# Per-Bank Memory Bandwidth Regulation for Predictable and Performant Real-Time Systems

Connor Rudy Sullivan, Amin Mamandipoor, Cole Ridge Strickler, Heechul Yun
*Electrical Engineering and Computer Science*
*The University of Kansas*
Lawrence, KS, USA
{connor.sullivan13, aminm, colestrickler, heechul.yun}@ku.edu

***Abstract*—Modern multicore system-on-chips (SoCs) share off-chip DRAM across cores, where bank-level interference can significantly degrade performance and threaten real-time guarantees. While prior work has focused on per-core bandwidth regulation, these approaches treat main memory as a monolithic resource and overlook DRAM's inherent bank-level parallelism.**

**We show that DRAM interference is fundamentally a bank-level phenomenon. We characterize the guaranteed bandwidth of modern DRAM, demonstrate that it remains effectively constant across generations, and show how this limitation can be exploited by single-bank attacks. These results highlight the need for bank-aware memory management for predictable and efficient real-time systems.**

**We design and implement a novel per-bank memory bandwidth regulator in an open-source RISC-V SoC and evaluate it using FireSim with both synthetic and real-world workloads. Our evaluation demonstrates that per-bank regulation effectively mitigates adversarial bank contention and achieves a 5.74× average throughput improvement for best-effort workloads over traditional bank-oblivious approaches while providing the same-level of performance isolation guarantees for real-time workloads.**

## I. Introduction

Modern multicore real-time systems face a fundamental challenge: unregulated sharing of DRAM can cause severe and unpredictable interference among applications. When multiple cores concurrently access memory, their requests contend for limited DRAM resources, creating interference that can significantly degrade application performance and undermine predictable timing guarantees.

A substantial body of prior work has sought to mitigate memory interference through bandwidth regulation. State-of-the-art solutions [1]–[3] aim to provide isolation guarantees by enforcing per-core bandwidth budgets at varying levels of granularity. These approaches have been effective in reducing interference, but they treat DRAM as a monolithic resource, ignoring its inherently parallel, banked organization. This structure-unaware abstraction leads to overly pessimistic performance because it regulates bandwidth globally rather than at the actual points of contention.

Our work is motivated by a simple but critical observation: DRAM interference is fundamentally a bank-level phenomenon. Requests to different banks can proceed largely in parallel, whereas requests targeting the same bank must be serialized. As a result, contention within a single bank dominates worst-case interference.

This observation suggests a fundamental shift in regulation strategy: rather than controlling aggregate DRAM bandwidth, regulation should operate at the granularity of individual banks. Aligning regulation with DRAM's internal structure enables strong temporal isolation while preserving parallelism and improving overall system throughput.

In this paper, we analytically and empirically characterize the worst-case, guaranteed memory bandwidth [4] of several generations of DRAM on representative commercial off-the-shelf multicore platforms. We show that this guaranteed bandwidth of modern DRAM is surprisingly small—and effectively constant across generations and configurations—because it is dictated by the worst-case bandwidth of a single DRAM bank.

Building on this insight, we construct a single-bank memory performance attack that induces severe cross-core slowdowns across hardware platforms, even though the attacker generates minimal aggregate traffic. This result exposes a fundamental weakness in existing bandwidth-regulation approaches and demonstrates the need for new regulation approaches aligned with DRAM's internal bank structure.

We propose a per-bank memory bandwidth regulation approach that enables predictable and high-performance use of shared DRAM in real-time systems. Our approach applies regulation at the bank level, scaling the throughput of best-effort tasks with the number of banks while preserving the temporal isolation required by real-time workloads.

We implement a novel per-bank bandwidth regulator design within an open-source RISC-V SoC framework [5] and evaluate it on a Xilinx UltraScale+ VCU118 FPGA [6] using FireSim [7]. We study the effects of bank-level contention attacks and evaluate the regulator's ability to protect real-time tasks from such interference. Our results show that per-bank regulation effectively neutralizes bank contention attacks while providing, on average, a 5.74× throughput improvement for best-effort tasks over an all-bank regulation approach.

In summary, we make the following contributions:

- We characterize and empirically evaluate the guaranteed memory bandwidth of multiple DRAM technologies on representative multicore platforms. This includes reverse-engineering sophisticated XOR-based address mapping schemes and developing a precise methodology for measuring guaranteed bandwidth.

- We demonstrate the limitations of existing memory-bandwidth regulation solutions through a single-bank memory performance attack and evaluate its impact on real hardware through extensive empirical analysis.
- We design and implement a per-bank memory bandwidth regulator within a RISC-V SoC, evaluate it using a cycle-accurate, FPGA-accelerated full-system simulator, and show its ability to provide strong isolation and significant performance improvements.
- We release our tools, benchmarks, evaluation scripts, and proposed per-bank DRAM regulator hardware design as open source.[1]

The remainder of this paper is organized as follows. Section II provides background information. Section III characterizes guaranteed memory bandwidth. Section IV presents and evaluates a memory performance attack. Section V introduces our regulator design, with implementation details in Section VI and evaluation results in Section VII. Section VIII discusses broader implications. Section IX reviews related work, and Section X concludes the paper.

## II. Background

In this section we discuss modern DRAM structure, DRAM bandwidth regulation, and guaranteed memory bandwidth.

### A. DRAM Structure

Modern DRAM systems are organized hierarchically, in channels, ranks, and banks, and achieve high performance via parallelism. A primary driver of parallelism is the DRAM bank—a 2D memory array composed of rows and columns [8].

Memory accesses to a DRAM bank proceed through a sequence of three commands: ACTIVATE, CAS, and PRECHARGE. An ACTIVATE command opens a row by transferring its contents into the row buffer of the bank. A subsequent CAS command reads or writes the desired data from the row buffer. Additional accesses to the same row can be serviced directly from the row buffer through repeated CAS commands. However, when an access targets a different row, a PRECHARGE command must first be issued to write the buffer contents back to the cell array before the next row can be activated. Because these commands are issued per bank, multiple banks can be accessed in parallel, allowing the DRAM subsystem to exploit significant bank-level parallelism under typical workloads.

Memory controllers receive memory requests from the CPU and other clients, and schedule them on DRAM while respecting numerous timing parameters defined by DRAM standards [9]–[11]. Modern memory controllers maintain separate read and write transaction queues and commonly use FR-FCFS scheduling [12] to maximize overall throughput. The controller also manages the read/write turnaround time of the shared bidirectional data bus. Switching the bus from write mode to read mode incurs the *tWTR* (Write-to-Read) timing penalty. To amortize this cost, controllers issue writes in large batches, a behavior governed by high and low watermarking schemes [13]. When the number of outstanding writes exceeds the high watermark, the controller begins draining writes from the queue.

[1]https://github.com/CSL-KU/per-bank-dram-bru

### B. Memory Bandwidth Regulation

Memory bandwidth regulation is a well-established approach to tame the cross-core memory interference on multicore. Both software- and hardware-based solutions have been explored. Software-based approaches typically rely on architectural performance counters and stalling cores when they exceed a memory-access budget; MemGuard [1] is the canonical example.

Hardware-based regulators, in contrast, implement bandwidth enforcement directly in hardware, enabling finer-grained control. Intel Memory Bandwidth Allocation (MBA) [14] is a representative example, which has been widely deployed in recent Intel server processors.

Unfortunately, existing approaches treat the entire DRAM as a monolithic resource and overlook its internal structure and bank-level parallelism. As a result, they either provide weak worst-case guarantees or incur unnecessary inefficiencies and bandwidth waste.

In [15], the authors characterize such structure-oblivious regulation schemes as *all-bank* regulation and propose a structure-aware alternative, *per-bank* regulation, showing its benefits for managing bandwidth within banked last-level caches (LLCs).

Since DRAM exposes even more parallelism than a banked LLC, extending per-bank regulation principles to main memory promises even greater improvements—an opportunity we explore in this work.

### C. Guaranteed Memory Bandwidth

While modern DRAM scaling has achieved impressive increases in peak bandwidth, these gains are primarily enabled by increasing the number of concurrently accessible banks. However, the worst-case bandwidth of an individual bank has remained largely stagnant.

