLCGuard, which uses per-core performance counters to regulate (limit) each core's LLC access bandwidth (as opposed to DRAM bandwidth regulation proposed by MemGuard [13]) at a regular time interval (e.g., 1ms). However, the software-based approach is known to incur very high performance cost to the throttled best-effort cores. Concretely, the study reports up-to 300× slowdown of the tasks on the throttled cores to ensure no more than 1.1× slowdown of the protected real-time tasks on the unregulated core. As discussed in [12], part of the reason for such a massive performance loss is due to software implementation overhead. With the

*https://github.com/CSL-KU/per-bank-regulation-firesim regulation period of 1ms, a large amount of LLC accesses can still occur in short bursts, which results in LLC bank contention.

In contrast, a hardware-based cache bandwidth solution can operate at a much finer granularity (in cycles), which can help spread the LLC accesses more evenly across the entire throttle period, thereby reducing the negative performance impact of throttling best-effort cores. While existing hardware-based bandwidth regulators, such as Intel RDT [14] and ARM MPAM [15], are mainly designed to regulate memory bandwidth, they can potentially be modified to regulate cache bandwidth to mitigate cache bank contention.

Unfortunately, all aforementioned regulation schemes, both software and hardware, suffer from a common limitation—they do not regulate at the *bank level*, where the actual contention occurs. Instead, they treat the entire cache (or DRAM) as a single resource and regulate its total access bandwidth. We henceforth refer to the latter as *all-bank* regulation. In the following, we show why this all-bank regulation is overly *pessimistic*.

III. Threat Model of Cache Bank DoS Attacks

In this work, we consider the same threat model used in [12]. That is, we assume: (1) a victim task and one or more attacker tasks are co-located on a multicore platform, which has a shared last-level cache (LLC) and main memory (DRAM); (2) the victim and the attackers are partitioned to run on dedicated CPU cores and LLC cache spaces; (3) the attackers have non-privileged access on the target platforms and can only execute code from the userspace; (4) the cache bank address mapping information is known beforehand either from datasheets [1], [2] or reverse engineering [20], [21]. Following these conditions, our goal is to guarantee temporal isolation of the victim accessing the shared cache in the presence of co-scheduled attackers, while maximizing cache bandwidth throughput available to the attackers.

IV. Motivation

In this section, we first evaluate the effect of cache bank-aware DoS attacks, synthetic workloads that generate severe cache bank contention [12], on two embedded multicore platforms (Section IV-A). We then discuss the limitations of bank-oblivious "all-bank" cache bandwidth regulation approaches in mitigating such attacks (Section IV-B).

A. Effects of Cache Bank-Aware DoS Attacks

In this experiment, we use two contemporary embedded multicore platforms: Raspberry Pi 4 Model B [22] and BeagleV Ahead [5]. The Raspberry Pi 4 is based on the Broadcom BCM2711 SoC and is equipped with four ARM Cortex-A72 [1] cores with a 1MB shared L2 cache. Comparably, the BeagleV Ahead is based on the Alibaba T-Head TH1520 SoC, equipped with four Xuantie C910 RISC-V cores with a 1MB shared L2 cache. Table I shows the basic characteristics of the two platforms.

Platform	Raspberry Pi 4 (B)	Beagle V Ahead
SoC	BCM2711	TH1520
Architecture	ARMv8-A	RISC-V 64GC
CPU	4x Cortex-A72 out-of-order 1.5GHz 48KB(I)/32KB(D)	4x Xuantie C910 out-of-order 2.0GHz 64KB(I)/64KB(D)
Shared L2 Cache	1MB	1MB
Memory	4GB LPDDR4	4GB LPDDR4

TABLE I: COTS embedded multicore platforms.

For software, the Pi 4 runs Raspberry Pi OS with Linux kernel 6.6, and the Beagle V runs Ubuntu 20.20 with Linux kernel 5.10. In both platforms, the kernels are patched with PALLOC [23], a page coloring mechanism for Linux, to partition the L2 cache space equally between the victim and the attackers.

For evaluation, we use the *BkPLL* workload from [12]. As both the victim and the attackers, *BkPLL* is a pointer chasing workload that can generate a configurable number of parallel memory requests targeting a specific cache bank. We first run the victim on one core in isolation and measure its performance. We then repeat the experiment in the presence of co-running attacker tasks on the other cores. We evaluate different combinations of target cache banks for the victim and the attackers: *Same Bank* refers to the case where the victim and the attacker target the same cache bank, whereas *Diff Bank* refers to the case where they target different banks.

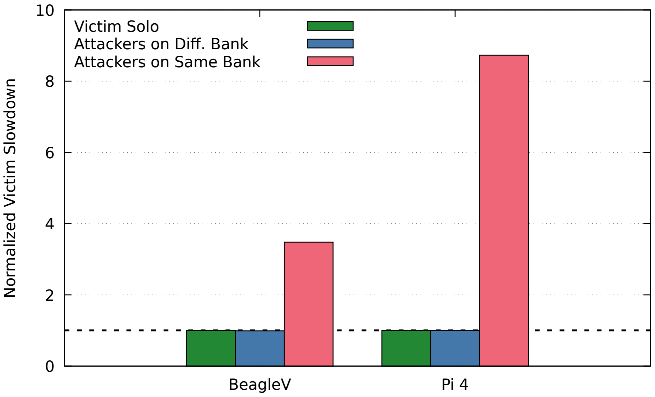


Fig. 2: Effects of cache bank-aware DoS attacks

Figure 2 shows the normalized slowdowns of the victim on different cache bank mapping configurations. The dashed horizontal line denotes the baseline 1.00× slowdown (in this case, solo performance). First we notice that, consistent with the findings in [12], contention occurs at the cache bank level, not on the shared bus level. The victim sees no slowdown when the attackers target a different bank, whereas severe slowdown is observed when both target the same cache bank. Second, we observe up to 8.7× slowdown on the Pi 4 platform, which is considerably worse than the 8.3× reported worst-case slowdown on the same platform [12]. Interestingly, we find different target data bank selections for the victim and the attackers contribute to the increased worst-case slowdown.

Third, the BeagleV platform shows similar trends but its worst-case slowdown is considerably less (3.5×) than that of the Pi 4 (8.7×). This is due to the differences in baseline performance—i.e., the CPU core's ability to concurrently generate requests and the peak bandwidth of the cache. Note that Raspberry Pi 4's peak cache bandwidth is 2× higher than that of the BeagleV. In general, faster processors tend to suffer larger worst-case slowdowns.

B. Limitations of "All-Bank" Bandwidth Regulation

The results in the previous subsection show that the contention created by the DoS attack is not on the shared bus, but at the targeted cache bank. This indicates that, in order to mitigate the bank contention attack, we only need to limit (throttle) the traffic (bandwidth) going into the contended bank. Furthermore, the banks in the cache are independent of each other. As such, regulation should be applied on a per-bank basis rather than applied unnecessarily across all banks. Unfortunately, existing bandwidth regulation approaches cannot be applied to individual cache banks.