Following the definition in [4], we call the bandwidth achievable in the worst case the *guaranteed memory bandwidth*. The worst case arises when all requests access different rows within the same DRAM bank, repeatedly causing row misses. In such scenarios, two consecutive requests must be separated by $tRC$ (row-cycle time) [9]. Consequently, the guaranteed bandwidth $BW_g$ can be expressed as:

$$BW_g = \frac{64}{tRC} \tag{1}$$

where each request corresponds to one 64-byte cache line from the CPU. Note that the $tRC$ parameter is largely determined by the intrinsic electrical properties of the DRAM cell array (capacitance, resistance, and sensing delay). Therefore, its value has remained relatively constant—typically 45 to 65 ns—across successive memory generations [9]–[11], [16], even as interface speeds and parallelism have increased.

## III. Measuring Guaranteed Memory Bandwidth

In this section, we present a DRAM bank-mapping reverse-engineering tool and an evaluation methodology that can accurately measure the guaranteed memory bandwidth.

To accurately measure the guaranteed memory bandwidth, one must generate successive memory requests to different rows within the same DRAM bank. In some recent works [2], [17], the authors attempted to measure the guaranteed bandwidth[2] by increasing the step size of memory accesses in powers of two, to trigger consecutive row misses within the same DRAM bank. Unfortunately, this approach works only if the physical address bits are mapped to DRAM banks via a simple direct mapping (i.e., a physical address bit directly corresponds to a DRAM bank bit). When the DRAM controller employs sophisticated XOR-based address mapping schemes [18], as seen in many x86 and high-end ARM systems, which map DRAM banks by XORing multiple physical address bits, simply probing different step sizes fails to generate the desired worst-case memory access patterns, resulting in significant overestimations [17].

### A. Reverse Engineering of DRAM Bank Map Functions

Our DRAM bank-mapping reverse-engineering tool is a modified version of DRAMA [19], an experimental framework for reverse-engineering the address-mapping functions used by DRAM controllers. It measures pairwise access latencies between different memory addresses to identify those that map to the same DRAM bank, which incur longer access times due to row-conflict penalties. Using this information, DRAMA reconstructs the mapping between physical address bits and DRAM banks through statistical and linear-algebraic analysis over the Galois Field with two elements [20], or GF(2) for short. DRAMA was originally designed for x86 platforms, as it relies on x86-specific instructions—`CLFLUSH` and `RDTSC`—for cache management and timing.

To enable DRAMA on ARM architectures, we made several modifications. First, we replaced the x86 `CLFLUSH` instruction with ARM's cache maintenance operation `DC CIVAC`, which invalidates cache lines by virtual address to the point of coherency so that memory accesses are served from DRAM rather than from caches. Second, ARM lacks a direct equivalent to x86's `RDTSC` instruction, as its closest counterpart, reading from the virtual timer counter (`CNTVCT_EL0`), does not provide the fine-grained resolution needed to distinguish row-conflict timings. To overcome this limitation, we employ a signal amplification technique that performs repeated accesses to the same address pairs and measures the aggregate access time. This amplification compensates for the timer's coarser granularity while maintaining sufficient timing resolution to differentiate between row-buffer hits and conflicts.

In addition, we identified and fixed several major issues, including a logic error that prevented high-order address bits from being considered and a performance bottleneck caused by the GF(2) solver's exponential time complexity, which previously made successful address recovery difficult. In contrast, our modified version is much faster, runs in polynomial time, and is more robust, enabling reliable and efficient map recovery. We call our modified version *DRAMA++*[3].

### B. Hardware Platforms and Found DRAM Bank Maps

For our study, we used three ARM platforms—Raspberry Pi 4, Raspberry Pi 5, and Jetson Orin AGX—and one x86 platform, an Intel i7-8700 desktop. Table I summarizes the key characteristics of these hardware platforms, including the DRAM bank address-mapping information obtained using our reverse-engineering tool. The DRAM organization and the $tRC$ timing parameters of the first three platforms were obtained from the datasheets of the actual memory modules that we confirmed were used in our systems. For the Jetson Orin AGX, however, these parameters are our best estimates, as we could not find publicly available information and the memory chips were not visible without disassembling the entire system. Note that all platforms employ different memory technologies—LPDDR4, LPDDR4X, DDR4, and LPDDR5—and the number of DRAM banks also varies significantly, from 8 to 256, representing a wide spectrum of systems.

The Raspberry Pi 4 and Raspberry Pi 5 platforms employ simple direct mapping schemes that use a subset of physical address bits to directly map DRAM banks. On the Raspberry Pi 4, physical address bits 12, 13, and 14 are mapped to three DRAM bank address bits in the memory controller, corresponding to the eight banks of the LPDDR4 memory chip [21]. The Raspberry Pi 5[4] uses physical address bits 12, 13, 14, and 31 to map its sixteen banks of its dual-die LPDDR4X memory [24].

In contrast, the Intel i7-8700 desktop and the Jetson Orin AGX employ more complex XOR-based address-mapping schemes. In the Intel desktop platform, four 8 GB dual-rank DDR4 DIMMs are installed, resulting in a total of 128 banks that are mapped using seven XOR-based mapping functions derived from multiple physical address bits. The Jetson Orin AGX platform, with a total of 256 banks, uses even more complex XOR-based mapping functions.

### C. Bank-Aware Parallel Linked-List (PLL) Benchmark

We use a modified version of the Parallel Linked-List (PLL) benchmark [25] to measure the guaranteed bandwidth. PLL traverses multiple independent linked lists concurrently, where pointer-chasing dependencies within each list ensure that only one outstanding memory request exists per list.

Our modification in this work adds support for XOR-based address-mapping functions when creating linked-list entries, allowing them to be allocated on specific user-controlled DRAM banks.

[2] In [2], [17], the term *sustainable memory bandwidth* was used instead, but its definition is identical to that of the guaranteed memory bandwidth.

[3] https://github.com/CSL-KU/drama-pp

[4] Raspberry Pi 5's bootloader supports four different DRAM bank maps [22]. Recently, the default map was changed to map higher-order bits to banks [23]. Our mapping corresponds to the older default map function.

| Platform | Raspberry Pi 4 | Raspberry Pi 5 | Intel Coffee Lake | Jetson Orin AGX |
|---|---|---|---|---|
| **CPU** | 4 × Cortex-A72 | 4 × Cortex-A76 | i7-8700 (6C/12T) | 12 × Cortex-A78E |
| **DRAM** | 2GB LPDDR4-3200<br>1-die × 8-banks<br>= 8 banks | 4GB LPDDR4X-4267<br>2-die × 8-bank<br>= 16 banks | 4 × 8GB DDR4-2133<br>4 × 2-rank × 16-banks<br>= 128 banks | 4 × 8GB LPDDR5-6400<br>4 × 4-die × 16-banks<br>= 256 banks |
| **Peak B/W (GB/s)** | 12.8 | 17.1 | 34.1 | 204.8 |
| **tRC (ns)** | 60 | 60 | 47 | 60 |
| **Found Bank Map** | b0:12<br>b1:13<br>b2:14 | b0:12<br>b1:13<br>b2:14<br>b3:31 | b0:7⊕14<br>b1:15⊕20<br>b2:16⊕21<br>b3:17⊕22<br>b4:18⊕23<br>b5:19⊕24<br>b6:8⊕9⊕12⊕13⊕18⊕19 | b0:11⊕14⊕16⊕20⊕21⊕22⊕33<br>b1:9⊕11⊕12⊕16⊕19⊕23⊕27⊕28<br>b2:12⊕13⊕18⊕22⊕25⊕29⊕30⊕31<br>b3:10⊕11⊕12⊕17⊕19⊕20⊕23⊕32<br>b4:10⊕11⊕13⊕14⊕18⊕27⊕28⊕34<br>b5:11⊕12⊕13⊕16⊕19⊕24⊕33⊕35<br>b6:10⊕13⊕7⊕21⊕24⊕25⊕26⊕29⊕34<br>b7:14⊕15⊕17⊕21⊕25⊕28⊕31⊕34⊕35 |

TABLE I: Evaluated hardware platforms.

**Algorithm 1** Physical Address to Bank Conversion

```
function PADDR_TO_BANK(paddr)
    bank ← 0
    for i = 0 to |functions| − 1 do
        res ← 0
        for each bit_pos in functions[i] do
            res ← res ⊕ ((paddr ≫ bit_pos)&1)
        end for
        if res = 1 then
            bank ← bank | (1 ≪ i)
        end if
    end for
    return bank
end function
```

Algorithm 1 shows the physical address to DRAM bank conversion function in our modified PLL benchmark. Here, $functions[i]$ refers to the XORed physical address bits that determine the $i$-th DRAM bank bit ($b_i$ in Table I). For example, AGX's $functions[0] = \{11, 14, ..., 33\}$.