For example, BRU [16] is a hardware-level bandwidth regulator inserted between the L1 caches and the shared L2 cache [16]. As such, it regulates the L2 access traffic of the subset of cores that may be executing the DoS attackers. However, BRU tracks all L2 access traffic, without consideration for the individual bank destination. In other words, it implements an "all-bank" bandwidth regulation scheme.

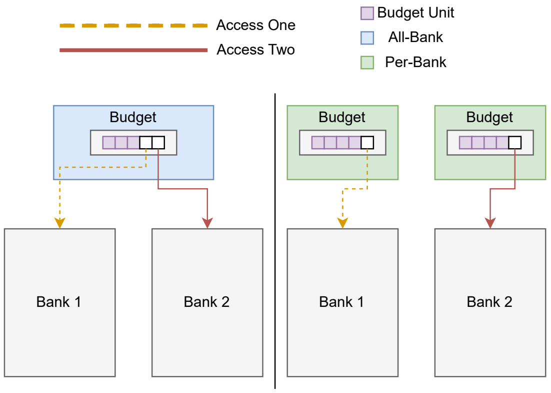


Fig. 3: All-bank vs. per-bank bandwidth regulation on multi-bank shared caches

Figure 3 depicts the high-level intuition illustrating why all-bank regulation is needlessly pessimistic. Consider two bandwidth regulation systems, one with all-bank regulation (left) and one with per-bank (right). Both systems have two cache banks. Suppose that we need to limit the traffic to 5 accesses to a cache bank per regulation period to mitigate the contention on the bank. In the case of all-bank regulation it is bank oblivious, thus the global cache access budget must be set to 5 accesses, to counter the worst-case where all traffic goes to one cache bank. This budget is deducted on every cache access, even though in reality, the two cache accesses are interleaved across two different banks.

Per-bank regulation, in contrast, can apply the access budget of 5 to each bank separately, only then is a budget deducted when the specific bank is accessed. As such, when the two accesses are interleaved over the two banks, each bank's budget is depleted by one, leaving a remaining budget of 4 for each bank, whereas only 3 would be left in the all-bank case. As more accesses are interleaved, per-bank regulation can provide higher aggregate bandwidth while still providing worst-case cache bank contention guarantees. With this intuition in mind, we now discuss our proposed per-bank cache bandwidth regulation system design.

V. Per-Bank Cache Bandwidth Regulation

In this section, we describe the design and implementation details of the proposed per-bank cache bandwidth regulation approach.

A. Design Overview

Our per-bank bandwidth regulation solution is implemented as an extension to an open-source hardware bandwidth regulator called BRU [16], designed to drop into an SoC design between the cores/accelerators and the shared cache. Figure 4 depicts a high-level view of our bandwidth regulation unit in a basic dual-core setup.

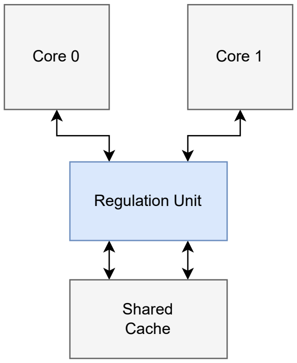


Fig. 4: High-level view of a regulation unit in a dual-core SoC

BRU supports creation of multiple arbitrary domains, each of which may be composed of one or more cores. A domain is the primary entity that bandwidth regulation is applied to. In the original BRU design, each domain can be configured with a period (cycles) and a budget (number of memory requests). The budget is decremented for any memory request made to the shared cache (regardless of which bank it targets) and once the budget is depleted, all cores in the domain are denied access to the shared cache until the period expires and the budget is replenished. As discussed earlier, we call this all-bank regulation because it counts an access to any bank equally.

Our modifications to BRU enable tracking and regulating the budget for each shared cache bank rather than for the entire cache. Specifically, for each request to the cache, we decode its destination cache bank address and charge it to the corresponding bank's budget. This means, for an N bank shared cache, we have N separate bank budgets to keep track of. When any one bank's budget is depleted, further access to the bank will be prevented(throttled) until the next period begins. Accesses to other banks can still occur as long as their budgets are not depleted.

B. Per-bank Bandwidth Regulation Interface

To enable fixed user-defined bandwidth regulation, our design exposes memory-mapped I/O (MMIO) registers. The *Access Budget Register (ABR)* is used to program the maximum number of accesses per period and the *Regulation Period Register (RPR)* sets the regulation period in cycles. Equation 1 represents the bandwidth budget assigned to each bank given the values of the *ABR* and *RPR* registers. Note that each bank gets the budget *BW*, rather than it being evenly distributed among the banks. *TS* represents the *transaction size* and f the *clock frequency*. When a core accesses the cache, *TS* is equivalent to the line size (commonly 64 bytes).

$$\mathrm{BW} = \frac{ABR}{RPR} \times TS \times f \tag{1}$$

The *RPR* is applied globally to all logical groupings of cores that are being regulated concurrently—we refer to these groupings as regulation *domains*. Each domain has an *ABR*, allowing unique per-domain budgets to be applied to each cache bank.

Along with a budget configuration interface, fixed bandwidth regulation requires mechanisms to track accesses (per-bank counters in our design) and regulate these accesses as necessary. We organize the per-bank access counters in a *Domain Control Interface (DCI)*, with a *Core Control Interface (CCI)* containing regulation enable registers and domain assignment registers.

Domain Control Interface. Each domain has its own access counter registers. We denote these registers as *Bank Access Counters (BAC)*. A given domain has N of these registers, where N is equal to the number of banks in the shared cache. These registers are used solely for bandwidth regulation. Note that this interface also includes the user configurable per-domain *ABR* registers.

Core Control Interface. This interface includes logic for assigning cores to domains and enabling regulation for a given core. Domain assignment is handled through the *Domain Assignment Registers (DAR)*, enabling each core to be configured to any one domain. A *Regulation Enable Register (RER)* is generated for each core and allows for regulation to be enabled or disabled seamlessly.

Figure 5 depicts the regulation unit's control interface for an arbitrary quad-core system configured to have two regulation domains. In the *CCI* there is logic for the four cores. Cores 0-2 are assigned to Domain 0, with their *RERs* set high. Core 3 is assigned to Domain 1, but its *RER* is kept low, meaning that the core's accesses are not being regulated. Bracketed register names(i.e. [Domain]) indicate that they are memory mapped and user configurable.

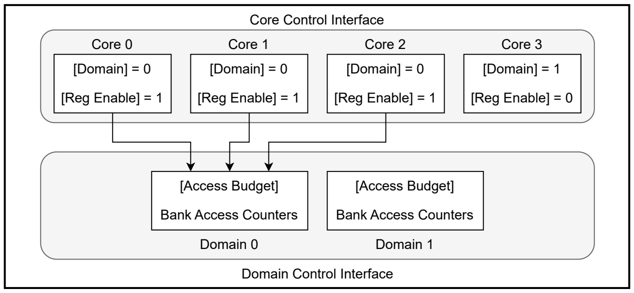


Fig. 5: Example of group regulation in a quad-core system. Core 0-2 belong to Domain 0, which is regulated. Core 3, on the other hand, belongs to Domain 1, which is not regulated. In this example, Domain 0 is for *best-effort* tasks while Domain 1 is for the *real-time* tasks.

C. Per-bank Bandwidth Monitoring Interface

In addition to per-bank regulation, we also provide a per-bank bandwidth monitoring interface to enable software(OS) based fine-grained monitoring and adaptive bandwidth regulation capabilities. Specifically, our bandwidth regulation unit includes per-bank monitoring registers that are separate from the regulation interface. The per-bank monitoring registers form the *monitoring interface*. For every core in the system, a set of N counters is generated, where N is equal to the number of banks. Similar to typical performance counters, these counters can be reset and read by a user to determine the per-bank bandwidth and access pattern of a core.

D. Regulation and Monitoring Algorithms

Algorithm 1 shows the pseudo-code of our regulation unit. At a high level, it manages the global period counter (lines 1-8), the per-bank access counters, and throttling (lines 9-22).

Each clock cycle, *PeriodCounter* is incremented to advance the current regulation period (line 7). The *PeriodCounter* is reset when its value equals or exceeds the user defined *RegulationPeriod*. As part of this reset, all bank budgets are replenished by zeroing the *BankCounters* (lines 1-6).

Lines 9-22 make up the main body of our regulation algorithm, with the logic being evaluated per-core and per-bank (lines 9 and 10). In a given domain, if the budget of bank j is depleted, then all further accesses to that bank are stalled for the cores in that domain (lines 13-16). When a core sends a bank access, the corresponding domain's bank access counter is incremented (lines 17-20). Specifically, bank accesses occur on Channel A ($A(i).isAccess$), a TileLink notation which we further explain below (Section V-E).

The monitoring interface is handled similarly. Algorithm 2 shows the pseudo-code for the monitoring interface. The logic happens per-core and per-bank (line 1 and 2). Lines 3-8

Algorithm 1 Per-bank Regulation Algorithm

```
if PeriodCounter $\geq$ RegulationPeriod then
    PeriodCounter $\leftarrow$ 0
    for all c in BankCounters do
        c $\leftarrow$ 0
    end for
else
    PeriodCounter++
end if
for i $\leftarrow$ 0 to nCores - 1 do
    for j $\leftarrow$ 0 to nBanks - 1 do
        stall(i)(j) $\leftarrow$ False
        AccessIsBank(i)(j) $\leftarrow$ (j == bankBits)
        if (BankCounters(Domain(i))(j)$\geq$AccessBudget)
                and AccessIsBank(i)(j) then
            stall(i)(j) $\leftarrow$ True
        end if
        if A(i).isAccess and AccessIsBank(i)(j) then
            BankCounters(Domain(i))(j)++
        end if
    end for
end for
```

are similar to the main body of 1, where a bank monitor counter is incremented when that specific bank is accessed. The difference being that there is no notion of domains or period in the monitoring interface.

Algorithm 2 Per-bank Monitoring Algorithm

```
for i $\leftarrow$ 0 to nCores - 1 do
    for j $\leftarrow$ 0 to nBanks - 1 do
        AccessIsBank(i)(j) $\leftarrow$ (j == bankBits)
        if CoreAccess(i) and AccessIsBank(i)(j) then
            BankMonitor(i)(j)++
        end if
    end for
end for
```

Note that our implementation is written in the Chisel hardware description language (HDL) [24]. This allows our design to support any number of domains and cache banks through configurable parameters, eliminating the need to modify the hardware design code.

E. Implementation

We implement our design using the Chipyard SoC Framework [25]. In this subsection, we discuss details of both the TileLink [26] interconnect specification and the Rocket Chip SoC [27] as they relate to our implementation.

Our regulation unit interfaces with TileLink Cached (TL-C) edges. TL-C edges connect the cores to the shared memory subsystem and are cache coherent [26]. There are five channels of communication on TL-C edges: *A, B, C, D* and *E*. We focus on *Channel A*, which carries requests from the core's private caches to the shared caches and memories.

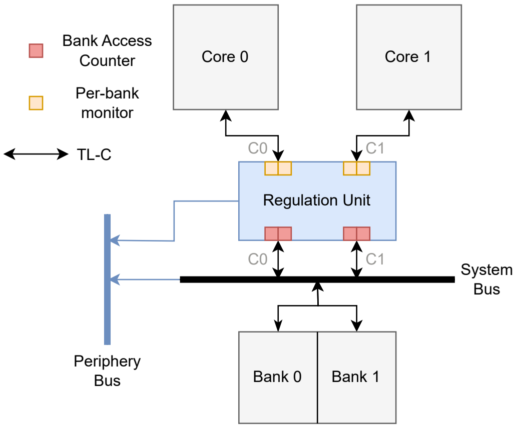


Fig. 6: Dual-core Rocket SoC with per-bank regulation unit

Figure 6 depicts our bandwidth regulation unit in a generic dual-core Rocket Chip SoC design. The connections between the cores and the shared system bus are TL-C edges. When a data or instruction cache miss occurs in the private L1 caches of the cores, a request is sent over Channel A. By monitoring this channel we can track per-core accesses and regulate when a domain's bank budget is depleted.

For synchronizing messages on a given channel, TileLink uses a ready-valid interface for sender/receiver handshaking. To regulate a channel, we can simply set the ready and valid signals to low, effectively stalling the request. Channel A also carries information about the the memory address being requested. From this we can extract the bank address bits to count per-bank accesses. All TL-C channels going from the core to the shared system bus pass through the regulation unit. However, only Channel A is monitored for accesses and regulated via the ready-valid signals. All other channels and signals remain unmodified, passing through and connecting directly to the shared system bus.

Along with connections to the system bus, the regulation unit also connects to the *periphery bus*. This bus is utilized by cores to read/write to the MMIO registers. Figure 6 also depicts the *BAC* counter registers and monitoring interface registers.

VI. Evaluation

In this section, we evaluate hardware bandwidth regulation's ability to defend against the cache-bank DoS attack and show the benefits of per-bank over all-bank bandwidth regulation.

A. Experimental Setup

We use FireSim, an FPGA accelerated cycle-exact full system simulator [18]. This allows us to accurately evaluate the performance of the proposed hardware design when deployed in ASIC, which operates at a higher clock (e.g., >1GHz) while being simulated on a FPGA at an actual clock speed of 100MHz.