By varying the number of lists, the PLL benchmark explores the degree of exploitable memory-level parallelism (MLP) available in the memory hierarchy. This parallelism is influenced by the core's ability to generate concurrent requests and the inherent parallelism of the DRAM banks.

However, when the number of DRAM banks available for access is constrained, the exploitable MLP is likewise limited, which in turn affects performance. If it is restricted to a single bank, all parallel requests from the core are directed to that bank, creating a severe bottleneck. Moreover, because the addresses are randomly shuffled, they are likely to target different rows within the bank, producing the worst-case access pattern required to measure the guaranteed bandwidth, as defined in Section II-C.

### *D. Evaluation Results*

Using the modified PLL benchmark, we conduct the following experiments to measure the guaranteed bandwidth of each hardware platform. For each platform, we swept the MLP (i.e., the number of lists $L$) from 1 to 16 and measured the aggregate bandwidth under four different configurations:

- *1×pll (SB)*: 1 instance targeting 1 (single) bank
- *4×pll (SB)*: 4 instances each targeting the same 1 bank
- *1×pll (AB)*: 1 instance targeting all banks
- *4×pll (AB)*: 4 instances each targeting all banks

Figure 1 shows the results. Note that X-axis represents the number of linked-lists $L$, which determines the MLP of the PLL benchmark. Y-axis is the measured bandwidth reported by the PLL. For 4 instance results, we report the aggregate bandwidth. In all experiments, each PLL instance was pinned to a dedicated core.

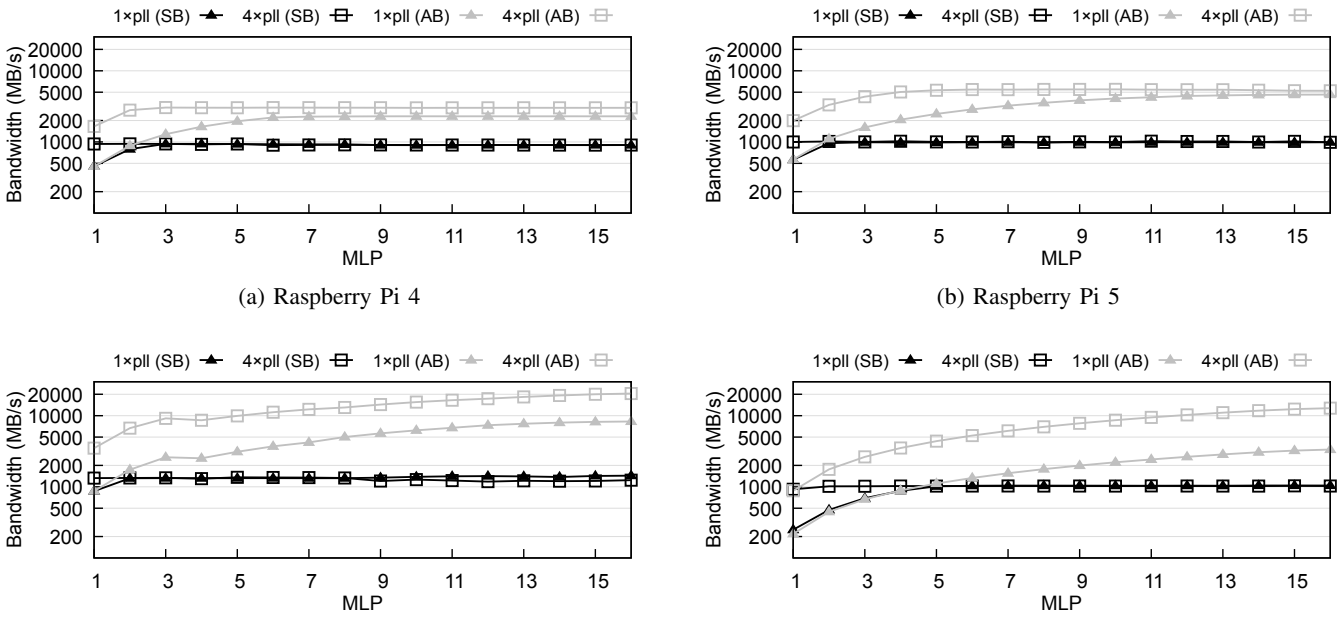
Let us first consider *1×pll (AB)* where the PLL benchmark can access all available DRAM banks on the platform. In this configuration, as the MLP increases, the measured bandwidth also increases—up to a point. The increase in bandwidth represents the core's (or LLC's) ability to extract memory-level parallelism as the MLP is not constrained by the parallelism of the DRAM banks. In *1×pll (SB)*, however, the bandwidth quickly saturates at around 1 GB/s. This is because a single DRAM bank cannot provide sufficient parallelism even though the core generates parallel requests.

Similarly, in *4×pll (AB)*, the aggregate bandwidth further improves before saturation. This means that the multiple cores can extract additional parallelism from the memory hierarchy. In *4×pll (SB)*, however, adding more instances (i.e., more cores) does not increase aggregate bandwidth at all (Note that both (SB) lines are virtually indistinguishable except at MLP 1). This is because the maximum bandwidth is constrained by a single DRAM bank's maximum—i.e., the guaranteed bandwidth.

| Platform | Pi 4 | Pi 5 | Intel | AGX |
|---|---|---|---|---|
| **Theory** | 1067 | 1067 | 1362 | 1067 |
| **Measured** | 939 | 945 | 1324 | 1042 |

TABLE II: Guaranteed bandwidth (MB/s)

Table II compares the theoretical guaranteed bandwidth (calculated using Eq. 1) with our measured results—that is, the maximum observed bandwidth in either 1×pll (SB) or 4×pll (SB). Note that both the calculated and measured values are closely aligned, at around 1 GB/s on ARM platforms using LPDDR memories and 1.3 GB/s on the Intel platform with DDR4 memory. This close alignment supports the validity of our measurement method.




(a) Raspberry Pi 4

(b) Raspberry Pi 5

(c) Intel Coffee Lake

(d) Jetson Orin AGX

Fig. 1: Measured bandwidth (logarithmic scale) of *PLL* benchmarks as a function of MLP

In summary, we have demonstrated that despite the high degree of parallelism and high peak bandwidth in modern memory systems, the guaranteed memory bandwidth remains small and stable, as it is determined by the worst-case bandwidth of a single DRAM bank. In the following, we will explore how this observation can be exploited to induce severe memory performance interference in multicore systems.

## IV. Single-Bank Memory Performance Attack

In this section, we investigate how the execution time of an application can be affected by interfering memory requests from co-running applications at the DRAM bank level.

We begin by posing the following question: *What is the most effective way to delay an application accessing DRAM?*

Suppose the application under study, which we call the *victim*, generates eight concurrent memory requests to an eight-bank DRAM. Without interference, the victim's requests can ideally be processed in parallel—one per bank—taking one unit of time in total. Now suppose that an interfering application, which we call the *attacker*, also generates eight parallel requests simultaneously, potentially interfering with the victim's requests. If the attacker's requests are evenly distributed across the eight banks, the maximum delay these interfering requests can cause is bounded by the delay at a single bank—one unit time—rather than the aggregate of all banks. On the other hand, if the attacker's requests are directed to a single bank, then the worst-case delay the victim may experience is eight, as the victim's single request to that bank might be delayed by up to eight time units. In this case, even though the victim's other requests experience no delay, they have no impact on the overall execution time, which is determined by the single bank suffering the maximum delay.

From this hypothetical thought experiment, we therefore expect that the worst-case delay will occur when interfering memory requests are concentrated on a single DRAM bank—a hypothesis that we will validate next.