Cores	1×BOOM, 1GHz, out-of-order, 3-wide, ROB: 96, LSQ: 24/24, L1: 32K(I)/32K(D) 2×Rocket, 1GHz, in-order, L1: 16K(I)/16K(D), attached with Mempress traffic generators
Shared L2 Cache	1MB (16-way)
Memory	4GB DDR3

TABLE II: Evaluation platform specifications

Table II shows the basic characteristics of the tri-core heterogeneous SoC we constructed on FireSim for evaluation. The SoC is composed of one out-of-order core, the Berkeley Out-of-Order Machine (BOOM) [28], and two in-order Rocket [27] cores, which are connected to Mempress traffic generators [29]. All cores share a 1MB L2 cache and a 4GB DDR3 main memory subsystem.

Note that cache bank-aware DoS attacks [12] require out-of-order CPU cores to be able to generate many concurrent memory requests on a specific target cache bank. As such, we initially attempted to construct a quad-core BOOM based SoC on FireSim using the *LargeBoom* configuration. However, due to physical constraints of our FPGA platform, we were unable to fit four large BOOM cores simultaneously in the FPGA. Furthermore, there is an unresolved bug in BOOM that results in a simulation hang when executing certain memory intensive workloads on multi-core configurations[*]. As such, in our simulation setup, we instead utilize the Mempress traffic generator [29] to act as the cache bank attacker tasks.

Mempress is a configurable hardware unit that can generate multiple parallel streams of requests to the shared memory at varying access patterns. Implemented as an on-chip RoCC accelerator [30], Mempress has access to the shared memory subsystem. For all following experiments, the attackers will be two separate Mempress units targeting the same last-level cache (LLC) bank.

The Mempress traffic generators are attached to two Rocket cores (one per core). Since Rocket cores are in-order, they cannot create significant contention in the shared cache on their own. However, Mempress enhances the cores by enabling them to generate parallel accesses through the traffic generators, all while still meeting FPGA space constraints. All targets are clocked at 1GHz.

For the shared L2 cache, we use SiFive's open-source inclusive cache [31], which is a real synthesizable hardware cache design that supports a configurable number of cache banks. The bank mapping bits start at address bit 6 for a two bank design, while bits 6 and 7 are used in a four bank design. Throughout our experimentation, we vary the number of LLC banks to be either two or four, but the size and associativity remain constant. Cache lines are set to be 64 bytes.

All simulations are run with the RISC-V version of Linux kernel 6.2. For synthetic workloads, we use *BkPLL* (described in section IV-A) and *Bandwidth* from [8]. *Bandwidth* accesses a chunk of memory sequentially, striding at a step size of a cache line. Both workloads can be configured to perform either read or write accesses. Lastly, the San Diego Vision Benchmark Suite (SD-VBS) [32] with CIF input format is used for real-world evaluation.

To ensure all slowdowns are solely due to bank contention and not impacted by set conflict misses, we apply the PALLOC patch to the Linux kernel [23] to enable cache set partitioning. Using PALLOC, we create two partitions dividing the LLC of 1MB into equal segments of 512KB each. We assign one partition to victim tasks and one partition to best-effort (attacker) tasks.

B. SD-VBS Profiling

To guide our evaluation, we first profile the workloads from SD-VBS to find each workload's LLC bandwidth and bank access pattern. With our implemented per-bank monitoring interface, we collect these results on a single-core BOOM system with a four bank LLC. It should be noted that we exclude *multi_ncut* due to long simulation times.

Workload	LLC Read B/W	LLC Write B/W
Disparity	**2663.1**	**1330.6**
MSER	**967.9**	**270.7**
Sift	356.9	90.6
Stitch	**795.1**	**405.1**
Localization	55.9	0.326
Tracking	405.8	173.1
SVM	179.6	45.5

TABLE III: SD-VBS LLC bandwidth characteristics (MB/s)

Table III shows the collected bandwidth results. From this, we select *Disparity*, *MSER* and *Stitch* for best-effort task evaluation in section VI-F, as workloads that do not make frequent accesses to the LLC will not be noticeably affected by regulation. Through experimentation, we determine 700MB/s to be a suitable bandwidth threshold.

Workload	Bank 1	Bank 2	Bank 3	Bank 4
Disparity	5723148	5694269	5679896	5693761
MSER	476464	467716	466571	468930
Sift	1938519	1908043	1868291	1922350
Stitch	3867787	3821181	3786533	3896458
Localization	**309455**	**1843**	**1647**	**1795**
Tracking	**496624**	**252594**	**251577**	**259686**
SVM	540773	505279	557619	525794

TABLE IV: SD-VBS per-bank LLC access counts

Table IV shows the per-bank access counts of the SD-VBS workloads across the four cache banks. In our simulated design, a four-bank LLC uses address bits 6 and 7 to index the banks. Of the seven workloads, *Disparity, MSER, Sift* and *Stitch* have an even access spread across all banks. On the other hand, *Localization* and *Tracking* have heavier traffic to specific banks. *Localization* specifically sees 58× more accesses directed at bank one than the other three banks combined and 187× more accesses than the least accessed bank (bank two). *Tracking* sees 2× more accesses directed at bank one than the least accessed bank. These results show that, in most benign (not malicious) workloads, requests to the shared cache are generally distributed evenly across the cache banks, although there are some notable exceptions.

[*] https://github.com/riscv-boom/riscv-boom/issues/690

As such, if we use the all-bank regulation approach to defend against potential cache-bank DoS attacks, which target only one bank, we significantly under utilize the cache bandwidth when benign workloads are executed on the throttled best-effort cores, as we will show later in this section.

C. Cache Bank-Aware DoS Attack on FireSim

To begin our experimentation, we first mount the cache bank-aware DoS attack [12] in our simulation environment, establishing the maximum base-line slowdown for our setup. These results are collected on a system with two banks in the LLC.

The experiments are set up as follows. We utilize the *BkPLL* workload described in Section IV-A as our victim task. The victim is configured to perform reads (denoted as *BkPLLRead*), has a working-set-size (WSS) of 128KB and is executed on the BOOM core. The Mempress attackers are configured to each have a WSS of 64KB. We first run the victim in isolation and measure its performance. The attackers are then co-run with the victim in order to see the attackers impact on the victim's performance. The attackers are applied in the two following scenarios:

1) **Diff. Bank:** The attackers and the victim target different (disjoint) cache banks (victim: bank 0, attacker: bank 1)
2) **Same Bank:** Both the attackers and the victim target the same cache bank (both attacker and victim: bank 0).

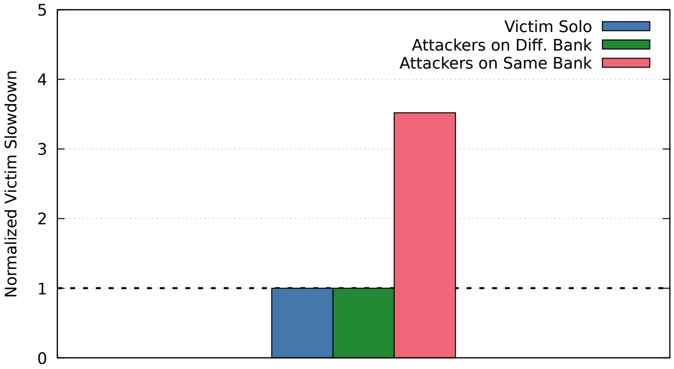


Fig. 7: Impact of cache bank-aware DoS attack on synthetic read victim in the FireSim platform. The bank attackers target is varied.

Figure 7 shows the results. Note first that, when the attackers and victim target separate banks, the victim experiences no slowdown. Yet, when the same bank is targeted the victim experiences a 3.52× slowdown from the write attackers. The results are very similar to what we have observed on the BeagleV platform in Section IV-A, demonstrating the validity of our evaluation setup. To parallel the conclusion on the real platforms, we observe complete temporal isolation when the victim and the attackers target different banks. This confirms that any contention created by the attacker is at the *bank level*, not the interconnect (bus) level.

Three key takeaways are: (1) each cache bank should be considered as an independent shared resource, which has limited bandwidth. If the bandwidth is over-saturated, then contention occurs and the subsequent requests to the bank will be delayed; (2) the targeted bank DoS attack is effective because it saturates the bank's limited bandwidth; (3) Using bandwidth regulation to mitigate contention by preventing bandwidth saturation will be effective.

D. Evaluation of Hardware Bandwidth Regulation

In this experiment, we evaluate the effectiveness of BRU's bandwidth regulation in providing temporal isolation to the victim task in the presence of cache bank DoS attackers.

For this experiment, we use all-bank regulation as implemented in the baseline BRU [16]. As discussed earlier, BRU allows for cores to be regulated alone or in groups using *domains*. Using this capability, we create a "real-time" domain for the victim task and a "best-effort" domain for the attackers. We then assign the BOOM core to the real-time domain and the two other Rocket cores, enhanced with the Mempress traffic generators, to the best-effort domain. As in Section VI-C, Mempress instances are configured to generate overwhelming traffic to cache bank 0, to simulate the worst-case. For the victim tasks, we use *BkPLLRead* (synthetic) and *Disparity* (real-world), both configured to target cache bank 0. We vary the best-effort (attacker) domain's bandwidth budget from 640MB/s to 15.36GB/s, measuring the slowdown that each victim experiences normalized to the solo victim run (no attackers). The budget is set by programming the *Regulation Period Register* to 400 cycles (400 ns in our setup), and increasing the best-effort domain's *Access Budget Register* from 16 accesses (640MB/s) per-period to 384 accesses (15.36GB/s) per-period. Unless otherwise mentioned, all subsequent experiments use a regulation period of 400 cycles.

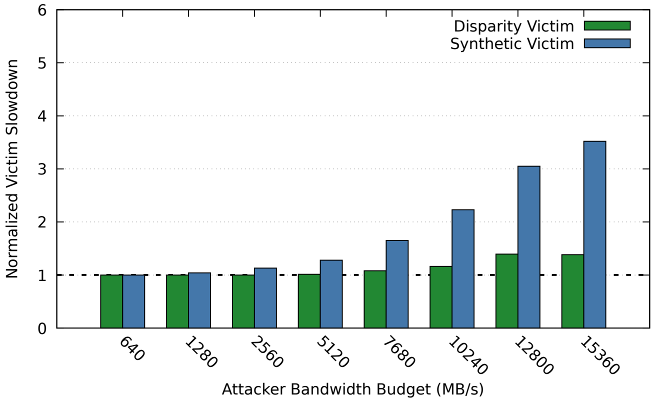


Fig. 8: Impact of increasing attacker bandwidth budget on BkPLLRead and Disparity. The WSS of attackers is 64KB.

Figure 8 shows the results. As we can see, up to an attacker budget of 1.28GB/s, the BkPLLRead victim experiences a 1.03× slowdown, which is small and acceptable in may applications. Beyond this threshold, however, the victim's execution begins to be impacted, increasing to 1.12× at a budget of 2.56GB/s and growing to 3.52× at 15.36GB/s. Note that 15.36GB/s is bigger than the observed peak cache memory bandwidth of the attackers, which is effectively identical to not using the bandwidth regulation at all. When we repeat the experiment with *Disparity* as the victim, the performance degrades at a slower rate, peaking only at 1.39× slowdown when the attackers are allotted their full bandwidth. This is because Disparity, unlike BkPLLRead, generates fewer accesses to the shared cache that are more evenly distributed among the cache banks. In other words, Disparity is not the worst-case and its performance impact from the cache bank attack will be upper bounded by that of the BkPLLRead victim. Since Disparity has the highest measured bandwidth of the SD-VBS workloads (see section VI-B), all other workloads are similarly upper bounded.

The key takeaways are (1) cache bandwidth regulation can effectively regulate the attackers to provide worst-case slowdown guarantees for the victim; (2) the regulation budget should be set based on the worst-case scenario when both the attackers and the victim target one single cache bank.

E. All-Bank vs. Per-Bank Regulation

In the following experiments, we evaluate how different bandwidth regulation methods impact the performance of the victim and the attackers.

We first show the impact of the all-bank and per-bank regulation methods in providing isolation guarantees to the victim task in the presence of the co-running cache-bank DoS attacks.

The experiment setup is the same as before: the victim (BkPLLRead) runs on the real-time domain and the attackers run on the best-effort domain, which is regulated. The regulation budget of the best-effort domain is configured at 1.28GB/s (found to be the maximum allowable budget in the previous experiment) in both all-bank and per-bank regulation methods. Note that under per-bank regulation, each bank receives the budget of 1.28GB/s.

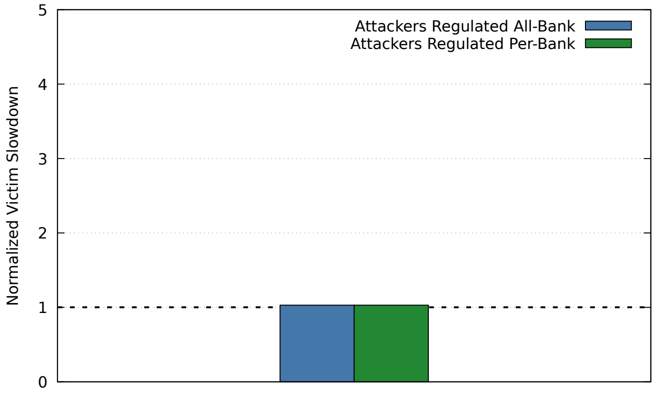


Fig. 9: Normalized slowdown of the victim when a 1.28GB/s regulation budget is applied to throttle write attackers under both all-bank and per-bank regulations.