### A. Methodology

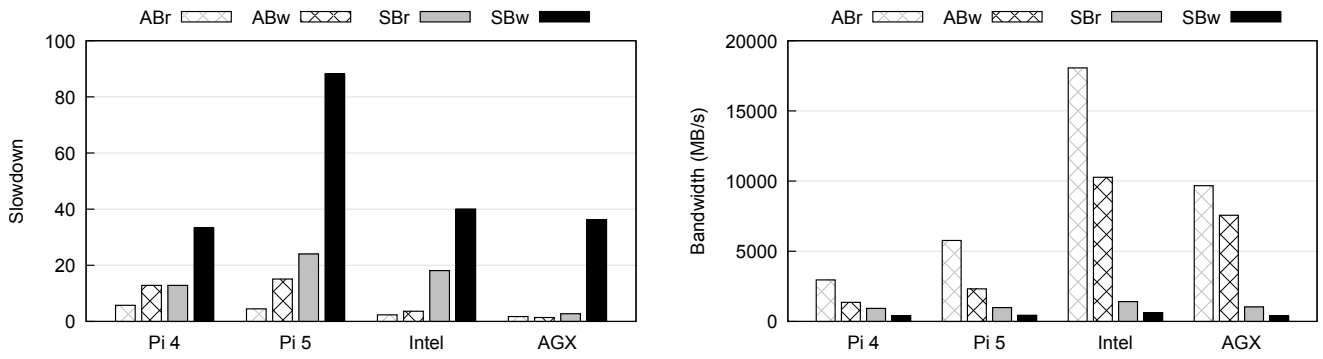
We evaluate the impact of bank-level interference using one victim process running on a dedicated core and three attacker processes running concurrently on separate cores. The attackers execute the PLL benchmark that either accesses all DRAM banks (AB) or a single bank (SB) and issues either read or write requests. We denote these configurations as *ABr* (all-bank read), *ABw* (all-bank write), *SBr* (single-bank read), and *SBw* (single-bank write), respectively.

For each experiment, we first measure the victim's execution time in isolation. We then launch three attacker processes on dedicated cores (cores 1–3) and re-run the victim on core 0 to measure its execution time under interference. We report the victim's slowdown ratio together with the aggregate bandwidth consumed by the attackers.[5]

### B. Effects on Synthetic Victim

In this experiment, we use the *Bandwidth* benchmark from IsolBench [26] as the victim. The benchmark simply performs

[5]Throughout the paper, all experiments were repeated multiple times, but we report only one data point for each as no significant run-to-run variation was observed. Our code repository includes scripts for reproduction study.




(a) Victim slowdown.

(b) Attacker bandwidth.

Fig. 2: Victim (*Bandwidth*) slowdown and attacker bandwidth consumption of all-bank (AB) and single-bank (SB) attacks.

sequential reads over a large array—twice the size of the LLC—forcing most of its accesses to require DRAM accesses.

Figure 2 shows the results on all four platforms we tested. Note first that, compared to the all-bank attacks (ABr and ABw), the single-bank attacks (SBr and SBw) are significantly more effective in delaying the victim, as expected. This is despite the fact that the single-bank attacks generate significantly *less* bandwidth. For example, on the Pi 5, SBw attackers cause the highest slowdown of the victim—88×—while generating the least amount of memory bandwidth, less than 440 MB/s.[6] In fact, among the four attackers we tested, the least effective one (ABr) consumes the highest memory bandwidth, while the most effective one (SBw) consumes the least—exactly the *opposite* of conventional wisdom.

### C. Effects on Real-world Applications

In this experiment, we use the same experimental setup, but use different victims from a set of real-world benchmarks: two matrix multiplication kernels [27] and five workloads from SD-VBS [28] (with *fullhd* input). The two matrix multiplication kernels differ in their access pattern. The *mm-opt0* is a naïve implementation, which suffers from poor locality. In contrast, *mm-opt1* is an optimized version with improved spatial locality along the innermost loop. The SD-VBS benchmarks represent diverse memory access patterns.

Figure 3 shows the results. Across all platforms and benchmarks, SB attacks—especially *SBw*—consistently cause extremely high victim slowdowns despite consuming the least memory bandwidth (we omit the attacker-bandwidth figure here because it is similar to Figure 2b). Although the degree of effectiveness varies across platforms due to microarchitectural differences and across benchmarks due to differences in memory-access characteristics, the overall trends are consistent with those observed for the synthetic victim in the previous experiment (Section IV-B), further supporting our hypothesis.

[6]Write bandwidth observed from the application. It does not account for write-induced cache-line refill read traffic. The combined read and write traffic at the DRAM is still upper-bounded by the guaranteed bandwidth.

These results present a major challenge for system designers seeking to provide real-time guarantees using existing memory bandwidth regulation techniques, which limit (throttle) memory bandwidth without distinguishing how that bandwidth is distributed across DRAM banks. To provide worst-case performance guarantees, one must assume the worst-case scenario in which traffic is concentrated on a single bank. However, doing so requires the bandwidth budget to be extremely low—below the guaranteed bandwidth—thereby wasting much of the available memory bandwidth in typical cases where traffic is more evenly distributed across banks. We argue that this limitation arises fundamentally because all existing bandwidth regulation techniques are *bank-oblivious*, which we aim to address in the following.

## V. Per-Bank DRAM Bandwidth Regulation

As discussed earlier, all state-of-the-art memory bandwidth regulators operate in a bank-oblivious manner, which we call the *all-bank* approach. This approach treats memory as a monolithic structure, ignoring the parallel, banked organization of modern main memory.

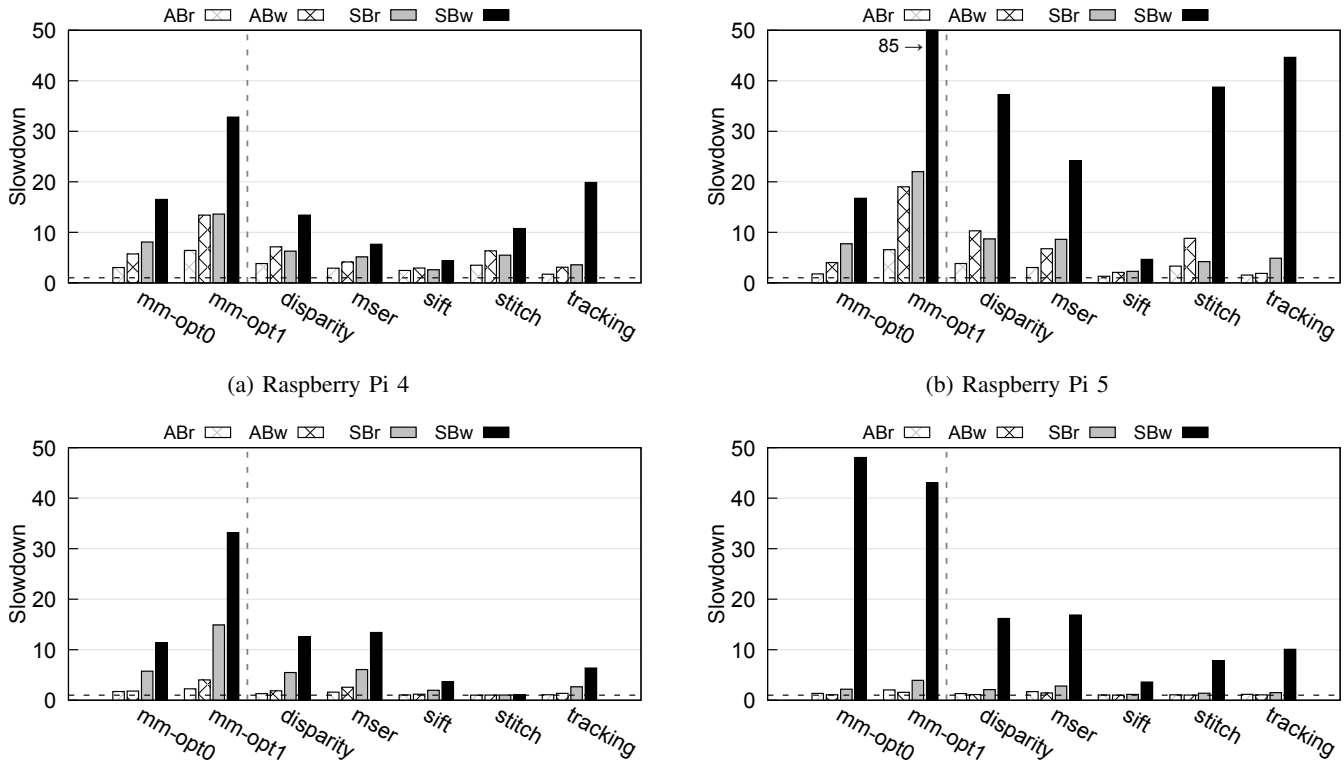
In contrast, we propose a *per-bank* approach, which regulates accesses to each DRAM bank individually. In the worst-case scenario—i.e., when traffic is concentrated on a single bank—the per-bank regulation approach can provide the same isolation guarantees as the all-bank regulation (given the same bandwidth budget). At the same time, it greatly improves overall memory throughput in typical cases where traffic is more evenly distributed across banks, since the same budget, $B_{\text{per-bank}}$, applies to each individual bank. As a result, the maximum usable bandwidth, $BW_{\text{max}}$, scales linearly with the number of banks $N_{\text{bank}}$ as follows:

$$BW_{\text{max}} = B_{\text{per-bank}} \times N_{\text{bank}} \tag{2}$$

In the following, we present the design of our proposed per-bank DRAM bandwidth regulator.