Figure 9 shows the results. When running without regulation, we see the same 3.52× slowdown of the victim as was shown in the previous section. On the other hand, for both regulation schemes, the victim sees only a 1.03× slowdown with regulated attackers. This is because in both per-bank and all-bank regulation methods, only one cache bank is stressed by the attacks and the requests to the same bank are charged equally in both regulation methods.

The results demonstrate that all-bank and per-bank bandwidth regulation methods are identical in protecting the victim in the worst-case (i.e., the cache bank DoS attackers are running on the best-effort domain). However, they will have very different effects in non worst-case scenarios as we will show in the following.

Next, we evaluate the throughput impact of the regulation methods on the regulated cores. For this, we use the *Bandwidth* workload from [8] as described in section VI-A. We configure the workload to perform read accesses with a WSS of 128KB (4× the L1 size). The workload is pinned to the system's BOOM core. We measure the slowdown normalized to a non-regulated run of the workload. This experiment is performed on four different LLC designs as follows:

1) All-bank regulation with two LLC banks.
2) Per-bank regulation with two LLC banks.
3) All-bank regulation with four LLC banks.
4) Per-bank regulation with four LLC banks.

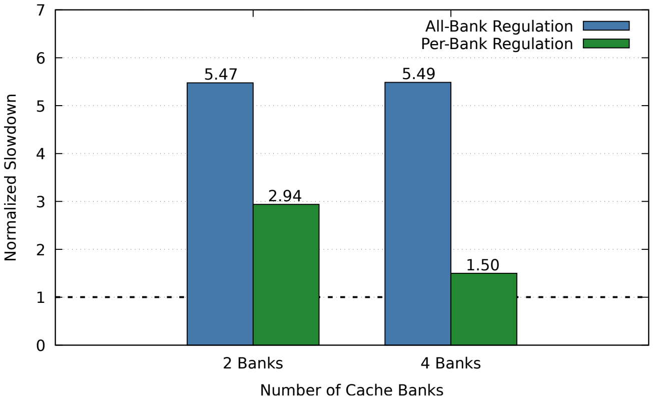


Fig. 10: Normalized slowdown of *Bandwidth* using per-bank and all-bank regulations on two and four bank cache configurations. Regulation budget is 1.28GB/s. Workload is pinned to BOOM core.

Figure 10 shows the results. For the synthetic best effort task, regulating the entire cache as one unit (all-bank) results in performance degradation of 5.47× in the two-bank case and 5.49× in the four-bank case. In contrast, per-bank regulation sees a 2.94× and a 1.50× degradation in the two and four-bank cases, respectively. To directly compare, per-bank sees a 1.86× and 3.66× improvement over all-bank in the respective cases. Recall that this improvement is due to per-bank regulation allotting each bank a budget of 1.28GB/s. It should be noted that one would expect a 2× difference in the two bank cases and a 4× difference in the four bank cases. Of course, our prototype has some inefficiencies, however we deem these acceptable as the benefits of per-bank regulation are still clear.

From this synthetic experiment, we draw two major conclusions. First, per-bank regulation demonstrates a clear improvement in best-effort task throughput compared to the overly pessimistic all-bank regulation. This throughput improvement is achieved all while guaranteeing the same temporal isolation of victim tasks. Second, performance benefits of our per-bank implementation scale effectively as the number of banks increases.

F. Impact of Per-Bank Regulation on Real-World Applications

The *Bandwidth* workload is a synthetic workload, not representative of real-world applications. We further evaluate the benefits of per-bank regulation over all-bank regulation using a set of benchmarks from SD-VBS [32] and SPEC2017 [33]. Specifically, we select *Disparity*, *MSER* and *Stitch* from SD-VBS (all *CIF* input) and *gcc*, *xalanc* and *mcf* from SPEC2017 (*ref* input). The SD-VBS benchmarks are chosen because they are relatively LLC bandwidth intensive workloads (see Section VI-B). Likewise, the SPEC2017 benchmarks are chosen as they are relatively cache sensitive and have fast simulation run times.

For these experiments, we configure a system with one BOOM core to avoid any cross-core interference. We set a regulation budget of 1.28GB/s and measure each workload's performance, computing the slowdown normalized to an unregulated run of the workload. As was done in section VI-E, we evaluate using both two bank and four bank cache designs.

Figure 11 shows the results for our selected workloads. In general, we see improvement when using per-bank regulation over all-bank regulation. Moreover, performance scales well from two to four banks, such as in *Disparity* which suffers less slowdown as more cache banks are used. Specifically, in the two-bank case, *Disparity* sees a 2.04× and 1.86× slowdown under all-bank and per-bank regulation respectively. When the cache is configured with four banks, all-bank regulation creates a 2.33× slowdown, while per-bank regulation has only a 1.38× slowdown. Thus, Disparity is an excellent example of the improvement of per-bank over all-bank regulation and the performance scalability as the number of banks increases. The results for *MSER, Stitch* and *gcc* also show similar improvement when going from an all-bank to per-bank regulation scheme.

These experiments clearly demonstrate that, as with the synthetic case, real-world workloads see noticeable improvement when using per-bank regulation over all-bank regulation. Again, it must be emphasized that this improvement is accompanied by per-bank regulation's guarantee of isolating victim tasks to the same degree as all-bank regulation. Overall, this highlights the superiority of per-bank regulation over all-bank.

G. Software vs. Hardware Bandwidth Regulation

In this experiment, we compare the software-based cache bandwidth regulation method proposed in [12] with our hardware-based bandwidth regulation.

Ideally, we would like to implement the software-based regulation approach directly on experimental platform. However, because we leverage Mempress traffic generators instead of using BOOM CPU cores for the attack, the software approach cannot be properly tested. Instead, we implement the software regulation approach on the BeagleV board, which is equipped with four RISC-V out-of-order cores (see Section IV-A) comparable to the BOOM core used in our testbed.

On the BeagleV platform, we observe up to 76× slowdown of the best-effort (attacker) tasks (throttled at 100MB/s) for the software regulation method to protect the victim. These results are in-line with the up to 300× slowdown reported in [12] on the Pi 4 platform. While using the hardware-based regulation methods in our FireSim setup, the worst-case slowdown of the attackers is up to 5.49× for all-bank regulation, and up to 2.94× slowdown for the per-bank regulation in 2-bank configuration. The slowdown is further reduced down 1.5× in the 4-bank configuration.

Note that the CPU performance of the BeagleV platform is on-par with that of our simulation setup. As such, we posit that the dramatic performance differences we observe come from the superior effectiveness of hardware-based regulation over the software-based one.

H. Hardware Implementation Overhead

To evaluate the cost-to-performance benefit of our per-bank design, we synthesize a quad-core BOOM SoC and perform power and area analysis. We run place and route from the Cadence Suite's Innovus tool, along with the Hammer [34] VLSI flow scripts targeting the ASAP 7nm technology node [35], to characterize the overhead and compare to the all-bank implementation in [16].

Implementation	Regulation Unit (nm^2)	SoC (nm^2)	Percent
All-Bank [16]	429	465305	0.09%
Per-Bank (Ours)	1372	466248	0.29%

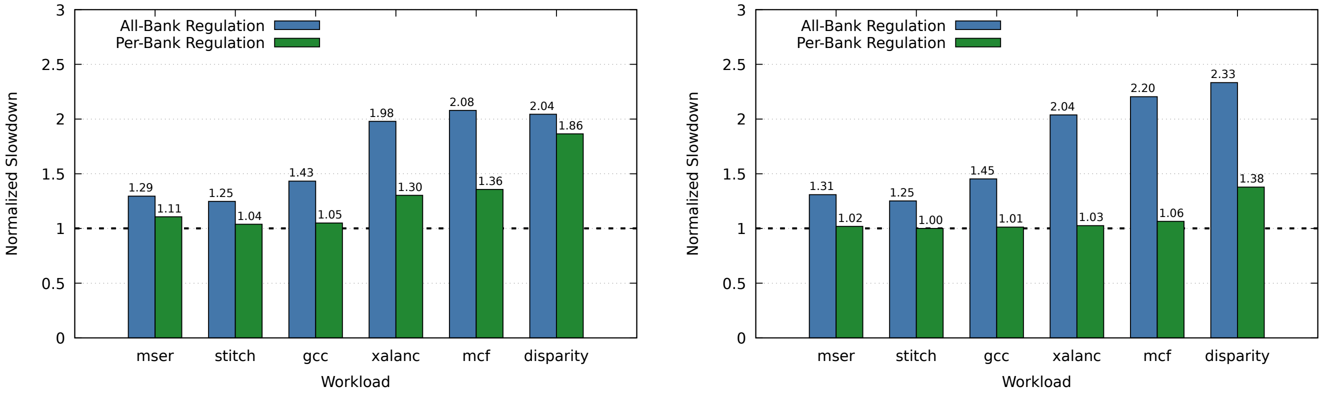
TABLE V: Comparative area analysis of the two regulation unit implementations. Percent is the area consumed by the implementation from the total SoC area.

Table V shows the area utilization of the two configurations after place and route has been completed. We find that the area overhead added in our per-bank design comes to 3.2× that of the all-bank implementation. However, it is still less than 0.3% of the entire SoC area.

Design Under Test	Total Power (mW)	Percent
SoC	110	
All-Bank [16]	0.67	0.6%
Per-Bank (Ours)	2.36	2.1%

TABLE VI: Comparative power analysis of the two regulation unit implementations. Percent is the power consumed by the implementation from the total SoC power.

Table VI shows the power analysis results. As shown, the per-bank design again consumes 3.5× that of the all-bank design, yet it is still only just over 2% of the total power. It



(a) 2 Banks

(b) 4 Banks

Fig. 11: Comparison of all-bank and per-bank regulation when running real-world workloads. Each workload is run with a regulation budget of 1.28GB/s. Slowdown is in comparison to the unregulated run.

can be stated that the area and power overhead of our per-bank design is acceptable considering its significant performance benefits seen in previous sections.

VII. Related Work

In the real-time community, correctly estimating worst-case task execution timing is of paramount importance, yet it has been difficult to do so in multicore systems due to the vast and diverse set of shared hardware resources that can dramatically impact task execution timing. Microarchitectural DoS attacks on shared hardware resources are therefore important for the real-time community to study as they can shed light on the impacts on worst-case timing. Moscibroda et al. proposed the "memory performance attack" [36], which exploits the DRAM controller's FR-FCFS [37] scheduling policy to induce contention. Attacks on shared cache space [38], bus bandwidth [39], shared cache MSHRs [8] and write-back buffers [9], shared GPU [40], and shared cache between the CPU and the integrated GPU [41] have been explored. Most recently, bank contention on multi-bank shared caches has shown to be an effective DoS attack avenue [12], which we focus on in this work. Several studies have investigated the effects of these microarchitectural attacks in actual cyber-physical systems [42], [43].

Providing strong isolation in multicore has long been a topic of intense research over the past two decades. This includes various software and hardware mechanisms to manage the shared resources [13], [44]–[53]. Broadly, these resource management studies fall into two categories: space partitioning and bandwidth throttling. Cache space partitioning has been extensively studied in the real-time community to prevent unwanted cache-line evictions of high-priority real-time tasks by lower priority tasks. Cache space partitioning can be realized in software, through page coloring [54], or in hardware, such as the way-based partitioning capabilities found in Intel RDT [14] and ARM MPAM [15].

Memory bandwidth throttling is another extensively studied mechanism for isolation. Most software-based memory bandwidth throttling techniques utilize the CPU core's performance counters to monitor the bandwidth. Then the periodic timers and interrupts regulate the allowed bandwidth of the cores at fixed time intervals [13], [53]. MemPol [55] instead utilizes a dedicated real-time micro-controller unit (MCU) to asynchronously monitor and regulate the memory traffic through polling. This approach reduces the interrupt overhead at the expense of wasting the real-time MCU. Hardware-based bandwidth regulation can eliminate such software overhead and can be enforced at a very fine granularity (in cycles rather than in milliseconds). BRU [16], Intel RDT [14], ARM MPAM [15] all provide memory bandwidth regulation capabilities in hardware.

Until recently, cache bandwidth has received little attention as it was believed to be of less importance compared to cache space partitioning or memory bandwidth. However, a recent study demonstrated its implications in multi-bank caches within high-performance multicore architectures [12]. The study proposed a software cache bandwidth throttling mechanism as a potential mitigation solution, but acknowledged the unacceptably high overhead of such a software implementation. In this work, we present a hardware solution to manage cache bandwidth in real-time systems. To the best of our knowledge, we are the first to present a hardware-based cache bandwidth regulation solution. More importantly, we are the first to propose a per-bank cache bandwidth regulation approach that can significantly improve average throughput on the regulated cores.

VIII. Conclusion

In this paper, we presented a per-bank cache bandwidth regulation approach to effectively and efficiently mitigate potential cache bank bandwidth contention. We make the observation that the contention occurs at the individual cache bank rather than at the interconnect (bus), therefore our key contribution is to apply a well-known bandwidth regulation mechanism at the cache bank level. We evaluate that this approach can effectively minimize the effect of worst-case cache bank contention while maximizing allowed cache bandwidth and guaranteeing the isolation. We implemented the proposed per-bank regulation solution in hardware by extending an open-source bandwidth regulator design. We demonstrated its effectiveness in providing isolation guarantees to critical real-time tasks in the presence of adversarial cache bank DoS attackers. Furthermore, we illustrated that our per-bank bandwidth regulation approach can significantly improve performance of throttled best-effort tasks without compromising isolation guarantees allotted to real-time tasks. Specifically, we achieved up to 3.66× throughput improvement over the baseline bank-oblivious bandwidth throttling approach. As future work, we plan to apply the proposed per-bank regulation approach to DRAM banks.