### A. Regulator Placement

We first start by discussing where such a regulator could be placed. Ideally, DRAM bandwidth should be controlled




(a) Raspberry Pi 4

(b) Raspberry Pi 5

(c) Intel Coffee Lake

(d) Jetson Orin AGX

Fig. 3: Victim (X-axis) slowdown caused by all-bank (AB) vs. single-bank (SB) attacks.

by regulating the rate at which LLC misses are served to memory—between the LLC and the memory controller.

A simple option is a drop-in module, similar to BRU [15], [29] for cache bandwidth regulation, placed on the interconnect between the LLC and memory controller. However, we find this to be suboptimal for DRAM regulation since the regulator's clients are the shared LLC cache banks, not the cores. Stalling an entire LLC bank due to one core exceeding its budget would block unrelated requests from other cores, severely degrading performance.

Adding request queues between the LLC and memory controller can potentially address such unnecessary stalls, but this adds significant area overhead, especially when regulating both reads and writes.

To avoid these issues, we place our regulator inside the shared LLC, where it controls when Miss Status Holding Registers (MSHRs) are allowed to issue requests to memory. This enables per-core, per-bank bandwidth regulation without requiring additional queuing structures.

### B. Regulation Scheme

Our regulator uses a fixed-rate regulation scheme similar to MemGuard [1]. This means that the regulator operates with a fixed regulation period (in cycles) and a per-period access budget. Regulation works by counting the number of memory accesses issued by a core within the current period. If that count exceeds the assigned access budget, all subsequent requests from the core are prevented from reaching memory. The counters are reset at the beginning of each new period, effectively replenishing the budgets and allowing cache misses to proceed as usual. The per-bank bandwidth budget, $B_{\text{per-bank}}$, can be calculated as follows:

$$B_{\text{per-bank}} = \frac{N_{\text{acc}}}{P} \times G \times f \tag{3}$$

where $N_{\text{acc}}$ is the number of accesses per period $P$ (in cycles), $G$ is the access granularity (in bytes, typically a 64-byte cache line), and $f$ is the frequency of the clock domain in which the regulator resides, which in our current design is the frequency of the LLC. The choice of the fixed-rate regulation scheme also informs the rest of our design.

### C. Regulator Design

Our design adopts regulation domains, as in [29], which enables arbitrary grouping of cores to be regulated together as a group. Therefore, each regulation domain has its own unique budget configuration register. The number of domains can be configured at design synthesis.

Our regulator requires a way to attribute accesses to cores. For this, our design includes a "tagging unit" that sits immediately after the cores. This unit exposes an MMIO interface to configure which cores are in which regulation domain. The tagging unit adds the domain information to each request that is sent from the cores to the shared LLC. With this domain

information, the regulation unit can attribute LLC misses to the correct counters and selectively enable or disable regulation for a given domain.

For regulation configuration, we expose a set of MMIO control registers. A dedicated register is used to set the global regulation period $P$, while another set of registers configures the access budget, $N_{\text{acc}}$, for each domain during the period.

## VI. Implementation

We implement our design in a RISC-V Rocket Chip SoC [30] using the Chipyard [5] framework. Figure 4 offers a high-level view of our design integrated into this SoC context. This section discusses the implementation details of our work.

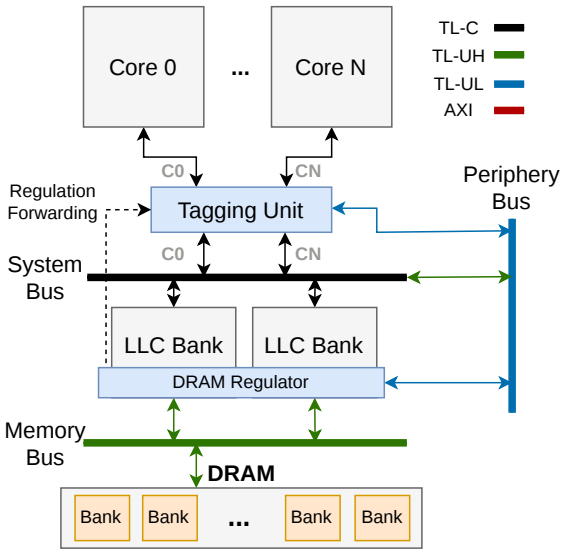



Fig. 4: Rocket SoC with our DRAM bandwidth regulator

### A. Cache Integration

The LLC available in Chipyard is the *Rocket Chip inclusive cache* [31]. As the name suggests, it is an inclusive cache with a configurable number of banks and serves as the shared LLC in our simulated SoC. Supporting directory-based coherence, this cache is implemented as a hierarchical set of modules, with a top-level module wrapping the cache banks, and each cache bank containing its own scheduler, MSHRs, directory, and related components.

In our design, the top-level module of the LLC contains the MMIO interface and counters for regulation. We count TileLink *AcquireBlock* requests (reads) sent to main memory and attribute them to the appropriate domain and bank counters. When a bank access counter exceeds the configured budget for a domain, a throttle signal is asserted and propagated to the cache schedulers. Within the schedulers, this throttle signal is used by the round-robin arbiter that selects which MSHR will service its request next. By default, the arbiter considers an MSHR schedulable if all the cache resources it requires are available. We modify this scheduling decision to also gate schedulability based on the throttle signals.

### B. Regulation Forwarding

Stalling MSHRs enables precise control over the bandwidth that a core can send to the shared main memory. This mechanism alone, however, is insufficient, as residual contention can still occur within the cache banks themselves. To address this issue, we enable regulation of cache bandwidth via throttle signals forwarded from the DRAM regulator up to the tagging unit (see Figure 4). With these forwarded signals, the tagging unit can control cache bandwidth by stalling *AcquireBlock* requests from cores in a throttled domain when the corresponding throttle signal is asserted. This set of signals consists of $D \times N_{\text{bank}}$ bits, where $D$ is the number of domains and $N_{\text{bank}}$ is the number of DRAM banks.

## VII. Evaluation

In this section, we evaluate the proposed per-bank regulator in terms of its worst-case guarantees and average throughput.

### A. Experimental Setup

We use FireSim [7] as our simulation environment. FireSim is an FPGA-accelerated, cycle-exact, full-system simulator. It enables simulation of a complete SoC, with the FPGA operating at 100 MHz while the simulated target runs at a configurable frequency. Simulation results are not affected by the FPGA speed limitation, as a specialized compiler infrastructure decouples the target design from the FPGA host [32], enabling realistic performance evaluation.

| | |
|---|---|
| Cores | 4×BOOM, 1GHz, out-of-order, 2-wide, ROB: 64, LSQ: 16/16, L1: 16K(I)/16K(D), 6 mshrs |
| Shared L2 Cache | 1MB (16-way), 2 banks, 27 mshrs per bank, random replacement |
| DRAM | 4GB 1-rank 8-bank DDR3, FR-FCFS, Bank map: 9, 10, 11 (direct map); tRC = 47ns |

TABLE III: Evaluation platform specifications.

Our simulation configuration is shown in Table III[7]. The SoC consists of four Berkeley Out-of-Order Machine (BOOM) [33] cores in their medium (2-wide) configuration. We include the *Rocket Chip inclusive cache* [31] in the design, which serves as a 1 MB shared last-level cache (LLC). The entire SoC is configured to operate at 1 GHz. For main memory, FireSim utilizes the FASED memory controller which uses a memory timing model to bridge the gap between FPGA and real DDR backing memory [32]. We configure it to model a single-channel, single-rank DDR3 memory system with an FR-FCFS scheduling policy. All simulations run on FireSim, running real RISC-V Linux kernel v6.2. We patch the kernel with PALLOC [34] to enable LLC set partitioning.

[7]We choose address mapping functions for LLC sets and DRAM banks to enable LLC set partitioning without DRAM bank co-partitioning. We also provision enough LLC MSHRs to avoid MSHR contention [26].

### B. FASED DRAM Controller Enhancements

To effectively evaluate DRAM contention, it is important that the memory model in the simulation environment accurately represents real hardware. The baseline FASED memory controller [32] implements a unified transaction queue, meaning that both read and write transactions are buffered together and sent to the command scheduler in FIFO order. While simple to implement, as discussed in Section II-A, this design is unrealistic, as most modern memory controllers implement separate read and write buffers and watermark based batched scheduling [13] to improve performance.

We enhance the FASED memory controller by introducing separate transaction queues and write batching with parameterizable watermarks. To demonstrate the impact of these enhancements, we run a random-access, write-heavy workload (*PLL*) on a simulated SoC. One run uses the baseline FASED controller, and the other uses our enhanced version. We then collect the number of bus mode switches that occur during the execution of the benchmark.

| **FASED Version** | **Mode Switches** |
|---|---|
| Baseline | 1,813,936 |
| Ours (w/ Batching) | 577,155 |

TABLE IV: Write-batching impact on bus mode switches.

Table IV presents the results, showing a 3.14× reduction in mode switches when using the write-batching–enabled version of the model. Our enhancements enable more realistic memory performance evaluation, which we plan to release as open-source.

### C. Guaranteed Bandwidth in FireSim

We begin our evaluation by exploring the guaranteed DRAM bandwidth in our simulated SoC. Following Eq. 1, we first calculate the theoretical guaranteed bandwidth of the simulated DRAM system. For validation, we also run the same PLL workload on FireSim, targeting a single DRAM bank, as described in Section III.

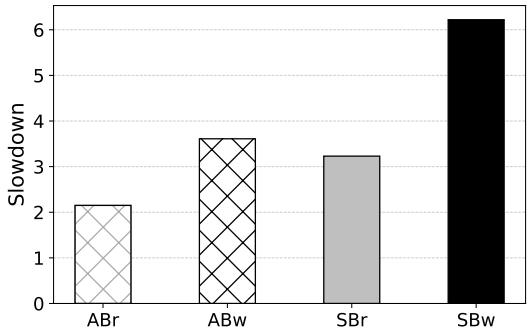
| | |
|---|---|
| Theory | 1362 |
| Measured | 1271 |

TABLE V: Guaranteed memory bandwidth (MB/s) in FireSim.

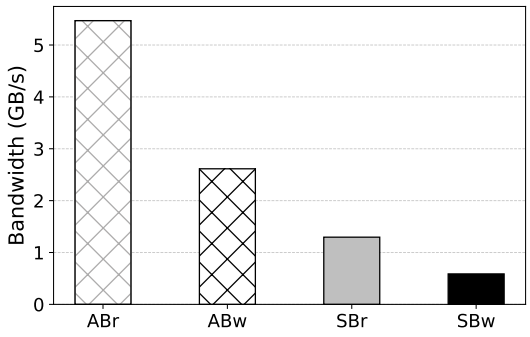
Table V shows the results. As can be seen, the measured bandwidth closely matches the expected theoretical number. These results validate that the FASED memory model offers a realistic evaluation platform.

### D. Effects of DRAM Bank Contention

In this experiment, we repeat the all-bank and single-bank attack experiment against a synthetic victim benchmark as described in Section IV-B—that is, we use *Bandwidth* from IsolBench [26] as the *victim* and use *ABr* (all-bank read), *ABw* (all-bank write), *SBr* (single-bank read), and *SBw* (single-bank write) as the attackers to understand the attackers' impact on the victim as well as the bandwidth usage of the attackers on our simulated RISC-V SoC platform.




(a) Victim Slowdown.




(b) Attacker Bandwidth.

Fig. 5: Effects of all-bank (AB) and single-bank (SB) DRAM bank attacks against the synthetic victim on FireSim.

Figure 5 shows the results. Note that the trends observed here closely mirror those from the real platform experiments (cf. Figure 2). Specifically, *ABr* causes the least slowdown (∼2.1×) to the victim while generating the highest memory bandwidth (>5 GB/s), whereas *SBw* causes the greatest slowdown (∼6.2×) yet consumes the least bandwidth (<1 GB/s).

### E. Effects of Per-Bank Bandwidth Regulation

In this subsection, we evaluate the performance of the proposed *per-bank regulator*. For comparison, we also implement an *all-bank regulator*, which is a modified version of our regulator where each domain maintains a single global access counter, instead of per-bank counters as in our proposed design. This effectively treats DRAM as a monolithic structure, similar to existing memory bandwidth regulation solutions.

In both regulators, we set the regulation period to *1 ms* with a bandwidth budget of 53 MB/s, which we empirically (sweeping the budgets) determined to be the maximum budget needed to bound the worst-case victim slowdown to an acceptable level (less than 10% of the solo execution time). We configure the four cores into two regulation domains: *real-time* and *best-effort*. The real-time domain includes only Core 0 and remains unregulated to ensure maximum real-time

performance. The best-effort domain includes the remaining three cores (Cores 1–3) and is regulated using the above settings to keep their interference to the real-time task within bounded limits. The LLC space is evenly partitioned between the real-time and best-effort domains, using PALLOC [34], to avoid cache evictions between the two domains.

First, we examine the regulators' ability to provide isolation guarantees to the task in the real-time domain against interference from the all-bank and single-bank write attackers—*ABw* and *SBw*, respectively—on the best-effort domain. The rest of the experimental setup is identical to the previous one, except that the attackers are now regulated.

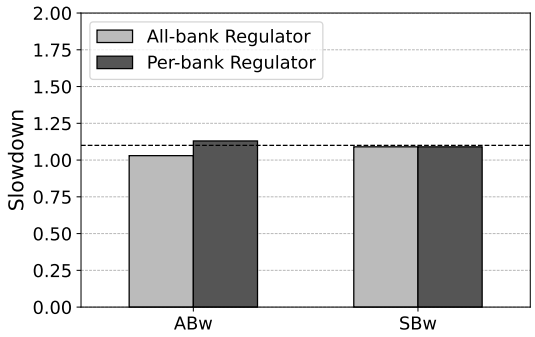



Fig. 6: Victim slowdown under all-bank vs. per-bank regulation against *ABw* and *SBw* attackers.

Figure 6 shows the results. Both the all-bank and per-bank regulators limit the victim's slowdown to 1.1× the solo execution time against the single-bank write attack (*SBw*). Against the all-bank write attacker (*ABw*), however, our per-bank regulator yields a slightly higher slowdown (1.13×) than in the single-bank case, whereas the all-bank regulator limits the slowdown to 1.03×. Relative to the 1.1× baseline, this means that the per-bank regulator incurs only a 3% additional overhead due to delays outside the DRAM banks, while enabling the *ABw* attacker—and any benign best-effort task—to achieve up to an 8× performance improvement over the all-bank regulator. We consider this a highly favorable tradeoff. We now turn to empirical results demonstrating the throughput gains of our per-bank regulator.

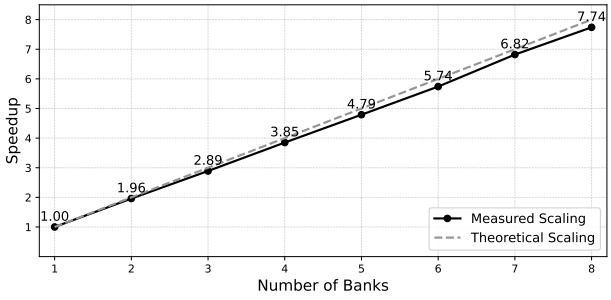



Fig. 7: Per-bank regulation scaling.

Figure 7 shows the speedup of the regulated attacker tasks on the best-effort domain as a function of the number of banks of the DDR3 memory. The results clearly show that, as the number of DRAM banks increases, the observed speedup linearly increases, confirming that throughput scaling as predicted by Eq. 2. Although it does not achieve the perfect linear scaling (at 8 banks, the speedup was 7.74×) because accesses to DRAM banks are not completely parallel, this shows the potential performance benefits of the per-bank approach.