Acknowledgments

This research is supported in part by NSF grants CPS-2038923, CCF-2239020, CCF-2403013, and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] ARM, "ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3," ARM Holdings, Tech. Rep., 2024, accessed: 2024-05-09. [Online]. Available: https://developer.arm.com/documentation/100095/0003/

[2] ——, "ARM Cortex-A57 MPCore Processor Technical Reference Manual r1p3," ARM Holdings, Tech. Rep., 2024, accessed: 2024-05-09. [Online]. Available: https://developer.arm.com/documentation/ddi0488/

[3] ——, "ARM DynamIQ Shared Unit Technical Reference Manual," ARM Holdings, Tech. Rep. 100453, 2024, accessed: 2024-05-09. [Online]. Available: https://developer.arm.com/documentation/100453/0401/

[4] The Raspberry Pi Foundation, "Raspberry Pi 5 - raspberrypi.com," https://www.raspberrypi.com/products/raspberry-pi-5/, 2024, accessed: 2024-05-10.

[5] BeagleBoard.org Foundation, "BeagleV Ahead - BeagleBoard.org," https://www.beagleboard.org/boards/beaglev-ahead, 2024, accessed: 2024-05-03.

[6] NXP, "T4240RM, T4240 QorIQ Integrated Multicore Communications Processor Family Reference Manual - Reference Manual," Tech. Rep., 2017.

[7] ——, "QorIQ LX2160A Reference Manual," Tech. Rep., 2021.

[8] P. K. Valsan, H. Yun, and F. Farshchi, "Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems," in *RTAS*, 2016.

[9] M. G. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," in *RTAS*, 2019.

[10] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *RTAS*, 2020.

[11] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, "PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference," in *RTSS*, 2022.

[12] M. Bechtel and H. Yun, "Cache bank-aware denial-of-service attacks on multicore arm processors," in *RTAS*, 2023.

[13] 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 *RTAS*, 2013.

[14] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," Intel Corporation, Tech. Rep., 2024, accessed: 2024-05-09. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[15] ARM, "Arm Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification," ARM Holdings, Tech. Rep., 2024, accessed: 2024-05-09. [Online]. Available: https://developer.arm.com/documentation/ihi0099/aa

[16] F. Farshchi, Q. Huang, and H. Yun, "Bru: Bandwidth regulation unit for real-time multicore processors," in *RTAS*, 2020.

[17] "Amd virtex ultrascale+ vcu118 fpga," https://www.xilinx.com/products/boards-and-kits/vcu118.html, 2024, accessed: 2024-05-13.

[18] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *ISCA*, 2018.

[19] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, *Multi-Core Cache Hierarchies*. Morgan & Claypool Publishers, 2011.

[20] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *EuroSys*, 2019.

[21] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *DSD*. IEEE, 2015.

[22] The Raspberry Pi Foundation, "Raspberry Pi 4 - raspberrypi.com," https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, 2024, accessed: 2024-05-10.

[23] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms," in *RTAS*, 2014.

[24] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012.

[25] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, 2020.

[26] SiFive, "SiFive TileLink Specification," 2017, accessed: 2024-05-10.

[27] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[28] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[29] "Mempress," https://github.com/ucb-bar/mempress/tree/main, 2024, accessed: 2024-05-09.

[30] "RoCC Accelerators," https://chipyard.readthedocs.io/en/stable/Customization/RoCC-Accelerators.html, 2024, accessed: 2024-05-13.

[31] "SiFive Inclusive Cache," https://github.com/chipsalliance/rocket-chip-inclusive-cache/tree/main, 2024, accessed: 2024-05-09.

[32] 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 *IISWC*, 2009.

[33] "SPEC CPU2017," https://www.spec.org/cpu2017.

[34] H. Liew, D. Grubb, J. Wright, C. Schmidt, N. Krzysztofowicz, A. Izraelevitz, E. Wang, K. Asanović, J. Bachrach, and B. Nikolić, "Hammer: a modular and reusable physical design flow tool: invited," in *DAC*, 2022.

[35] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, 2016. [Online]. Available: https://doi.org/10.1016/j.mejo.2016.04.006

[36] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security Symposium*, 2007.

[37] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *ACM SIGARCH Computer Architecture News*, 2000.

[38] G. Keramidas, P. Petoumenos, S. Kaxiras, A. Antonopoulos, and D. Serpanos, "Preventing denial-of-service attacks in shared cmp caches," in *SAMOS*, 2006.

[39] D. H. Woo and H. Lee, "Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors," in *CMP-MSI*, 2007.

[40] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. Smith, "Making Powerful Enemies on NVIDIA GPUs," in *RTSS*, 2022.

[41] M. Bechtel and H. Yun, “Denial-of-Service Attacks on Shared Resources in Intel’s Integrated CPU-GPU Platforms,” in *ISORC*, 2022.

[42] A. Li, J. Wang, S. Baruah, B. Sinopoli, and N. Zhang, “An empirical study of performance interference: Timing violation patterns and impacts,” in *RTAS*, 2024.

[43] M. Bechtel and H. Yun, “Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study,” *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1753–1766, 2024.

[44] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-Time Cache Management Framework for Multi-core Architectures,” in *RTAS*, 2013.

[45] H. Kim, A. Kandhalu, and R. Rajkumar, “A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems,” in *ECRTS*, 2013.

[46] Y. Ye, R. West, Z. Cheng, and Y. Li, “Coloris: a Dynamic Cache Partitioning System Using Page Coloring,” in *PACT*, 2014.

[47] 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*, 2017.

[48] S. Roozkhosh and R. Mancuso, “The Potential of Programmable Logic in the Middle: Cache Bleaching,” in *RTAS*, 2020.

[49] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, “E-WarP: a System-wide Framework for Memory Bandwidth Profiling and Management,” in *RTSS*, 2020.

[50] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun, “Deterministic Memory Abstraction and Supporting Multicore System Architecture,” in *ECRTS*, 2018.

[51] W. Ali and H. Yun, “RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems,” in *RTAS*, 2019.

[52] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, “Holistic Resource Allocation for Multicore Real-Time Systems,” in *RTAS*, 2019.

[53] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, “Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores,” in *RTAS*, 2022.

[54] S. Mittal, “A survey of techniques for cache partitioning in multicore processors,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, 2017.

[55] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, “Mempol: Policing core memory bandwidth from outside of the cores,” in *RTAS*, 2023.