The additional bandwidth provided by our per-bank regulation approach can be especially beneficial when real-world applications—unlike the specially engineered attackers used above—are executed on the best-effort domain. To demonstrate this, we use a set of representative real-world workloads: two matrix multiplication kernels and five benchmarks from SD-VBS [28] (with *fullhd* input), as described in Section IV-C. Each workload is executed on a single core in the *best-effort* domain. We measure its runtime under both all-bank and per-bank regulation and compare the results against the unregulated baseline.

Figure 8 shows the results. As expected, our per-bank regulator achieves significant gains over the all-bank regulator across all benchmarks. Compute-intensive workloads such as *sift* predictably show smaller improvements, whereas memory-intensive workloads such as *disparity* show substantial gains. Overall, the per-bank regulator achieves an average performance improvement of 5.74× over the all-bank regulator.

In summary, we have shown that per-bank memory bandwidth regulation can provide isolation benefits comparable to those of all-bank regulation while offering substantially higher, linearly scalable memory performance benefits. As the parallelism in modern DRAM technologies continues to increase (for instance, HBM memories have thousands of DRAM banks), we believe the potential benefits of the per-bank approach will become even more pronounced.

### F. Implementation Overhead

To quantify the implementation cost of our design, we synthesize a quad-core BOOM SoC with and without our regulator using Cadence Genus with Hammer [35] automation, targeting the ASAP7 7nm technology node [36]. In addition to the default 8-bank DRAM configuration, we also evaluate a 16 bank DRAM configuration to evaluate scalability.

| # Banks | Area | Timing |
|---|---|---|
| 8 | 0.35% | 3% |
| 16 | 0.47% | 3% |

TABLE VI: Area and timing overhead of per-bank regulator.

Table VI shows the results. Compared to the baseline, our per-bank regulator incurs minimal area overhead (0.34-0.47%) and modest timing overhead, reducing the maximum achievable clock frequency by 3% across both bank configurations.

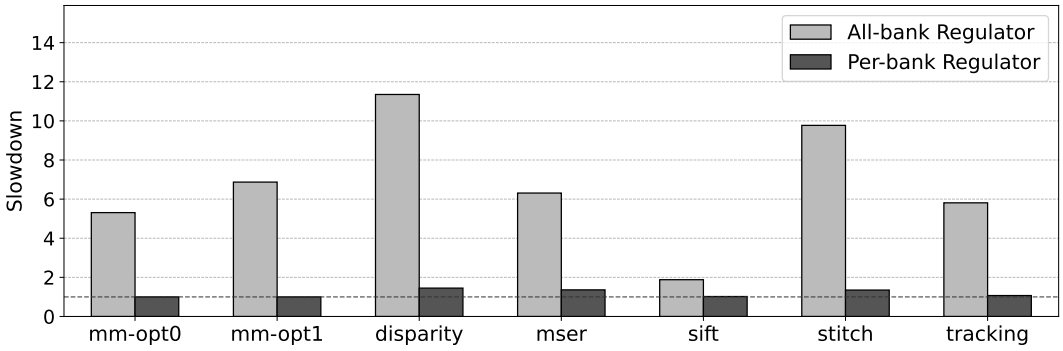



Fig. 8: Average case (benign best-effort tasks) performance.

## VIII. Discussion

In this section, we discuss how per-bank bandwidth regulation can be integrated into existing industry QoS standards and explore its broader implications and use cases.

Intel's Resource Director Technology (RDT) [37], ARM's Memory Partitioning and Monitoring (MPAM) [38], and, more recently, RISC-V's Capacity/Bandwidth QoS Register Interface (CBQRI) [39] are industry standards for resource partitioning and bandwidth regulation. Each provide similar abstractions and enforcement interfaces for x86, ARM, and RISC-V architectures, respectively. Together, these standards illustrate the growing architectural support for hardware-assisted cache and memory QoS across modern multicore systems—a welcome development for real-time systems.

However, when it comes to memory bandwidth regulation, none of these standards provide abstractions for controlling individual DRAM banks. This is problematic for real-time systems, where worst-case performance guarantees are of paramount importance. Without bank-level control, system designers face two undesirable options: accept extremely weak worst-case guarantees or adopt heavy-handed throttling that sacrifices the high throughput potential of modern DRAM technologies, as demonstrated in this work.

Fortunately, we believe this limitation can be addressed without extensive changes to existing standards. The proposed per-bank regulation approach does not require distinct bandwidth budgets for each bank. Thus, the existing bandwidth budget specified in these standards can simply be *interpreted* as a per-bank budget rather than a system-wide budget. In other words, per-bank regulation could be realized entirely within the implementation of these standards—without modifying their architectural interfaces—provided that vendors clearly document the semantics of the bandwidth budget.

The benefits of per-bank regulation extend beyond real-time systems. For example, a recent work [40] proposes bandwidth regulation as a defense mechanism against RowHammer attacks [41]. The authors explicitly highlight the need for per-bank regulation to constrain bank-level activation rates, directly addressing the root cause of the problem while avoiding the substantial performance penalties associated with existing approaches. This example illustrates the broader utility of per-bank regulation well beyond real-time systems.

## IX. Related Work

Timing unpredictability caused by memory interference in shared-memory multicores has long been a major challenge for real-time systems. Over the past decade, the real-time systems community has extensively studied this problem. These efforts include developing memory-interference analysis models [42]–[46], proposing predictable memory controllers [47]–[51], exploring DRAM bank partitioning [34], [52]–[55], and bandwidth regulation mechanisms [1]–[3], [56]–[60].

Among these efforts, memory bandwidth regulation has been one of the most actively explored approaches. Software-based mechanisms, in particular, have been popular because they are easy to deploy and widely applicable to existing COTS multicore platforms. Since MemGuard [1] first introduced a performance-counter-based regulation mechanism, numerous follow-up works have expanded the design space. BWLOCK [56] provided a fine-grained API for application-level bandwidth control. Xu et al. [57] integrated cache partitioning and bandwidth regulation into a unified resource-allocation framework. Saeed et al. [58] proposed a dynamic regulation approach driven by DRAM-controller feedback. MemPol [2] explored a polling-based mechanism using a dedicated real-time microcontroller. While versatile, software-based bandwidth-regulation mechanisms suffer from relatively high overhead and limited regulation granularity.

To address these limitations, researchers have also explored hardware-based bandwidth regulators. BRU [29] introduced a drop-in regulation unit that controls traffic between CPU cores and the shared LLC. MemCoRe [3] proposed a coherence-based hardware regulator for Xilinx PS+PL systems. AXI-REALM [61] extended bandwidth regulation to heterogeneous SoCs by controlling accelerator traffic at the AXI interconnect.

As discussed in Section VIII, major industry players have also begun incorporating architectural support for shared-resource management, including DRAM-bandwidth regulation. Intel's RDT [37] was the first to provide such mechanisms, offering memory-bandwidth monitoring (MBM) and memory-bandwidth allocation (MBA). MBA exposes a set of

predefined throttling levels, but their semantics are not precisely documented and have evolved across processor generations. Unfortunately, MBA has been shown to be insufficient for achieving the isolation guarantees required by real-time workloads, even at its most restrictive configuration [62].

More recently, ARM's MPAM [63] and RISC-V's CBQRI [39] have introduced architectural specifications with similar goals. MPAM provides additional capabilities such as minimum/maximum bandwidth limits, and CBQRI likewise enables resource tagging and bandwidth control within the RISC-V ecosystem. However, unlike Intel RDT, both MPAM and CBQRI are still in the early stages of adoption, limiting their practical impact to date. Nevertheless, once they are broadly deployed, these hardware-based mechanisms have strong potential to become viable tools for real-time systems.

However, existing software- and hardware-based regulators are generally unaware of the banked, parallel organization of modern memory systems, which are often mapped using sophisticated XOR-based schemes to distribute load and avoid hotspots [18]. Therefore, as demonstrated in this work, their worst-case effectiveness is limited. Within the security community, the importance of banked memory organization has long been recognized because of its security implications, such as Rowhammer [40], [41]. Consequently, many reverse-engineering tools have been proposed and used [19], [64], [65], although most of these work has focused on x86-based Intel and AMD platforms. Within the real-time systems community, reverse engineering of DRAM bank mapping on ARM-based platforms has gained prominence for analyzing and mitigating memory contention [25], [66]. However, most prior efforts on ARM platforms have assumed simple direct mappings. In contrast, our enhancements to DRAMA [19] enable reverse engineering of DRAM bank mappings on both x86 and ARM platforms and have successfully uncovered sophisticated XOR-based bank-mapping schemes, thereby enabling bank-aware contention analysis and mitigation.

We believe that our proposed per-bank bandwidth-regulation approach can be implemented in a manner compatible with these existing architectural standards, and that it offers significant performance benefits. Broader adoption of architectural QoS mechanisms like the one we propose here will be crucial for enabling predictable and high-performance real-time systems on future multicore and heterogeneous platforms.

## X. Conclusion

In this work, we presented a per-bank memory bandwidth regulator that enables predictable and high-performance real-time systems. Using improved DRAM bank-mapping reverse-engineering tools, we characterized the guaranteed bandwidth of modern platforms and showed that worst-case DRAM contention remains consistent across DDR generations. Building on this insight, we demonstrated a single-bank contention attack, revealing that interference peaks when attackers target the same bank and that higher aggregate bandwidth does not necessarily increase contention. Motivated by these findings, we designed and implemented a per-bank bandwidth regulator that enforces domain isolation while preserving system throughput. Our evaluation demonstrates strong temporal isolation and a 5.74× average throughput improvement over all-bank regulation. Overall, per-bank regulation is an effective and practical strategy for mitigating interference in shared DRAM systems.

## Acknowledgments

This research is supported in part by NSF grants CPS-2038923 and CCF-2403013. Connor Rudy Sullivan is supported by the Madison and Lila Self Graduate Fellowship at the University of Kansas.

## References

[1] 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.

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

[3] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo, and R. Mancuso, "Coherence-aided memory bandwidth regulation," in *RTSS*, 2024.

[4] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, 2015.

[5] 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," in *Micro*, 2020.

[6] "AMD Virtex UltraScale+ VCU118 FPGA," https://www.xilinx.com/products/boards-and-kits/vcu118.html, 2024, accessed: 2024-05-13.

[7] 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.

[8] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann,, 2010.

[9] JEDEC, "DDR4 SDRAM STANDARD." [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4a

[10] ——, "DDR5 SDRAM." [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-5c01

[11] ——, "Low Power Double Data Rate 4 (LPDDR4)." [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd209-4b

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

[13] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged reads: Mitigating the impact of dram writes on dram reads," in *HPCA*, 2012.

[14] Intel, "Intel Resource Director Technology (RDT) Framework," https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[15] C. Sullivan, A. Manley, M. Alian, and H. Yun, " Per-Bank Bandwidth Regulation of Shared Last-Level Cache for Real-Time Systems ," in *RTSS*, 2024.

[16] JEDEC, "High Bandwidth Memory (HBM)." [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[17] A. Pradhan, D. Ottaviano, Y. Jiang, H. Huang, J. Zhang, A. Zuepke, A. Bastoni, and M. Caccamo, "Predictable memory bandwidth regulation for dynamiq arm systems," in *RTCSA*, 2025.

[18] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," in *MICRO*, 2000.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting dram addressing for cross-cpu attacks," in *USENIX*, 2016.

[20] R. Lidl and H. Niederreiter, *Finite Fields*, 2nd ed. Cambridge University Press, 1997.

[21] "MT53D512M32D2DS-046-AAT-D LPDDR4 part detail." [Online]. Available: https://www.micron.com/products/memory/dram-components/lpddr4/part-catalog/part-detail/mt53d512m32d2ds-046-aat-d

[22] "Raspberry Pi computer hardware - Raspberry Pi Documentation." [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[23] J. Greeling, "Raspberry Pi boosts Pi 5 performance with SDRAM tuning." [Online]. Available: https://www.jeffgeerling.com/blog/2024/raspberry-pi-boosts-pi-5-performance-sdram-tuning

[24] "K4UBE3D4AB-MGCL datasheet." [Online]. Available: https://www.lcsc.com/product-detail/C5827966.html

[25] M. Bechtel and H. Yun, "Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems," *IEEE Transactions on Computers*, 2021.

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

[27] "Matrix Multiplication Microbenchmark," https://github.com/CSL-KU/matmult.

[28] 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.

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

[30] 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

[31] "Rocket Chip SoC Inclusive Cache Generator," https://github.com/chipsalliance/rocket-chip-inclusive-cache, 2025, accessed: 2025-11-12.

[32] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanovic, "Fased: Fpga-accelerated simulation and evaluation of dram," in *FPGA*, 2019.

[33] 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

[34] 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.

[35] 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.

[36] 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

[37] 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

[38] 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

[39] RISC-V CBQRI Task Group, "RISC-V Capacity and Bandwidth QoS Register Interface (CBQRI) Specification," RISC-V International, Specification Version 1.0-rc3, 2024, release candidate 3 (rc3). [Online]. Available: https://github.com/riscv-non-isa/riscv-cbqri/releases/download/v1.0-rc3/riscv-cbqri.pdf

[40] C. Fiedler, J. Juffinger, S. R. Neela, M. Heckel, H. Weissteiner, A. G. Yağlıkçı, F. Adamsky, and D. Gruss, "Memory Band-Aid: A Principled Rowhammer Defense-in-Depth," in *NDSS*, 2026.

[41] 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," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014.

[42] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *DATE*, 2010.

[43] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *RTAS*, 2014.

[44] H. Yun, R. Pellizzoni, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *ECRTS*, 2015.

[45] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[46] ——, "Analysis of memory-contention in heterogeneous cots mpsocs," in *ECRTS*, 2020.

[47] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *CODES+ISSS*, 2011, pp. 99–108.

[48] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *ECRTS*. IEEE, 2014.

[49] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *RTSS*. IEEE, 2015, pp. 53–64.

[50] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *RTAS*. IEEE, 2017.

[51] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A comparative study of predictable dram controllers," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 2, 2018.

[52] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

[53] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *CSE*. IEEE, 2013.

[54] X. Pan and F. Mueller, "Numa-aware memory coloring for multicore real-time systems," *Journal of Systems Architecture*, vol. 118, 2021.

[55] H. Yang, R. Shao, Y. Cheng, Y. Chen, R. Zhou, G. Liu, G. Xie, and Q. Zhou, "Redb: Real-time enhancement of docker containers via memory bank partitioning in multicore systems," *Journal of Systems Architecture*, vol. 151, 2024.

[56] W. Ali and H. Yun, "Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms," in *ECRTS*, 2018.

[57] 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*. IEEE, 2019.

[58] 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.

[59] B. Sun, Z. Wei, A. Bastoni, D. Roy, M. Theile, T. Kloda, R. Pellizzoni, and M. Caccamo, "Multi-objective memory bandwidth regulation and cache partitioning for multicore real-time systems," in *ECRTS*, 2025.

[60] H. Park, J. Lee, H. Lee, T. Kwon, W. Choi, S. Moon, and C.-G. Lee, "A field practical approach to memory bandwidth allocation for consolidating multi-domain automotive applications on a single soc," in *RTAS*. IEEE, 2025.

[61] T. Benz, A. Ottaviano, R. Balas, A. Garofalo, F. Restuccia, A. Biondi, and L. Benini, "AXI-REALM: A lightweight and modular interconnect extension for traffic regulation and monitoring of heterogeneous real-time socs," in *DATE*. IEEE, 2024.

[62] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A Closer Look at Intel Resource Director Technology (RDT)," in *RTNS*, 2022.

[63] ARM, "Arm Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification," Arm Ltd., Specification IHI 0099A.a, 2023, issue A.a. [Online]. Available: https://developer.arm.com/documentation/ihi0099/aa/

[64] M. Wi, S. Baek, S. Park, M. Erez, and J. H. Ahn, "Sudoku: Decomposing dram address mapping into component functions," *arXiv preprint arXiv:2506.15918*, 2025.

[65] M. Heckel, F. Adamsky, J. Juffinger, F. Rauscher, and D. Gruss, "Verifying dram addressing in software," in *ESORICS*, 2025.

[66] A. Stevanato, M. Zini, A. Biondi, B. Morelli, and A. Biasci, "Learning memory-contention timing models with automated platform profiling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, 2024.